

Free Component Library (FCL):  
Reference guide.

---

Reference guide for FCL units.  
Document version 3.2.0  
February 2021

Michaël Van Canneyt

---

# Contents

0.1	Overview	87
<b>1</b>	<b>Reference for unit 'ascii85'</b>	<b>88</b>
1.1	Used units	88
1.2	Overview	88
1.3	Constants, types and variables	88
1.3.1	Types	88
1.4	TASCII85DecoderStream	89
1.4.1	Description	89
1.4.2	Method overview	89
1.4.3	Property overview	89
1.4.4	TASCII85DecoderStream.Create	89
1.4.5	TASCII85DecoderStream.Decode	90
1.4.6	TASCII85DecoderStream.Close	90
1.4.7	TASCII85DecoderStream.ClosedP	90
1.4.8	TASCII85DecoderStream.Destroy	90
1.4.9	TASCII85DecoderStream.Read	91
1.4.10	TASCII85DecoderStream.Seek	91
1.4.11	TASCII85DecoderStream.BExpectBoundary	91
1.5	TASCII85EncoderStream	91
1.5.1	Description	91
1.5.2	Method overview	92
1.5.3	Property overview	92
1.5.4	TASCII85EncoderStream.Create	92
1.5.5	TASCII85EncoderStream.Destroy	92
1.5.6	TASCII85EncoderStream.Write	92
1.5.7	TASCII85EncoderStream.Width	93
1.5.8	TASCII85EncoderStream.Boundary	93
1.6	TASCII85RingBuffer	93
1.6.1	Description	93
1.6.2	Method overview	93

1.6.3	Property overview	93
1.6.4	TASCII85RingBuffer.Write	94
1.6.5	TASCII85RingBuffer.Read	94
1.6.6	TASCII85RingBuffer.FillCount	94
1.6.7	TASCII85RingBuffer.Size	94
<b>2</b>	<b>Reference for unit 'AVL_Tree'</b>	<b>95</b>
2.1	Used units	95
2.2	Overview	95
2.3	Constants, types and variables	95
2.3.1	Types	95
2.3.2	Variables	96
2.4	TAVLTree	96
2.4.1	Description	96
2.4.2	Method overview	97
2.4.3	Property overview	98
2.4.4	TAVLTree.Create	98
2.4.5	TAVLTree.CreateObjectCompare	98
2.4.6	TAVLTree.Destroy	98
2.4.7	TAVLTree.SetNodeManager	98
2.4.8	TAVLTree.NewNode	99
2.4.9	TAVLTree.DisposeNode	99
2.4.10	TAVLTree.Add	99
2.4.11	TAVLTree.AddAscendingSequence	100
2.4.12	TAVLTree.Delete	100
2.4.13	TAVLTree.Remove	100
2.4.14	TAVLTree.RemovePointer	101
2.4.15	TAVLTree.MoveDataLeftMost	101
2.4.16	TAVLTree.MoveDataRightMost	101
2.4.17	TAVLTree.Clear	101
2.4.18	TAVLTree.FreeAndClear	102
2.4.19	TAVLTree.FreeAndDelete	102
2.4.20	TAVLTree.Equals	102
2.4.21	TAVLTree.IsEqual	102
2.4.22	TAVLTree.Assign	103
2.4.23	TAVLTree.Compare	103
2.4.24	TAVLTree.Find	103
2.4.25	TAVLTree.FindKey	103
2.4.26	TAVLTree.FindNearestKey	104
2.4.27	TAVLTree.FindSuccessor	104

2.4.28	TAVLTree.FindPrecessor	104
2.4.29	TAVLTree.FindLowest	105
2.4.30	TAVLTree.FindHighest	105
2.4.31	TAVLTree.FindNearest	105
2.4.32	TAVLTree.FindPointer	105
2.4.33	TAVLTree.FindLeftMost	106
2.4.34	TAVLTree.FindRightMost	106
2.4.35	TAVLTree.FindLeftMostKey	106
2.4.36	TAVLTree.FindRightMostKey	106
2.4.37	TAVLTree.FindLeftMostSameKey	107
2.4.38	TAVLTree.FindRightMostSameKey	107
2.4.39	TAVLTree.GetEnumerator	107
2.4.40	TAVLTree.GetEnumeratorHighToLow	107
2.4.41	TAVLTree.ConsistencyCheck	108
2.4.42	TAVLTree.WriteReportToStream	108
2.4.43	TAVLTree.NodeToReportStr	108
2.4.44	TAVLTree.ReportAsString	108
2.4.45	TAVLTree.OnCompare	109
2.4.46	TAVLTree.OnObjectCompare	109
2.4.47	TAVLTree.NodeClass	109
2.4.48	TAVLTree.Root	109
2.4.49	TAVLTree.Count	110
2.5	TAVLTreeNode	110
2.5.1	Description	110
2.5.2	Method overview	110
2.5.3	TAVLTreeNode.Successor	110
2.5.4	TAVLTreeNode.Precessor	110
2.5.5	TAVLTreeNode.Clear	111
2.5.6	TAVLTreeNode.TreeDepth	111
2.5.7	TAVLTreeNode.ConsistencyCheck	111
2.5.8	TAVLTreeNode.GetCount	111
2.6	TAVLTreeNodeEnumerator	111
2.6.1	Description	111
2.6.2	Method overview	112
2.6.3	Property overview	112
2.6.4	TAVLTreeNodeEnumerator.Create	112
2.6.5	TAVLTreeNodeEnumerator.GetEnumerator	112
2.6.6	TAVLTreeNodeEnumerator.MoveNext	112
2.6.7	TAVLTreeNodeEnumerator.Current	112
2.6.8	TAVLTreeNodeEnumerator.LowToHigh	113

2.7	TAVLTreeNodeMemManager	113
2.7.1	Description	113
2.7.2	Method overview	113
2.7.3	Property overview	113
2.7.4	TAVLTreeNodeMemManager.DisposeNode	113
2.7.5	TAVLTreeNodeMemManager.NewNode	114
2.7.6	TAVLTreeNodeMemManager.Clear	114
2.7.7	TAVLTreeNodeMemManager.Create	114
2.7.8	TAVLTreeNodeMemManager.Destroy	114
2.7.9	TAVLTreeNodeMemManager.MinimumFreeNode	115
2.7.10	TAVLTreeNodeMemManager.MaximumFreeNodeRatio	115
2.7.11	TAVLTreeNodeMemManager.Count	115
2.8	TBaseAVLTreeNodeManager	115
2.8.1	Description	115
2.8.2	Method overview	116
2.8.3	TBaseAVLTreeNodeManager.DisposeNode	116
2.8.4	TBaseAVLTreeNodeManager.NewNode	116
<b>3</b>	<b>Reference for unit 'base64'</b>	<b>117</b>
3.1	Used units	117
3.2	Overview	117
3.3	Constants, types and variables	117
3.3.1	Types	117
3.4	Procedures and functions	118
3.4.1	DecodeStringBase64	118
3.4.2	EncodeStringBase64	118
3.5	EBase64DecodingException	118
3.5.1	Description	118
3.6	TBase64DecodingStream	118
3.6.1	Description	118
3.6.2	Method overview	119
3.6.3	Property overview	119
3.6.4	TBase64DecodingStream.Create	119
3.6.5	TBase64DecodingStream.Reset	119
3.6.6	TBase64DecodingStream.Read	119
3.6.7	TBase64DecodingStream.Seek	120
3.6.8	TBase64DecodingStream.EOF	120
3.6.9	TBase64DecodingStream.Mode	120
3.7	TBase64EncodingStream	121
3.7.1	Description	121

3.7.2	Method overview	121
3.7.3	TBase64EncodingStream.Destroy	121
3.7.4	TBase64EncodingStream.Flush	121
3.7.5	TBase64EncodingStream.Write	122
3.7.6	TBase64EncodingStream.Seek	122
<b>4</b>	<b>Reference for unit 'BlowFish'</b>	<b>123</b>
4.1	Used units	123
4.2	Overview	123
4.3	Constants, types and variables	123
4.3.1	Constants	123
4.3.2	Types	123
4.4	EBlowFishError	124
4.4.1	Description	124
4.5	TBlowFish	124
4.5.1	Description	124
4.5.2	Method overview	124
4.5.3	TBlowFish.Create	124
4.5.4	TBlowFish.Encrypt	125
4.5.5	TBlowFish.Decrypt	125
4.6	TBlowFishDeCryptStream	125
4.6.1	Description	125
4.6.2	Method overview	125
4.6.3	TBlowFishDeCryptStream.Create	125
4.6.4	TBlowFishDeCryptStream.Read	126
4.6.5	TBlowFishDeCryptStream.Seek	126
4.7	TBlowFishEncryptStream	126
4.7.1	Description	126
4.7.2	Method overview	126
4.7.3	TBlowFishEncryptStream.Destroy	127
4.7.4	TBlowFishEncryptStream.Write	127
4.7.5	TBlowFishEncryptStream.Seek	127
4.7.6	TBlowFishEncryptStream.Flush	127
4.8	TBlowFishStream	128
4.8.1	Description	128
4.8.2	Method overview	128
4.8.3	Property overview	128
4.8.4	TBlowFishStream.Create	128
4.8.5	TBlowFishStream.Destroy	129
4.8.6	TBlowFishStream.BlowFish	129

<b>5</b>	<b>Reference for unit 'BufDataset'</b>	<b>130</b>
5.1	Used units	130
5.2	Overview	130
5.3	Constants, types and variables	130
5.3.1	Types	130
5.4	Procedures and functions	132
5.4.1	RegisterDatapacketReader	132
5.5	TBlobBuffer	133
5.6	TBufBlobField	133
5.7	TBufBookmark	133
5.8	TBufRecLinkItem	133
5.9	TDBCompareRec	133
5.10	TRecUpdateBuffer	134
5.11	TArrayBufIndex	134
5.11.1	Description	134
5.11.2	Method overview	135
5.11.3	TArrayBufIndex.Create	135
5.11.4	TArrayBufIndex.ScrollBackward	135
5.11.5	TArrayBufIndex.ScrollForward	135
5.11.6	TArrayBufIndex.GetCurrent	135
5.11.7	TArrayBufIndex.ScrollFirst	136
5.11.8	TArrayBufIndex.ScrollLast	136
5.11.9	TArrayBufIndex.SetToFirstRecord	136
5.11.10	TArrayBufIndex.SetToLastRecord	136
5.11.11	TArrayBufIndex.StoreCurrentRecord	136
5.11.12	TArrayBufIndex.RestoreCurrentRecord	136
5.11.13	TArrayBufIndex.CanScrollForward	137
5.11.14	TArrayBufIndex.DoScrollForward	137
5.11.15	TArrayBufIndex.StoreCurrentRecIntoBookmark	137
5.11.16	TArrayBufIndex.StoreSpareRecIntoBookmark	137
5.11.17	TArrayBufIndex.GotoBookmark	137
5.11.18	TArrayBufIndex.InitialiseIndex	137
5.11.19	TArrayBufIndex.InitialiseSpareRecord	137
5.11.20	TArrayBufIndex.ReleaseSpareRecord	138
5.11.21	TArrayBufIndex.BeginUpdate	138
5.11.22	TArrayBufIndex.AddRecord	138
5.11.23	TArrayBufIndex.InsertRecordBeforeCurrentRecord	138
5.11.24	TArrayBufIndex.RemoveRecordFromIndex	138
5.11.25	TArrayBufIndex.EndUpdate	138
5.12	TBufBlobStream	139

5.12.1	Description	139
5.12.2	Method overview	139
5.12.3	TBufBlobStream.Create	139
5.12.4	TBufBlobStream.Destroy	139
5.13	TBufDataset	139
5.13.1	Description	139
5.13.2	Property overview	140
5.13.3	TBufDataset.MaxIndexesCount	140
5.13.4	TBufDataset.FieldDefs	140
5.13.5	TBufDataset.Active	141
5.13.6	TBufDataset.AutoCalcFields	141
5.13.7	TBufDataset.Filter	141
5.13.8	TBufDataset.Filtered	141
5.13.9	TBufDataset.ReadOnly	141
5.13.10	TBufDataset.AfterCancel	141
5.13.11	TBufDataset.AfterClose	142
5.13.12	TBufDataset.AfterDelete	142
5.13.13	TBufDataset.AfterEdit	142
5.13.14	TBufDataset.AfterInsert	142
5.13.15	TBufDataset.AfterOpen	142
5.13.16	TBufDataset.AfterPost	142
5.13.17	TBufDataset.AfterScroll	142
5.13.18	TBufDataset.BeforeCancel	143
5.13.19	TBufDataset.BeforeClose	143
5.13.20	TBufDataset.BeforeDelete	143
5.13.21	TBufDataset.BeforeEdit	143
5.13.22	TBufDataset.BeforeInsert	143
5.13.23	TBufDataset.BeforeOpen	143
5.13.24	TBufDataset.BeforePost	143
5.13.25	TBufDataset.BeforeScroll	144
5.13.26	TBufDataset.OnCalcFields	144
5.13.27	TBufDataset.OnDeleteError	144
5.13.28	TBufDataset.OnEditError	144
5.13.29	TBufDataset.OnFilterRecord	144
5.13.30	TBufDataset.OnNewRecord	144
5.13.31	TBufDataset.OnPostError	144
5.14	TBufIndex	145
5.14.1	Description	145
5.14.2	Method overview	146
5.14.3	Property overview	146



5.14.4	TBufIndex.Create	147
5.14.5	TBufIndex.ScrollBackward	147
5.14.6	TBufIndex.ScrollForward	147
5.14.7	TBufIndex.GetCurrent	147
5.14.8	TBufIndex.ScrollFirst	147
5.14.9	TBufIndex.ScrollLast	147
5.14.10	TBufIndex.GetRecord	148
5.14.11	TBufIndex.SetToFirstRecord	148
5.14.12	TBufIndex.SetToLastRecord	148
5.14.13	TBufIndex.StoreCurrentRecord	148
5.14.14	TBufIndex.RestoreCurrentRecord	148
5.14.15	TBufIndex.CanScrollForward	148
5.14.16	TBufIndex.DoScrollForward	149
5.14.17	TBufIndex.StoreCurrentRecIntoBookmark	149
5.14.18	TBufIndex.StoreSpareRecIntoBookmark	149
5.14.19	TBufIndex.GotoBookmark	149
5.14.20	TBufIndex.BookmarkValid	149
5.14.21	TBufIndex.CompareBookmarks	149
5.14.22	TBufIndex.SameBookmarks	150
5.14.23	TBufIndex.InitialiseIndex	150
5.14.24	TBufIndex.InitialiseSpareRecord	150
5.14.25	TBufIndex.ReleaseSpareRecord	150
5.14.26	TBufIndex.BeginUpdate	150
5.14.27	TBufIndex.AddRecord	151
5.14.28	TBufIndex.InsertRecordBeforeCurrentRecord	151
5.14.29	TBufIndex.RemoveRecordFromIndex	151
5.14.30	TBufIndex.OrderCurrentRecord	151
5.14.31	TBufIndex.EndUpdate	151
5.14.32	TBufIndex.SpareRecord	152
5.14.33	TBufIndex.SpareBuffer	152
5.14.34	TBufIndex.CurrentRecord	152
5.14.35	TBufIndex.CurrentBuffer	152
5.14.36	TBufIndex.IsInitialized	152
5.14.37	TBufIndex.BookmarkSize	153
5.14.38	TBufIndex.RecNo	153
5.15	TCustomBufDataset	153
5.15.1	Description	153
5.15.2	Method overview	154
5.15.3	Property overview	155
5.15.4	TCustomBufDataset.Create	155

5.15.5	TCustomBufDataset.GetFieldData	156
5.15.6	TCustomBufDataset.SetFieldData	156
5.15.7	TCustomBufDataset.ApplyUpdates	156
5.15.8	TCustomBufDataset.MergeChangeLog	157
5.15.9	TCustomBufDataset.RevertRecord	157
5.15.10	TCustomBufDataset.CancelUpdates	157
5.15.11	TCustomBufDataset.Destroy	157
5.15.12	TCustomBufDataset.Locate	158
5.15.13	TCustomBufDataset.Lookup	158
5.15.14	TCustomBufDataset.UpdateStatus	160
5.15.15	TCustomBufDataset.CreateBlobStream	160
5.15.16	TCustomBufDataset.AddIndex	160
5.15.17	TCustomBufDataset.ClearIndexes	161
5.15.18	TCustomBufDataset.SetDatasetPacket	161
5.15.19	TCustomBufDataset.GetDatasetPacket	161
5.15.20	TCustomBufDataset.LoadFromStream	162
5.15.21	TCustomBufDataset.SaveToStream	162
5.15.22	TCustomBufDataset.LoadFromFile	163
5.15.23	TCustomBufDataset.SaveToFile	163
5.15.24	TCustomBufDataset.CreateDataset	164
5.15.25	TCustomBufDataset.Clear	165
5.15.26	TCustomBufDataset.BookmarkValid	165
5.15.27	TCustomBufDataset.CompareBookmarks	165
5.15.28	TCustomBufDataset.CopyFromDataset	166
5.15.29	TCustomBufDataset.ChangeCount	166
5.15.30	TCustomBufDataset.MaxIndexesCount	166
5.15.31	TCustomBufDataset.ReadOnly	167
5.15.32	TCustomBufDataset.ManualMergeChangeLog	167
5.15.33	TCustomBufDataset.FileName	167
5.15.34	TCustomBufDataset.PacketRecords	168
5.15.35	TCustomBufDataset.OnUpdateError	168
5.15.36	TCustomBufDataset.IndexDefs	169
5.15.37	TCustomBufDataset.IndexName	169
5.15.38	TCustomBufDataset.IndexFieldNames	170
5.15.39	TCustomBufDataset.UniDirectional	170
5.16	TDataPacketReader	171
5.16.1	Description	171
5.16.2	Method overview	171
5.16.3	TDataPacketReader.Create	171
5.16.4	TDataPacketReader.LoadFieldDefs	171

5.16.5	<a href="#">TDataPacketReader.InitLoadRecords</a>	172
5.16.6	<a href="#">TDataPacketReader.GetCurrentRecord</a>	172
5.16.7	<a href="#">TDataPacketReader.GetRecordRowState</a>	172
5.16.8	<a href="#">TDataPacketReader.RestoreRecord</a>	172
5.16.9	<a href="#">TDataPacketReader.GotoNextRecord</a>	172
5.16.10	<a href="#">TDataPacketReader.StoreFieldDefs</a>	172
5.16.11	<a href="#">TDataPacketReader.StoreRecord</a>	172
5.16.12	<a href="#">TDataPacketReader.FinalizeStoreRecords</a>	173
5.16.13	<a href="#">TDataPacketReader.RecognizeStream</a>	173
5.17	<a href="#">TDoubleLinkedBufIndex</a>	173
5.17.1	<a href="#">Description</a>	173
5.17.2	<a href="#">Method overview</a>	174
5.17.3	<a href="#">TDoubleLinkedBufIndex.ScrollBackward</a>	174
5.17.4	<a href="#">TDoubleLinkedBufIndex.ScrollForward</a>	174
5.17.5	<a href="#">TDoubleLinkedBufIndex.GetCurrent</a>	174
5.17.6	<a href="#">TDoubleLinkedBufIndex.ScrollFirst</a>	175
5.17.7	<a href="#">TDoubleLinkedBufIndex.ScrollLast</a>	175
5.17.8	<a href="#">TDoubleLinkedBufIndex.GetRecord</a>	175
5.17.9	<a href="#">TDoubleLinkedBufIndex.SetToFirstRecord</a>	175
5.17.10	<a href="#">TDoubleLinkedBufIndex.SetToLastRecord</a>	175
5.17.11	<a href="#">TDoubleLinkedBufIndex.StoreCurrentRecord</a>	175
5.17.12	<a href="#">TDoubleLinkedBufIndex.RestoreCurrentRecord</a>	176
5.17.13	<a href="#">TDoubleLinkedBufIndex.CanScrollForward</a>	176
5.17.14	<a href="#">TDoubleLinkedBufIndex.DoScrollForward</a>	176
5.17.15	<a href="#">TDoubleLinkedBufIndex.StoreCurrentRecIntoBookmark</a>	176
5.17.16	<a href="#">TDoubleLinkedBufIndex.StoreSpareRecIntoBookmark</a>	176
5.17.17	<a href="#">TDoubleLinkedBufIndex.GotoBookmark</a>	176
5.17.18	<a href="#">TDoubleLinkedBufIndex.CompareBookmarks</a>	177
5.17.19	<a href="#">TDoubleLinkedBufIndex.SameBookmarks</a>	177
5.17.20	<a href="#">TDoubleLinkedBufIndex.InitialiseIndex</a>	177
5.17.21	<a href="#">TDoubleLinkedBufIndex.InitialiseSpareRecord</a>	177
5.17.22	<a href="#">TDoubleLinkedBufIndex.ReleaseSpareRecord</a>	177
5.17.23	<a href="#">TDoubleLinkedBufIndex.BeginUpdate</a>	177
5.17.24	<a href="#">TDoubleLinkedBufIndex.AddRecord</a>	178
5.17.25	<a href="#">TDoubleLinkedBufIndex.InsertRecordBeforeCurrentRecord</a>	178
5.17.26	<a href="#">TDoubleLinkedBufIndex.RemoveRecordFromIndex</a>	178
5.17.27	<a href="#">TDoubleLinkedBufIndex.OrderCurrentRecord</a>	178
5.17.28	<a href="#">TDoubleLinkedBufIndex.EndUpdate</a>	178
5.18	<a href="#">TFpcBinaryDatapacketReader</a>	178
5.18.1	<a href="#">Description</a>	178

5.18.2	Method overview	179
5.18.3	TFpcBinaryDatapacketReader.Create	179
5.18.4	TFpcBinaryDatapacketReader.LoadFieldDefs	179
5.18.5	TFpcBinaryDatapacketReader.StoreFieldDefs	180
5.18.6	TFpcBinaryDatapacketReader.InitLoadRecords	180
5.18.7	TFpcBinaryDatapacketReader.GetCurrentRecord	180
5.18.8	TFpcBinaryDatapacketReader.GetRecordRowState	180
5.18.9	TFpcBinaryDatapacketReader.RestoreRecord	180
5.18.10	TFpcBinaryDatapacketReader.GotoNextRecord	180
5.18.11	TFpcBinaryDatapacketReader.StoreRecord	181
5.18.12	TFpcBinaryDatapacketReader.FinalizeStoreRecords	181
5.18.13	TFpcBinaryDatapacketReader.RecognizeStream	181
5.19	TUniDirectionalBufIndex	181
5.19.1	Description	181
5.19.2	Method overview	182
5.19.3	TUniDirectionalBufIndex.ScrollBackward	182
5.19.4	TUniDirectionalBufIndex.ScrollForward	182
5.19.5	TUniDirectionalBufIndex.GetCurrent	182
5.19.6	TUniDirectionalBufIndex.ScrollFirst	182
5.19.7	TUniDirectionalBufIndex.ScrollLast	183
5.19.8	TUniDirectionalBufIndex.SetToFirstRecord	183
5.19.9	TUniDirectionalBufIndex.SetToLastRecord	183
5.19.10	TUniDirectionalBufIndex.StoreCurrentRecord	183
5.19.11	TUniDirectionalBufIndex.RestoreCurrentRecord	183
5.19.12	TUniDirectionalBufIndex.CanScrollForward	183
5.19.13	TUniDirectionalBufIndex.DoScrollForward	184
5.19.14	TUniDirectionalBufIndex.StoreCurrentRecIntoBookmark	184
5.19.15	TUniDirectionalBufIndex.StoreSpareRecIntoBookmark	184
5.19.16	TUniDirectionalBufIndex.GotoBookmark	184
5.19.17	TUniDirectionalBufIndex.InitialiseIndex	184
5.19.18	TUniDirectionalBufIndex.InitialiseSpareRecord	184
5.19.19	TUniDirectionalBufIndex.ReleaseSpareRecord	184
5.19.20	TUniDirectionalBufIndex.BeginUpdate	185
5.19.21	TUniDirectionalBufIndex.AddRecord	185
5.19.22	TUniDirectionalBufIndex.InsertRecordBeforeCurrentRecord	185
5.19.23	TUniDirectionalBufIndex.RemoveRecordFromIndex	185
5.19.24	TUniDirectionalBufIndex.OrderCurrentRecord	185
5.19.25	TUniDirectionalBufIndex.EndUpdate	185

## 6 Reference for unit 'bufstream'

186

6.1	Used units	186
6.2	Overview	186
6.3	Constants, types and variables	186
6.3.1	Constants	186
6.4	TBufferedFileStream	187
6.4.1	Method overview	187
6.4.2	TBufferedFileStream.Create	187
6.4.3	TBufferedFileStream.Destroy	187
6.4.4	TBufferedFileStream.Seek	187
6.4.5	TBufferedFileStream.Read	187
6.4.6	TBufferedFileStream.Write	187
6.4.7	TBufferedFileStream.Flush	187
6.4.8	TBufferedFileStream.InitializeCache	188
6.5	TBufStream	188
6.5.1	Description	188
6.5.2	Method overview	188
6.5.3	Property overview	188
6.5.4	TBufStream.Create	188
6.5.5	TBufStream.Destroy	189
6.5.6	TBufStream.Buffer	189
6.5.7	TBufStream.Capacity	189
6.5.8	TBufStream.BufferPos	189
6.5.9	TBufStream.BufferSize	190
6.6	TReadBufStream	190
6.6.1	Description	190
6.6.2	Method overview	190
6.6.3	TReadBufStream.Seek	190
6.6.4	TReadBufStream.Read	191
6.7	TWriteBufStream	191
6.7.1	Description	191
6.7.2	Method overview	191
6.7.3	TWriteBufStream.Destroy	191
6.7.4	TWriteBufStream.Seek	191
6.7.5	TWriteBufStream.Write	192
<b>7</b>	<b>Reference for unit 'CacheCls'</b>	<b>193</b>
7.1	Used units	193
7.2	Overview	193
7.3	Constants, types and variables	193
7.3.1	Resource strings	193

7.3.2	Types	193
7.4	TCacheSlot	194
7.5	ECacheError	194
7.5.1	Description	194
7.6	TCache	194
7.6.1	Description	194
7.6.2	Method overview	195
7.6.3	Property overview	195
7.6.4	TCache.Create	195
7.6.5	TCache.Destroy	195
7.6.6	TCache.Add	195
7.6.7	TCache.AddNew	196
7.6.8	TCache.FindSlot	196
7.6.9	TCache.IndexOf	196
7.6.10	TCache.Remove	197
7.6.11	TCache.Data	197
7.6.12	TCache.MRUSlot	197
7.6.13	TCache.LRUSlot	198
7.6.14	TCache.SlotCount	198
7.6.15	TCache.Slots	198
7.6.16	TCache.OnIsDataEqual	198
7.6.17	TCache.OnFreeSlot	199
<b>8</b>	<b>Reference for unit 'Contnrs'</b>	<b>200</b>
8.1	Used units	200
8.2	Overview	200
8.3	Constants, types and variables	200
8.3.1	Constants	200
8.3.2	Types	201
8.4	Procedures and functions	204
8.4.1	RSHash	204
8.5	TBucket	204
8.6	TBucketItem	204
8.7	THashItem	205
8.8	EDuplicate	205
8.8.1	Description	205
8.9	EKeyNotFound	205
8.9.1	Description	205
8.10	TBucketList	205
8.10.1	Description	205

8.10.2	Method overview	205
8.10.3	TBucketList.Create	205
8.11	TClassList	206
8.11.1	Description	206
8.11.2	Method overview	206
8.11.3	Property overview	206
8.11.4	TClassList.Add	206
8.11.5	TClassList.Extract	207
8.11.6	TClassList.Remove	207
8.11.7	TClassList.IndexOf	207
8.11.8	TClassList.First	207
8.11.9	TClassList.Last	208
8.11.10	TClassList.Insert	208
8.11.11	TClassList.Items	208
8.12	TComponentList	208
8.12.1	Description	208
8.12.2	Method overview	209
8.12.3	Property overview	209
8.12.4	TComponentList.Destroy	209
8.12.5	TComponentList.Add	209
8.12.6	TComponentList.Extract	209
8.12.7	TComponentList.Remove	210
8.12.8	TComponentList.IndexOf	210
8.12.9	TComponentList.First	210
8.12.10	TComponentList.Last	211
8.12.11	TComponentList.Insert	211
8.12.12	TComponentList.Items	211
8.13	TCustomBucketList	211
8.13.1	Description	211
8.13.2	Method overview	212
8.13.3	Property overview	212
8.13.4	TCustomBucketList.Destroy	212
8.13.5	TCustomBucketList.Clear	212
8.13.6	TCustomBucketList.Add	213
8.13.7	TCustomBucketList.Assign	213
8.13.8	TCustomBucketList.Exists	213
8.13.9	TCustomBucketList.Find	213
8.13.10	TCustomBucketList.ForEach	214
8.13.11	TCustomBucketList.Remove	214
8.13.12	TCustomBucketList.Data	214

8.14	TFPCustomHashTable	214
8.14.1	Description	214
8.14.2	Method overview	215
8.14.3	Property overview	215
8.14.4	TFPCustomHashTable.Create	215
8.14.5	TFPCustomHashTable.CreateWith	216
8.14.6	TFPCustomHashTable.Destroy	216
8.14.7	TFPCustomHashTable.ChangeTableSize	216
8.14.8	TFPCustomHashTable.Clear	216
8.14.9	TFPCustomHashTable.Delete	217
8.14.10	TFPCustomHashTable.Find	217
8.14.11	TFPCustomHashTable.IsEmpty	217
8.14.12	TFPCustomHashTable.HashFunction	217
8.14.13	TFPCustomHashTable.Count	218
8.14.14	TFPCustomHashTable.HashTableSize	218
8.14.15	TFPCustomHashTable.HashTable	218
8.14.16	TFPCustomHashTable.VoidSlots	218
8.14.17	TFPCustomHashTable.LoadFactor	219
8.14.18	TFPCustomHashTable.AVGChainLen	219
8.14.19	TFPCustomHashTable.MaxChainLength	219
8.14.20	TFPCustomHashTable.NumberOfCollisions	219
8.14.21	TFPCustomHashTable.Density	220
8.15	TFPDataHashTable	220
8.15.1	Description	220
8.15.2	Method overview	220
8.15.3	Property overview	220
8.15.4	TFPDataHashTable.Iterate	220
8.15.5	TFPDataHashTable.Add	221
8.15.6	TFPDataHashTable.Items	221
8.16	TFPHashList	221
8.16.1	Description	221
8.16.2	Method overview	222
8.16.3	Property overview	222
8.16.4	TFPHashList.Create	222
8.16.5	TFPHashList.Destroy	222
8.16.6	TFPHashList.Add	223
8.16.7	TFPHashList.Clear	223
8.16.8	TFPHashList.NameOfIndex	223
8.16.9	TFPHashList.HashOfIndex	223
8.16.10	TFPHashList.GetNextCollision	224



8.16.11	TFPHashList.Delete	224
8.16.12	TFPHashList.Error	224
8.16.13	TFPHashList.Expand	224
8.16.14	TFPHashList.Extract	224
8.16.15	TFPHashList.IndexOf	225
8.16.16	TFPHashList.Find	225
8.16.17	TFPHashList.FindIndexOf	225
8.16.18	TFPHashList.FindWithHash	225
8.16.19	TFPHashList.Rename	226
8.16.20	TFPHashList.Remove	226
8.16.21	TFPHashList.Pack	226
8.16.22	TFPHashList.ShowStatistics	226
8.16.23	TFPHashList.ForEachCall	227
8.16.24	TFPHashList.Capacity	227
8.16.25	TFPHashList.Count	227
8.16.26	TFPHashList.Items	227
8.16.27	TFPHashList.List	228
8.16.28	TFPHashList.Strs	228
8.17	TFPHashObject	228
8.17.1	Description	228
8.17.2	Method overview	228
8.17.3	Property overview	228
8.17.4	TFPHashObject.CreateNotOwned	229
8.17.5	TFPHashObject.Create	229
8.17.6	TFPHashObject.ChangeOwner	229
8.17.7	TFPHashObject.ChangeOwnerAndName	229
8.17.8	TFPHashObject.Rename	230
8.17.9	TFPHashObject.Name	230
8.17.10	TFPHashObject.Hash	230
8.18	TFPHashObjectList	231
8.18.1	Method overview	231
8.18.2	Property overview	231
8.18.3	TFPHashObjectList.Create	231
8.18.4	TFPHashObjectList.Destroy	231
8.18.5	TFPHashObjectList.Clear	232
8.18.6	TFPHashObjectList.Add	232
8.18.7	TFPHashObjectList.NameOfIndex	232
8.18.8	TFPHashObjectList.HashOfIndex	233
8.18.9	TFPHashObjectList.GetNextCollision	233
8.18.10	TFPHashObjectList.Delete	233

8.18.11	TFPHashObjectList.Expand	233
8.18.12	TFPHashObjectList.Extract	234
8.18.13	TFPHashObjectList.Remove	234
8.18.14	TFPHashObjectList.IndexOf	234
8.18.15	TFPHashObjectList.Find	234
8.18.16	TFPHashObjectList.FindIndexOf	235
8.18.17	TFPHashObjectList.FindWithHash	235
8.18.18	TFPHashObjectList.Rename	235
8.18.19	TFPHashObjectList.FindInstanceOf	235
8.18.20	TFPHashObjectList.Pack	236
8.18.21	TFPHashObjectList.ShowStatistics	236
8.18.22	TFPHashObjectList.ForEachCall	236
8.18.23	TFPHashObjectList.Capacity	236
8.18.24	TFPHashObjectList.Count	237
8.18.25	TFPHashObjectList.OwnsObjects	237
8.18.26	TFPHashObjectList.Items	237
8.18.27	TFPHashObjectList.List	237
8.19	TFPObjectHashTable	238
8.19.1	Description	238
8.19.2	Method overview	238
8.19.3	Property overview	238
8.19.4	TFPObjectHashTable.Create	238
8.19.5	TFPObjectHashTable.CreateWith	238
8.19.6	TFPObjectHashTable.Iterate	239
8.19.7	TFPObjectHashTable.Add	239
8.19.8	TFPObjectHashTable.Items	239
8.19.9	TFPObjectHashTable.OwnsObjects	240
8.20	TFPObjectList	240
8.20.1	Description	240
8.20.2	Method overview	240
8.20.3	Property overview	241
8.20.4	TFPObjectList.Create	241
8.20.5	TFPObjectList.Destroy	241
8.20.6	TFPObjectList.Clear	241
8.20.7	TFPObjectList.Add	241
8.20.8	TFPObjectList.Delete	242
8.20.9	TFPObjectList.Exchange	242
8.20.10	TFPObjectList.Expand	242
8.20.11	TFPObjectList.Extract	243
8.20.12	TFPObjectList.Remove	243

8.20.13	TFPObjectList.IndexOf	243
8.20.14	TFPObjectList.FindInstanceOf	243
8.20.15	TFPObjectList.Insert	244
8.20.16	TFPObjectList.First	244
8.20.17	TFPObjectList.Last	244
8.20.18	TFPObjectList.Move	245
8.20.19	TFPObjectList.Assign	245
8.20.20	TFPObjectList.Pack	245
8.20.21	TFPObjectList.Sort	245
8.20.22	TFPObjectList.ForEachCall	246
8.20.23	TFPObjectList.Capacity	246
8.20.24	TFPObjectList.Count	246
8.20.25	TFPObjectList.OwnsObjects	247
8.20.26	TFPObjectList.Items	247
8.20.27	TFPObjectList.List	247
8.21	TFPStringHashTable	247
8.21.1	Description	247
8.21.2	Method overview	248
8.21.3	Property overview	248
8.21.4	TFPStringHashTable.Iterate	248
8.21.5	TFPStringHashTable.Add	248
8.21.6	TFPStringHashTable.Items	248
8.22	THTCustomNode	249
8.22.1	Description	249
8.22.2	Method overview	249
8.22.3	Property overview	249
8.22.4	THTCustomNode.CreateWith	249
8.22.5	THTCustomNode.HasKey	249
8.22.6	THTCustomNode.Key	250
8.23	THTDataNode	250
8.23.1	Description	250
8.23.2	Property overview	250
8.23.3	THTDataNode.Data	250
8.24	THTObjectNode	250
8.24.1	Description	250
8.24.2	Property overview	251
8.24.3	THTObjectNode.Data	251
8.25	THTOwnedObjectNode	251
8.25.1	Description	251
8.25.2	Method overview	251

8.25.3	THTOwnedObjectNode.Destroy	251
8.26	THTStringNode	251
8.26.1	Description	251
8.26.2	Property overview	252
8.26.3	THTStringNode.Data	252
8.27	TObjectBucketList	252
8.27.1	Description	252
8.27.2	Method overview	252
8.27.3	Property overview	252
8.27.4	TObjectBucketList.Add	252
8.27.5	TObjectBucketList.Remove	253
8.27.6	TObjectBucketList.Data	253
8.28	TObjectList	253
8.28.1	Description	253
8.28.2	Method overview	253
8.28.3	Property overview	254
8.28.4	TObjectList.Create	254
8.28.5	TObjectList.Add	254
8.28.6	TObjectList.Extract	254
8.28.7	TObjectList.Remove	255
8.28.8	TObjectList.IndexOf	255
8.28.9	TObjectList.FindInstanceOf	255
8.28.10	TObjectList.Insert	256
8.28.11	TObjectList.First	256
8.28.12	TObjectList.Last	256
8.28.13	TObjectList.OwnsObjects	256
8.28.14	TObjectList.Items	257
8.29	TObjectQueue	257
8.29.1	Method overview	257
8.29.2	TObjectQueue.Push	257
8.29.3	TObjectQueue.Pop	257
8.29.4	TObjectQueue.Peek	258
8.30	TObjectStack	258
8.30.1	Description	258
8.30.2	Method overview	258
8.30.3	TObjectStack.Push	258
8.30.4	TObjectStack.Pop	258
8.30.5	TObjectStack.Peek	259
8.31	TOrderedList	259
8.31.1	Description	259

8.31.2	Method overview	259
8.31.3	TOrderedList.Create	259
8.31.4	TOrderedList.Destroy	260
8.31.5	TOrderedList.Count	260
8.31.6	TOrderedList.AtLeast	260
8.31.7	TOrderedList.Push	260
8.31.8	TOrderedList.Pop	261
8.31.9	TOrderedList.Peek	261
8.32	TQueue	261
8.32.1	Description	261
8.33	TStack	261
8.33.1	Description	261
<b>9</b>	<b>Reference for unit 'CustApp'</b>	<b>262</b>
9.1	Used units	262
9.2	Overview	262
9.3	Constants, types and variables	262
9.3.1	Types	262
9.3.2	Variables	263
9.4	TCustomApplication	263
9.4.1	Description	263
9.4.2	Method overview	264
9.4.3	Property overview	264
9.4.4	TCustomApplication.Create	264
9.4.5	TCustomApplication.Destroy	265
9.4.6	TCustomApplication.HandleException	265
9.4.7	TCustomApplication.Initialize	265
9.4.8	TCustomApplication.Run	266
9.4.9	TCustomApplication.ShowException	266
9.4.10	TCustomApplication.Terminate	266
9.4.11	TCustomApplication.FindOptionIndex	267
9.4.12	TCustomApplication.GetOptionValue	267
9.4.13	TCustomApplication.GetOptionValues	268
9.4.14	TCustomApplication.HasOption	268
9.4.15	TCustomApplication.CheckOptions	268
9.4.16	TCustomApplication.GetNonOptions	269
9.4.17	TCustomApplication.GetEnvironmentList	270
9.4.18	TCustomApplication.Log	270
9.4.19	TCustomApplication.ExeName	270
9.4.20	TCustomApplication.HelpFile	271

9.4.21	<a href="#">TCustomApplication.Terminated</a>	271
9.4.22	<a href="#">TCustomApplication.Title</a>	271
9.4.23	<a href="#">TCustomApplication.OnException</a>	271
9.4.24	<a href="#">TCustomApplication.ConsoleApplication</a>	272
9.4.25	<a href="#">TCustomApplication.Location</a>	272
9.4.26	<a href="#">TCustomApplication.Params</a>	272
9.4.27	<a href="#">TCustomApplication.ParamCount</a>	273
9.4.28	<a href="#">TCustomApplication.EnvironmentVariable</a>	273
9.4.29	<a href="#">TCustomApplication.OptionChar</a>	273
9.4.30	<a href="#">TCustomApplication.CaseSensitiveOptions</a>	273
9.4.31	<a href="#">TCustomApplication.StopOnException</a>	274
9.4.32	<a href="#">TCustomApplication.ExceptionExitCode</a>	274
9.4.33	<a href="#">TCustomApplication.EventLogFilter</a>	274
9.4.34	<a href="#">TCustomApplication.SingleInstance</a>	275
9.4.35	<a href="#">TCustomApplication.SingleInstanceClass</a>	275
9.4.36	<a href="#">TCustomApplication.SingleInstanceEnabled</a>	275
<b>10</b>	<b>Reference for unit 'daemonapp'</b>	<b>276</b>
10.1	<a href="#">Used units</a>	276
10.2	<a href="#">Overview</a>	276
10.3	<a href="#">Daemon application architecture</a>	277
10.4	<a href="#">Constants, types and variables</a>	277
10.4.1	<a href="#">Resource strings</a>	277
10.4.2	<a href="#">Types</a>	278
10.4.3	<a href="#">Variables</a>	281
10.5	<a href="#">Procedures and functions</a>	281
10.5.1	<a href="#">Application</a>	281
10.5.2	<a href="#">DaemonError</a>	282
10.5.3	<a href="#">RegisterDaemonApplicationClass</a>	282
10.5.4	<a href="#">RegisterDaemonClass</a>	282
10.5.5	<a href="#">RegisterDaemonMapper</a>	282
10.6	<a href="#">EDaemon</a>	283
10.6.1	<a href="#">Description</a>	283
10.7	<a href="#">TCustomDaemon</a>	283
10.7.1	<a href="#">Description</a>	283
10.7.2	<a href="#">Method overview</a>	283
10.7.3	<a href="#">Property overview</a>	283
10.7.4	<a href="#">TCustomDaemon.CheckControlMessages</a>	283
10.7.5	<a href="#">TCustomDaemon.LogMessage</a>	284
10.7.6	<a href="#">TCustomDaemon.ReportStatus</a>	284

10.7.7	<a href="#">TCustomDaemon.Definition</a>	284
10.7.8	<a href="#">TCustomDaemon.DaemonThread</a>	285
10.7.9	<a href="#">TCustomDaemon.Controller</a>	285
10.7.10	<a href="#">TCustomDaemon.Status</a>	285
10.7.11	<a href="#">TCustomDaemon.Logger</a>	285
10.8	<a href="#">TCustomDaemonApplication</a>	286
10.8.1	<a href="#">Description</a>	286
10.8.2	<a href="#">Method overview</a>	286
10.8.3	<a href="#">Property overview</a>	286
10.8.4	<a href="#">TCustomDaemonApplication.Create</a>	286
10.8.5	<a href="#">TCustomDaemonApplication.Destroy</a>	286
10.8.6	<a href="#">TCustomDaemonApplication.ShowException</a>	287
10.8.7	<a href="#">TCustomDaemonApplication.CreateDaemon</a>	287
10.8.8	<a href="#">TCustomDaemonApplication.StopDaemons</a>	287
10.8.9	<a href="#">TCustomDaemonApplication.InstallDaemons</a>	287
10.8.10	<a href="#">TCustomDaemonApplication.RunDaemons</a>	288
10.8.11	<a href="#">TCustomDaemonApplication.UnInstallDaemons</a>	288
10.8.12	<a href="#">TCustomDaemonApplication.ShowHelp</a>	288
10.8.13	<a href="#">TCustomDaemonApplication.CreateForm</a>	288
10.8.14	<a href="#">TCustomDaemonApplication.OnRun</a>	289
10.8.15	<a href="#">TCustomDaemonApplication.EventLog</a>	289
10.8.16	<a href="#">TCustomDaemonApplication.GUIMainLoop</a>	289
10.8.17	<a href="#">TCustomDaemonApplication.GuiHandle</a>	289
10.8.18	<a href="#">TCustomDaemonApplication.RunMode</a>	290
10.8.19	<a href="#">TCustomDaemonApplication.AutoRegisterMessageFile</a>	290
10.9	<a href="#">TCustomDaemonMapper</a>	290
10.9.1	<a href="#">Description</a>	290
10.9.2	<a href="#">Method overview</a>	290
10.9.3	<a href="#">Property overview</a>	290
10.9.4	<a href="#">TCustomDaemonMapper.Create</a>	291
10.9.5	<a href="#">TCustomDaemonMapper.Destroy</a>	291
10.9.6	<a href="#">TCustomDaemonMapper.DaemonDefs</a>	291
10.9.7	<a href="#">TCustomDaemonMapper.OnCreate</a>	291
10.9.8	<a href="#">TCustomDaemonMapper.OnDestroy</a>	292
10.9.9	<a href="#">TCustomDaemonMapper.OnRun</a>	292
10.9.10	<a href="#">TCustomDaemonMapper.OnInstall</a>	292
10.9.11	<a href="#">TCustomDaemonMapper.OnUnInstall</a>	292
10.10	<a href="#">TDaemon</a>	293
10.10.1	<a href="#">Description</a>	293
10.10.2	<a href="#">Property overview</a>	293

10.10.3	TDaemon.Definition	293
10.10.4	TDaemon.Status	293
10.10.5	TDaemon.OnStart	293
10.10.6	TDaemon.OnStop	294
10.10.7	TDaemon.OnPause	294
10.10.8	TDaemon.OnContinue	294
10.10.9	TDaemon.OnShutDown	295
10.10.10	TDaemon.OnExecute	295
10.10.11	TDaemon.BeforeInstall	295
10.10.12	TDaemon.AfterInstall	296
10.10.13	TDaemon.BeforeUnInstall	296
10.10.14	TDaemon.AfterUnInstall	296
10.10.15	TDaemon.OnControlCode	296
10.11	TDaemonApplication	297
10.11.1	Description	297
10.12	TDaemonController	297
10.12.1	Description	297
10.12.2	Method overview	297
10.12.3	Property overview	297
10.12.4	TDaemonController.Create	297
10.12.5	TDaemonController.Destroy	298
10.12.6	TDaemonController.StartService	298
10.12.7	TDaemonController.Main	298
10.12.8	TDaemonController.Controller	298
10.12.9	TDaemonController.ReportStatus	299
10.12.10	TDaemonController.Daemon	299
10.12.11	TDaemonController.Params	299
10.12.12	TDaemonController.LastStatus	299
10.12.13	TDaemonController.CheckPoint	300
10.13	TDaemonDef	300
10.13.1	Description	300
10.13.2	Method overview	300
10.13.3	Property overview	300
10.13.4	TDaemonDef.Create	300
10.13.5	TDaemonDef.Destroy	301
10.13.6	TDaemonDef.DaemonClass	301
10.13.7	TDaemonDef.Instance	301
10.13.8	TDaemonDef.DaemonClassName	301
10.13.9	TDaemonDef.Name	302
10.13.10	TDaemonDef.Description	302



10.13.1	<a href="#">TDaemonDef.DisplayName</a>	302
10.13.2	<a href="#">TDaemonDef.RunArguments</a>	302
10.13.3	<a href="#">TDaemonDef.Options</a>	303
10.13.4	<a href="#">TDaemonDef.Enabled</a>	303
10.13.5	<a href="#">TDaemonDef.WinBindings</a>	303
10.13.6	<a href="#">TDaemonDef.OnCreateInstance</a>	303
10.13.7	<a href="#">TDaemonDef.LogStatusReport</a>	304
10.14	<a href="#">TDaemonDefs</a>	304
10.14.1	<a href="#">Description</a>	304
10.14.2	<a href="#">Method overview</a>	304
10.14.3	<a href="#">Property overview</a>	304
10.14.4	<a href="#">TDaemonDefs.Create</a>	304
10.14.5	<a href="#">TDaemonDefs.IndexOfDaemonDef</a>	305
10.14.6	<a href="#">TDaemonDefs.FindDaemonDef</a>	305
10.14.7	<a href="#">TDaemonDefs.DaemonDefByName</a>	305
10.14.8	<a href="#">TDaemonDefs.Daemons</a>	305
10.15	<a href="#">TDaemonMapper</a>	306
10.15.1	<a href="#">Description</a>	306
10.15.2	<a href="#">Method overview</a>	306
10.15.3	<a href="#">TDaemonMapper.Create</a>	306
10.15.4	<a href="#">TDaemonMapper.CreateNew</a>	306
10.16	<a href="#">TDaemonThread</a>	307
10.16.1	<a href="#">Description</a>	307
10.16.2	<a href="#">Method overview</a>	307
10.16.3	<a href="#">Property overview</a>	307
10.16.4	<a href="#">TDaemonThread.Create</a>	307
10.16.5	<a href="#">TDaemonThread.Execute</a>	307
10.16.6	<a href="#">TDaemonThread.CheckControlMessage</a>	308
10.16.7	<a href="#">TDaemonThread.StopDaemon</a>	308
10.16.8	<a href="#">TDaemonThread.PauseDaemon</a>	308
10.16.9	<a href="#">TDaemonThread.ContinueDaemon</a>	308
10.16.10	<a href="#">TDaemonThread.ShutDownDaemon</a>	309
10.16.11	<a href="#">TDaemonThread.InterrogateDaemon</a>	309
10.16.12	<a href="#">TDaemonThread.Daemon</a>	309
10.17	<a href="#">TDependencies</a>	309
10.17.1	<a href="#">Description</a>	309
10.17.2	<a href="#">Method overview</a>	309
10.17.3	<a href="#">Property overview</a>	310
10.17.4	<a href="#">TDependencies.Create</a>	310
10.17.5	<a href="#">TDependencies.Items</a>	310

10.18	TDependency	310
10.18.1	Description	310
10.18.2	Method overview	310
10.18.3	Property overview	310
10.18.4	TDependency.Assign	311
10.18.5	TDependency.Name	311
10.18.6	TDependency.IsGroup	311
10.19	TWinBindings	311
10.19.1	Description	311
10.19.2	Method overview	311
10.19.3	Property overview	312
10.19.4	TWinBindings.Create	312
10.19.5	TWinBindings.Destroy	312
10.19.6	TWinBindings.Assign	312
10.19.7	TWinBindings.ErrCode	312
10.19.8	TWinBindings.Win32ErrCode	313
10.19.9	TWinBindings.Dependencies	313
10.19.10	TWinBindings.GroupName	313
10.19.11	TWinBindings.Password	314
10.19.12	TWinBindings.UserName	314
10.19.13	TWinBindings.StartType	314
10.19.14	TWinBindings.WaitHint	314
10.19.15	TWinBindings.IDTag	315
10.19.16	TWinBindings.ServiceType	315
10.19.17	TWinBindings.ErrorSeverity	315
<b>11</b>	<b>Reference for unit 'DB'</b>	<b>316</b>
11.1	Used units	316
11.2	Overview	316
11.3	Constants, types and variables	316
11.3.1	Constants	316
11.3.2	Types	318
11.3.3	Variables	332
11.4	Procedures and functions	332
11.4.1	BuffersEqual	332
11.4.2	DatabaseError	332
11.4.3	DatabaseErrorFmt	333
11.4.4	DateTimeRecToDateTime	333
11.4.5	DateTimeToDateTimeRec	333
11.4.6	DisposeMem	333

---

11.4.7	<a href="#">enumerator(TDataSet):TDataSetEnumerator</a>	334
11.4.8	<a href="#">ExtractFieldName</a>	334
11.4.9	<a href="#">SkipComments</a>	334
11.5	<a href="#">TLookupListRec</a>	335
11.6	<a href="#">EDatabaseError</a>	335
11.6.1	<a href="#">Description</a>	335
11.7	<a href="#">EUpdateError</a>	335
11.7.1	<a href="#">Description</a>	335
11.7.2	<a href="#">Method overview</a>	335
11.7.3	<a href="#">Property overview</a>	335
11.7.4	<a href="#">EUpdateError.Create</a>	335
11.7.5	<a href="#">EUpdateError.Destroy</a>	336
11.7.6	<a href="#">EUpdateError.Context</a>	336
11.7.7	<a href="#">EUpdateError.ErrorCode</a>	336
11.7.8	<a href="#">EUpdateError.OriginalException</a>	337
11.7.9	<a href="#">EUpdateError.PreviousError</a>	337
11.8	<a href="#">IProviderSupport</a>	337
11.8.1	<a href="#">Description</a>	337
11.8.2	<a href="#">Method overview</a>	338
11.8.3	<a href="#">IProviderSupport.PSEndTransaction</a>	338
11.8.4	<a href="#">IProviderSupport.PSExecute</a>	338
11.8.5	<a href="#">IProviderSupport.PSExecuteStatement</a>	338
11.8.6	<a href="#">IProviderSupport.PSGetAttributes</a>	339
11.8.7	<a href="#">IProviderSupport.PSGetCommandText</a>	339
11.8.8	<a href="#">IProviderSupport.PSGetCommandType</a>	339
11.8.9	<a href="#">IProviderSupport.PSGetDefaultOrder</a>	340
11.8.10	<a href="#">IProviderSupport.PSGetIndexDefs</a>	340
11.8.11	<a href="#">IProviderSupport.PSGetKeyFields</a>	340
11.8.12	<a href="#">IProviderSupport.PSGetParams</a>	340
11.8.13	<a href="#">IProviderSupport.PSGetQuoteChar</a>	341
11.8.14	<a href="#">IProviderSupport.PSGetTableName</a>	341
11.8.15	<a href="#">IProviderSupport.PSGetUpdateException</a>	341
11.8.16	<a href="#">IProviderSupport.PSInTransaction</a>	341
11.8.17	<a href="#">IProviderSupport.PSIsSQLBased</a>	342
11.8.18	<a href="#">IProviderSupport.PSIsSQLSupported</a>	342
11.8.19	<a href="#">IProviderSupport.PSReset</a>	342
11.8.20	<a href="#">IProviderSupport.PSSetCommandText</a>	342
11.8.21	<a href="#">IProviderSupport.PSSetParams</a>	343
11.8.22	<a href="#">IProviderSupport.PSStartTransaction</a>	343
11.8.23	<a href="#">IProviderSupport.PSUpdateRecord</a>	343

11.9 TAutoIncField . . . . .	343
11.9.1 Description . . . . .	343
11.9.2 Method overview . . . . .	344
11.9.3 TAutoIncField.Create . . . . .	344
11.10 TBCDField . . . . .	344
11.10.1 Description . . . . .	344
11.10.2 Method overview . . . . .	344
11.10.3 Property overview . . . . .	344
11.10.4 TBCDField.Create . . . . .	344
11.10.5 TBCDField.CheckRange . . . . .	345
11.10.6 TBCDField.Value . . . . .	345
11.10.7 TBCDField.Precision . . . . .	345
11.10.8 TBCDField.Currency . . . . .	346
11.10.9 TBCDField.MaxValue . . . . .	346
11.10.10 TBCDField.MinValue . . . . .	346
11.10.11 TBCDField.Size . . . . .	347
11.11 TBinaryField . . . . .	347
11.11.1 Description . . . . .	347
11.11.2 Method overview . . . . .	347
11.11.3 Property overview . . . . .	347
11.11.4 TBinaryField.Create . . . . .	347
11.11.5 TBinaryField.Size . . . . .	348
11.12 TBlobField . . . . .	348
11.12.1 Description . . . . .	348
11.12.2 Method overview . . . . .	348
11.12.3 Property overview . . . . .	348
11.12.4 TBlobField.Create . . . . .	348
11.12.5 TBlobField.Clear . . . . .	349
11.12.6 TBlobField.IsBlob . . . . .	349
11.12.7 TBlobField.LoadFromFile . . . . .	349
11.12.8 TBlobField.LoadFromStream . . . . .	349
11.12.9 TBlobField.SaveToFile . . . . .	350
11.12.10 TBlobField.SaveToStream . . . . .	350
11.12.11 TBlobField.SetFieldType . . . . .	350
11.12.12 TBlobField.BlobSize . . . . .	350
11.12.13 TBlobField.Modified . . . . .	351
11.12.14 TBlobField.Value . . . . .	351
11.12.15 TBlobField.Transliterate . . . . .	351
11.12.16 TBlobField.BlobType . . . . .	351
11.12.17 TBlobField.Size . . . . .	352

---

11.13TBooleanField	352
11.13.1 Description	352
11.13.2 Method overview	352
11.13.3 Property overview	352
11.13.4 TBooleanField.Create	352
11.13.5 TBooleanField.Value	353
11.13.6 TBooleanField.DisplayValues	353
11.14TBytesField	353
11.14.1 Description	353
11.14.2 Method overview	353
11.14.3 TBytesField.Create	354
11.15TCheckConstraint	354
11.15.1 Description	354
11.15.2 Method overview	354
11.15.3 Property overview	354
11.15.4 TCheckConstraint.Assign	354
11.15.5 TCheckConstraint.CustomConstraint	355
11.15.6 TCheckConstraint.ErrorMessage	355
11.15.7 TCheckConstraint.FromDictionary	355
11.15.8 TCheckConstraint.ImportedConstraint	355
11.16TCheckConstraints	356
11.16.1 Description	356
11.16.2 Method overview	356
11.16.3 Property overview	356
11.16.4 TCheckConstraints.Create	356
11.16.5 TCheckConstraints.Add	356
11.16.6 TCheckConstraints.Items	356
11.17TCurrencyField	357
11.17.1 Description	357
11.17.2 Method overview	357
11.17.3 Property overview	357
11.17.4 TCurrencyField.Create	357
11.17.5 TCurrencyField.Currency	357
11.18TCustomConnection	358
11.18.1 Description	358
11.18.2 Method overview	358
11.18.3 Property overview	358
11.18.4 TCustomConnection.Close	358
11.18.5 TCustomConnection.Destroy	359
11.18.6 TCustomConnection.Open	359

11.18.7 TCustomConnection.DataSetCount . . . . .	359
11.18.8 TCustomConnection.DataSets . . . . .	360
11.18.9 TCustomConnection.Connected . . . . .	360
11.18.10 TCustomConnection.LoginPrompt . . . . .	360
11.18.11 TCustomConnection.AfterConnect . . . . .	360
11.18.12 TCustomConnection.AfterDisconnect . . . . .	361
11.18.13 TCustomConnection.BeforeConnect . . . . .	361
11.18.14 TCustomConnection.BeforeDisconnect . . . . .	361
11.18.15 TCustomConnection.OnLogin . . . . .	361
11.19 TDatabase . . . . .	362
11.19.1 Description . . . . .	362
11.19.2 Method overview . . . . .	362
11.19.3 Property overview . . . . .	362
11.19.4 TDatabase.Create . . . . .	362
11.19.5 TDatabase.Destroy . . . . .	363
11.19.6 TDatabase.CloseDataSets . . . . .	363
11.19.7 TDatabase.CloseTransactions . . . . .	363
11.19.8 TDatabase.StartTransaction . . . . .	363
11.19.9 TDatabase.EndTransaction . . . . .	364
11.19.10 TDatabase.TransactionCount . . . . .	364
11.19.11 TDatabase.Transactions . . . . .	364
11.19.12 TDatabase.Directory . . . . .	364
11.19.13 TDatabase.IsSQLBased . . . . .	365
11.19.14 TDatabase.Connected . . . . .	365
11.19.15 TDatabase.DatabaseName . . . . .	365
11.19.16 TDatabase.KeepConnection . . . . .	366
11.19.17 TDatabase.Params . . . . .	366
11.20 TDataLink . . . . .	366
11.20.1 Description . . . . .	366
11.20.2 Method overview . . . . .	367
11.20.3 Property overview . . . . .	367
11.20.4 TDataLink.Create . . . . .	367
11.20.5 TDataLink.Destroy . . . . .	367
11.20.6 TDataLink.Edit . . . . .	367
11.20.7 TDataLink.UpdateRecord . . . . .	368
11.20.8 TDataLink.ExecuteAction . . . . .	368
11.20.9 TDataLink.UpdateAction . . . . .	368
11.20.10 TDataLink.Active . . . . .	369
11.20.11 TDataLink.ActiveRecord . . . . .	369
11.20.12 TDataLink.BOF . . . . .	369

11.20.13	TDDataLink.BufferCount	369
11.20.14	TDDataLink.DataSet	370
11.20.15	TDDataLink.DataSource	370
11.20.16	TDDataLink.DataSourceFixed	370
11.20.17	TDDataLink.Editing	370
11.20.18	TDDataLink.Eof	371
11.20.19	TDDataLink.ReadOnly	371
11.20.20	TDDataLink.RecordCount	371
11.21	TDDataSet	371
11.21.1	Description	371
11.21.2	Method overview	374
11.21.3	Property overview	375
11.21.4	TDDataSet.Create	376
11.21.5	TDDataSet.Destroy	376
11.21.6	TDDataSet.ActiveBuffer	376
11.21.7	TDDataSet.GetFieldData	376
11.21.8	TDDataSet.SetFieldData	377
11.21.9	TDDataSet.Append	377
11.21.10	TDDataSet.AppendRecord	377
11.21.11	TDDataSet.BookmarkValid	378
11.21.12	TDDataSet.Cancel	378
11.21.13	TDDataSet.CheckBrowseMode	378
11.21.14	TDDataSet.ClearFields	378
11.21.15	TDDataSet.Close	379
11.21.16	TDDataSet.ControlsDisabled	379
11.21.17	TDDataSet.CompareBookmarks	379
11.21.18	TDDataSet.CreateBlobStream	380
11.21.19	TDDataSet.CursorPosChanged	380
11.21.20	TDDataSet.DataConvert	380
11.21.21	TDDataSet.Delete	380
11.21.22	TDDataSet.DisableControls	381
11.21.23	TDDataSet.Edit	381
11.21.24	TDDataSet.EnableControls	382
11.21.25	TDDataSet.FieldByName	382
11.21.26	TDDataSet.FindField	382
11.21.27	TDDataSet.FindFirst	383
11.21.28	TDDataSet.FindLast	383
11.21.29	TDDataSet.FindNext	383
11.21.30	TDDataSet.FindPrior	383
11.21.31	TDDataSet.First	384

11.21.32TDataSet.FreeBookmark . . . . .	384
11.21.33TDataSet.GetBookmark . . . . .	384
11.21.34TDataSet.GetCurrentRecord . . . . .	385
11.21.35TDataSet.GetFieldList . . . . .	385
11.21.36TDataSet.GetFieldNames . . . . .	385
11.21.37TDataSet.GotoBookmark . . . . .	385
11.21.38TDataSet.Insert . . . . .	386
11.21.39TDataSet.InsertRecord . . . . .	386
11.21.40TDataSet.IsEmpty . . . . .	386
11.21.41TDataSet.IsLinkedTo . . . . .	386
11.21.42TDataSet.IsSequenced . . . . .	387
11.21.43TDataSet.Last . . . . .	387
11.21.44TDataSet.Locate . . . . .	387
11.21.45TDataSet.Lookup . . . . .	388
11.21.46TDataSet.MoveBy . . . . .	388
11.21.47TDataSet.Next . . . . .	388
11.21.48TDataSet.Open . . . . .	389
11.21.49TDataSet.Post . . . . .	389
11.21.50TDataSet.Prior . . . . .	390
11.21.51TDataSet.Refresh . . . . .	390
11.21.52TDataSet.Resync . . . . .	390
11.21.53TDataSet.SetFields . . . . .	390
11.21.54TDataSet.Translate . . . . .	391
11.21.55TDataSet.UpdateCursorPos . . . . .	391
11.21.56TDataSet.UpdateRecord . . . . .	391
11.21.57TDataSet.UpdateStatus . . . . .	392
11.21.58TDataSet.BlockReadSize . . . . .	392
11.21.59TDataSet.BOF . . . . .	392
11.21.60TDataSet.Bookmark . . . . .	392
11.21.61TDataSet.CanModify . . . . .	393
11.21.62TDataSet.DataSource . . . . .	394
11.21.63TDataSet.DefaultFields . . . . .	394
11.21.64TDataSet.EOF . . . . .	394
11.21.65TDataSet.FieldCount . . . . .	395
11.21.66TDataSet.FieldDefs . . . . .	395
11.21.67TDataSet.Found . . . . .	396
11.21.68TDataSet.Modified . . . . .	396
11.21.69TDataSet.IsUniDirectional . . . . .	396
11.21.70TDataSet.RecordCount . . . . .	397
11.21.71TDataSet.RecNo . . . . .	397



11.21.72	TDataSet.RecordSize	397
11.21.73	TDataSet.State	398
11.21.74	TDataSet.Fields	398
11.21.75	TDataSet.FieldValues	398
11.21.76	TDataSet.Filter	399
11.21.77	TDataSet.Filtered	399
11.21.78	TDataSet.FilterOptions	399
11.21.79	TDataSet.Active	400
11.21.80	TDataSet.AutoCalcFields	400
11.21.81	TDataSet.BeforeOpen	400
11.21.82	TDataSet.AfterOpen	401
11.21.83	TDataSet.BeforeClose	401
11.21.84	TDataSet.AfterClose	401
11.21.85	TDataSet.BeforeInsert	401
11.21.86	TDataSet.AfterInsert	402
11.21.87	TDataSet.BeforeEdit	402
11.21.88	TDataSet.AfterEdit	402
11.21.89	TDataSet.BeforePost	403
11.21.90	TDataSet.AfterPost	403
11.21.91	TDataSet.BeforeCancel	403
11.21.92	TDataSet.AfterCancel	404
11.21.93	TDataSet.BeforeDelete	404
11.21.94	TDataSet.AfterDelete	404
11.21.95	TDataSet.BeforeScroll	404
11.21.96	TDataSet.AfterScroll	405
11.21.97	TDataSet.BeforeRefresh	405
11.21.98	TDataSet.AfterRefresh	405
11.21.99	TDataSet.OnCalcFields	406
11.21.100	TDataSet.OnDeleteError	406
11.21.101	TDataSet.OnEditError	406
11.21.102	TDataSet.OnFilterRecord	407
11.21.103	TDataSet.OnNewRecord	407
11.21.104	TDataSet.OnPostError	408
11.22	TDataSetEnumerator	408
11.22.1	Description	408
11.22.2	Method overview	408
11.22.3	Property overview	409
11.22.4	TDataSetEnumerator.Create	409
11.22.5	TDataSetEnumerator.MoveNext	409
11.22.6	TDataSetEnumerator.Current	409

11.23	TDataSource	409
11.23.1	Description	409
11.23.2	Method overview	410
11.23.3	Property overview	410
11.23.4	TDataSource.Create	410
11.23.5	TDataSource.Destroy	410
11.23.6	TDataSource.Edit	410
11.23.7	TDataSource.IsLinkedTo	411
11.23.8	TDataSource.State	411
11.23.9	TDataSource.AutoEdit	411
11.23.10	TDataSource.DataSet	412
11.23.11	TDataSource.Enabled	412
11.23.12	TDataSource.OnStateChange	412
11.23.13	TDataSource.OnDataChange	413
11.23.14	TDataSource.OnUpdateData	413
11.24	TDateField	413
11.24.1	Description	413
11.24.2	Method overview	413
11.24.3	TDateField.Create	414
11.25	TDateTimeField	414
11.25.1	Description	414
11.25.2	Method overview	414
11.25.3	Property overview	414
11.25.4	TDateTimeField.Create	414
11.25.5	TDateTimeField.Value	415
11.25.6	TDateTimeField.DisplayFormat	415
11.25.7	TDateTimeField.EditMask	415
11.26	TDBDataset	415
11.26.1	Description	415
11.26.2	Method overview	416
11.26.3	Property overview	416
11.26.4	TDBDataset.destroy	416
11.26.5	TDBDataset.DataBase	416
11.26.6	TDBDataset.Transaction	417
11.27	TDBTransaction	417
11.27.1	Description	417
11.27.2	Method overview	417
11.27.3	Property overview	417
11.27.4	TDBTransaction.Create	417
11.27.5	TDBTransaction.Destroy	418

11.27.6 TDBTransaction.CloseDataSets	418
11.27.7 TDBTransaction.DataBase	418
11.27.8 TDBTransaction.Active	418
11.28 TDefCollection	419
11.28.1 Description	419
11.28.2 Method overview	419
11.28.3 Property overview	419
11.28.4 TDefCollection.create	419
11.28.5 TDefCollection.Find	419
11.28.6 TDefCollection.GetItemNames	420
11.28.7 TDefCollection.IndexOf	420
11.28.8 TDefCollection.Dataset	420
11.28.9 TDefCollection.Updated	420
11.29 TDetailDataLink	421
11.29.1 Description	421
11.29.2 Property overview	421
11.29.3 TDetailDataLink.DetailDataSet	421
11.30 TField	421
11.30.1 Description	421
11.30.2 Method overview	422
11.30.3 Property overview	424
11.30.4 TField.Create	425
11.30.5 TField.Destroy	425
11.30.6 TField.Assign	425
11.30.7 TField.AssignValue	425
11.30.8 TField.Clear	426
11.30.9 TField.FocusControl	426
11.30.10 TField.GetData	426
11.30.11 TField.IsBlob	427
11.30.12 TField.IsValidChar	427
11.30.13 TField.RefreshLookupList	427
11.30.14 TField.SetData	427
11.30.15 TField.SetFieldType	428
11.30.16 TField.Validate	428
11.30.17 TField.AsBCD	428
11.30.18 TField.AsBoolean	429
11.30.19 TField.AsBytes	429
11.30.20 TField.AsCurrency	429
11.30.21 TField.AsDateTime	430
11.30.22 TField.AsFloat	430

11.30.23	Field.AsLongint	430
11.30.24	Field.AsLargeInt	431
11.30.25	Field.AsInteger	431
11.30.26	Field.AsString	431
11.30.27	Field.AsAnsiString	432
11.30.28	Field.AsUnicodeString	432
11.30.29	Field.AsUTF8String	432
11.30.30	Field.AsWideString	433
11.30.31	Field.AsVariant	433
11.30.32	Field.AttributeSet	433
11.30.33	Field.Calculated	434
11.30.34	Field.CanModify	434
11.30.35	Field.CurValue	434
11.30.36	Field.DataSet	434
11.30.37	Field.DataSize	435
11.30.38	Field.DataType	435
11.30.39	Field.DisplayName	435
11.30.40	Field.DisplayText	435
11.30.41	Field.EditMask	436
11.30.42	Field.EditMaskPtr	436
11.30.43	Field.FieldNo	436
11.30.44	Field.IsIndexField	437
11.30.45	Field.IsNull	437
11.30.46	Field.NewValue	437
11.30.47	Field.Offset	437
11.30.48	Field.Size	438
11.30.49	Field.Text	438
11.30.50	Field.ValidChars	438
11.30.51	Field.Value	439
11.30.52	Field.OldValue	439
11.30.53	Field.LookupList	439
11.30.54	Field.FieldDef	439
11.30.55	Field.Alignment	440
11.30.56	Field.CustomConstraint	440
11.30.57	Field.ConstraintErrorMessage	440
11.30.58	Field.DefaultExpression	441
11.30.59	Field.DisplayLabel	441
11.30.60	Field.DisplayWidth	441
11.30.61	Field.FieldKind	442
11.30.62	Field.FieldName	442

11.30.63	Field.HasConstraints	442
11.30.64	Field.Index	442
11.30.65	Field.ImportedConstraint	443
11.30.66	Field.KeyFields	443
11.30.67	Field.LookupCache	443
11.30.68	Field.LookupDataSet	443
11.30.69	Field.LookupKeyFields	444
11.30.70	Field.LookupResultField	444
11.30.71	Field.Lookup	444
11.30.72	Field.Origin	444
11.30.73	Field.ProviderFlags	445
11.30.74	Field.ReadOnly	445
11.30.75	Field.Required	445
11.30.76	Field.Visible	446
11.30.77	Field.OnChange	446
11.30.78	Field.OnGetText	446
11.30.79	Field.OnSetText	446
11.30.80	Field.OnValidate	447
11.31	TFieldDef	447
11.31.1	Description	447
11.31.2	Method overview	447
11.31.3	Property overview	447
11.31.4	TFieldDef.Create	448
11.31.5	TFieldDef.Destroy	448
11.31.6	TFieldDef.Assign	448
11.31.7	TFieldDef.CreateField	448
11.31.8	TFieldDef.FieldClass	449
11.31.9	TFieldDef.FieldNo	449
11.31.10	TFieldDef.CharSize	449
11.31.11	TFieldDef.InternalCalcField	450
11.31.12	TFieldDef.Required	450
11.31.13	TFieldDef.Codepage	450
11.31.14	TFieldDef.Attributes	450
11.31.15	TFieldDef.DataType	451
11.31.16	TFieldDef.Precision	451
11.31.17	TFieldDef.Size	451
11.32	TFieldDefs	451
11.32.1	Description	451
11.32.2	Method overview	452
11.32.3	Property overview	452

11.32.4 TFieldDefs.Create	452
11.32.5 TFieldDefs.Add	452
11.32.6 TFieldDefs.AddFieldDef	453
11.32.7 TFieldDefs.Assign	453
11.32.8 TFieldDefs.Find	453
11.32.9 TFieldDefs.Update	453
11.32.10 TFieldDefs.MakeNameUnique	454
11.32.11 TFieldDefs.HiddenFields	454
11.32.12 TFieldDefs.Items	454
11.33 TFields	454
11.33.1 Description	454
11.33.2 Method overview	455
11.33.3 Property overview	455
11.33.4 TFields.Create	455
11.33.5 TFields.Destroy	455
11.33.6 TFields.Add	455
11.33.7 TFields.CheckFieldName	456
11.33.8 TFields.CheckFieldNames	456
11.33.9 TFields.Clear	456
11.33.10 TFields.FindField	456
11.33.11 TFields.FieldByName	457
11.33.12 TFields.FieldByNumber	457
11.33.13 TFields.GetEnumerator	457
11.33.14 TFields.GetFieldNames	457
11.33.15 TFields.IndexOf	458
11.33.16 TFields.Remove	458
11.33.17 TFields.Count	458
11.33.18 TFields.Dataset	458
11.33.19 TFields.Fields	459
11.34 TFieldsEnumerator	459
11.34.1 Description	459
11.34.2 Method overview	459
11.34.3 Property overview	459
11.34.4 TFieldsEnumerator.Create	460
11.34.5 TFieldsEnumerator.MoveNext	460
11.34.6 TFieldsEnumerator.Current	460
11.35 TFloatField	460
11.35.1 Description	460
11.35.2 Method overview	460
11.35.3 Property overview	461

11.35.4 TFloatField.Create	461
11.35.5 TFloatField.CheckRange	461
11.35.6 TFloatField.Value	461
11.35.7 TFloatField.Currency	461
11.35.8 TFloatField.MaxValue	462
11.35.9 TFloatField.MinValue	462
11.35.10 TFloatField.Precision	462
11.36 TFMTCBDField	463
11.36.1 Description	463
11.36.2 Method overview	463
11.36.3 Property overview	463
11.36.4 TFMTCBDField.Create	463
11.36.5 TFMTCBDField.CheckRange	463
11.36.6 TFMTCBDField.Value	464
11.36.7 TFMTCBDField.Precision	464
11.36.8 TFMTCBDField.Currency	464
11.36.9 TFMTCBDField.MaxValue	464
11.36.10 TFMTCBDField.MinValue	465
11.36.11 TFMTCBDField.Size	465
11.37 TGraphicField	465
11.37.1 Description	465
11.37.2 Method overview	465
11.37.3 TGraphicField.Create	465
11.38 TGUIDField	466
11.38.1 Description	466
11.38.2 Method overview	466
11.38.3 Property overview	466
11.38.4 TGUIDField.Create	466
11.38.5 TGUIDField.AsGUID	466
11.39 TIndexDef	467
11.39.1 Description	467
11.39.2 Method overview	467
11.39.3 Property overview	467
11.39.4 TIndexDef.Create	467
11.39.5 TIndexDef.Expression	467
11.39.6 TIndexDef.Fields	468
11.39.7 TIndexDef.CaseInsFields	468
11.39.8 TIndexDef.DescFields	468
11.39.9 TIndexDef.Options	469
11.39.10 TIndexDef.Source	469

11.40	TIndexDefs	469
11.40.1	Description	469
11.40.2	Method overview	469
11.40.3	Property overview	469
11.40.4	TIndexDefs.Create	470
11.40.5	TIndexDefs.Add	470
11.40.6	TIndexDefs.AddIndexDef	470
11.40.7	TIndexDefs.Find	470
11.40.8	TIndexDefs.FindIndexForFields	471
11.40.9	TIndexDefs.GetIndexForFields	471
11.40.10	TIndexDefs.Update	471
11.40.11	TIndexDefs.Items	471
11.41	TIntegerField	472
11.41.1	Description	472
11.42	TLargeintField	472
11.42.1	Description	472
11.42.2	Method overview	472
11.42.3	Property overview	472
11.42.4	TLargeintField.Create	472
11.42.5	TLargeintField.CheckRange	472
11.42.6	TLargeintField.Value	473
11.42.7	TLargeintField.MaxValue	473
11.42.8	TLargeintField.MinValue	473
11.43	TLongintField	474
11.43.1	Description	474
11.43.2	Method overview	474
11.43.3	Property overview	474
11.43.4	TLongintField.Create	474
11.43.5	TLongintField.CheckRange	474
11.43.6	TLongintField.Value	475
11.43.7	TLongintField.MaxValue	475
11.43.8	TLongintField.MinValue	475
11.44	TLookupList	475
11.44.1	Description	475
11.44.2	Method overview	476
11.44.3	TLookupList.Create	476
11.44.4	TLookupList.Destroy	476
11.44.5	TLookupList.Add	476
11.44.6	TLookupList.Clear	476
11.44.7	TLookupList.FirstKeyByValue	477



---

11.44.8 TLookupList.ValueOfKey	477
11.44.9 TLookupList.ValuesToStrings	477
11.45 TMasterDataLink	477
11.45.1 Description	477
11.45.2 Method overview	478
11.45.3 Property overview	478
11.45.4 TMasterDataLink.Create	478
11.45.5 TMasterDataLink.Destroy	478
11.45.6 TMasterDataLink.FieldNames	478
11.45.7 TMasterDataLink.Fields	479
11.45.8 TMasterDataLink.OnMasterChange	479
11.45.9 TMasterDataLink.OnMasterDisable	479
11.46 TMasterParamsDataLink	479
11.46.1 Description	479
11.46.2 Method overview	480
11.46.3 Property overview	480
11.46.4 TMasterParamsDataLink.Create	480
11.46.5 TMasterParamsDataLink.RefreshParamNames	480
11.46.6 TMasterParamsDataLink.CopyParamsFromMaster	480
11.46.7 TMasterParamsDataLink.Params	481
11.47 TMemoField	481
11.47.1 Description	481
11.47.2 Method overview	481
11.47.3 Property overview	481
11.47.4 TMemoField.Create	481
11.47.5 TMemoField.CodePage	482
11.47.6 TMemoField.Transliterate	482
11.48 TNamedItem	482
11.48.1 Description	482
11.48.2 Property overview	482
11.48.3 TNamedItem.DisplayName	482
11.48.4 TNamedItem.Name	483
11.49 TNumericField	483
11.49.1 Description	483
11.49.2 Method overview	483
11.49.3 Property overview	483
11.49.4 TNumericField.Create	483
11.49.5 TNumericField.Alignment	484
11.49.6 TNumericField.DisplayFormat	484
11.49.7 TNumericField.EditFormat	484

11.50	TParam	484
11.50.1	Description	484
11.50.2	Method overview	485
11.50.3	Property overview	485
11.50.4	TParam.Create	486
11.50.5	TParam.Assign	486
11.50.6	TParam.AssignField	486
11.50.7	TParam.AssignToField	486
11.50.8	TParam.AssignFieldValue	487
11.50.9	TParam.AssignFromField	487
11.50.10	TParam.Clear	487
11.50.11	TParam.GetData	487
11.50.12	TParam.GetDataSize	488
11.50.13	TParam.LoadFromFile	488
11.50.14	TParam.LoadFromStream	488
11.50.15	TParam.SetBlobData	489
11.50.16	TParam.SetData	489
11.50.17	TParam.AsBCD	489
11.50.18	TParam.AsBlob	489
11.50.19	TParam.AsBoolean	490
11.50.20	TParam.AsBytes	490
11.50.21	TParam.AsCurrency	490
11.50.22	TParam.AsDate	490
11.50.23	TParam.AsDateTime	491
11.50.24	TParam.AsFloat	491
11.50.25	TParam.AsInteger	491
11.50.26	TParam.AsLargeInt	491
11.50.27	TParam.AsMemo	492
11.50.28	TParam.AsSmallInt	492
11.50.29	TParam.AsString	492
11.50.30	TParam.AsAnsiString	492
11.50.31	TParam.AsUTF8String	493
11.50.32	TParam.AsUnicodeString	493
11.50.33	TParam.AsTime	493
11.50.34	TParam.AsWord	493
11.50.35	TParam.AsFmtBCD	494
11.50.36	TParam.Bound	494
11.50.37	TParam.Dataset	494
11.50.38	TParam.IsNull	494
11.50.39	TParam.NativeStr	495

11.50.40	TParam.Text	495
11.50.41	TParam.AsWideString	495
11.50.42	TParam.DataType	495
11.50.43	TParam.Name	496
11.50.44	TParam.NumericScale	496
11.50.45	TParam.ParamType	496
11.50.46	TParam.Precision	497
11.50.47	TParam.Size	497
11.50.48	TParam.Value	497
11.51	TParams	497
11.51.1	Description	497
11.51.2	Method overview	498
11.51.3	Property overview	498
11.51.4	TParams.Create	498
11.51.5	TParams.AddParam	498
11.51.6	TParams.AssignValues	499
11.51.7	TParams.CreateParam	499
11.51.8	TParams.FindParam	499
11.51.9	TParams.GetParamList	499
11.51.10	TParams.IsEqual	500
11.51.11	TParams.GetEnumerator	500
11.51.12	TParams.ParamByName	500
11.51.13	TParams.ParseSQL	500
11.51.14	TParams.RemoveParam	501
11.51.15	TParams.CopyParamValuesFromDataset	502
11.51.16	TParams.Dataset	502
11.51.17	TParams.Items	502
11.51.18	TParams.ParamValues	502
11.52	TParamsEnumerator	503
11.52.1	Description	503
11.52.2	Method overview	503
11.52.3	Property overview	503
11.52.4	TParamsEnumerator.Create	503
11.52.5	TParamsEnumerator.MoveNext	503
11.52.6	TParamsEnumerator.Current	504
11.53	TSmallintField	504
11.53.1	Description	504
11.53.2	Method overview	504
11.53.3	TSmallintField.Create	504
11.54	TStringField	504

---

11.54.1 Description . . . . .	504
11.54.2 Method overview . . . . .	505
11.54.3 Property overview . . . . .	505
11.54.4 TStringField.Create . . . . .	505
11.54.5 TStringField.SetFieldType . . . . .	505
11.54.6 TStringField.CodePage . . . . .	505
11.54.7 TStringField.FixedChar . . . . .	506
11.54.8 TStringField.Transliterate . . . . .	506
11.54.9 TStringField.Value . . . . .	506
11.54.10 TStringField.EditMask . . . . .	506
11.54.11 TStringField.Size . . . . .	507
11.55 TTimeField . . . . .	507
11.55.1 Description . . . . .	507
11.55.2 Method overview . . . . .	507
11.55.3 TTimeField.Create . . . . .	507
11.56 TVarBytesField . . . . .	508
11.56.1 Description . . . . .	508
11.56.2 Method overview . . . . .	508
11.56.3 TVarBytesField.Create . . . . .	508
11.57 TVariantField . . . . .	508
11.57.1 Description . . . . .	508
11.57.2 Method overview . . . . .	508
11.57.3 TVariantField.Create . . . . .	509
11.58 TWideMemoField . . . . .	509
11.58.1 Description . . . . .	509
11.58.2 Method overview . . . . .	509
11.58.3 Property overview . . . . .	509
11.58.4 TWideMemoField.Create . . . . .	509
11.58.5 TWideMemoField.Value . . . . .	510
11.59 TWideStringField . . . . .	510
11.59.1 Description . . . . .	510
11.59.2 Method overview . . . . .	510
11.59.3 Property overview . . . . .	510
11.59.4 TWideStringField.Create . . . . .	510
11.59.5 TWideStringField.SetFieldType . . . . .	510
11.59.6 TWideStringField.Value . . . . .	511
11.60 TWordField . . . . .	511
11.60.1 Description . . . . .	511
11.60.2 Method overview . . . . .	511
11.60.3 TWordField.Create . . . . .	511

<b>12 Reference for unit 'dbugintf'</b>	<b>512</b>
12.1 Overview	512
12.2 Writing a debug server	512
12.3 Constants, types and variables	512
12.3.1 Resource strings	512
12.3.2 Constants	513
12.3.3 Types	513
12.3.4 Variables	513
12.4 Procedures and functions	513
12.4.1 GetDebuggingEnabled	513
12.4.2 InitDebugClient	514
12.4.3 SendBoolean	514
12.4.4 SendDateTime	514
12.4.5 SendDebug	515
12.4.6 SendDebugEx	515
12.4.7 SendDebugFmt	515
12.4.8 SendDebugFmtEx	516
12.4.9 SendInteger	516
12.4.10 SendMethodEnter	516
12.4.11 SendMethodExit	517
12.4.12 SendPointer	517
12.4.13 SendSeparator	517
12.4.14 SetDebuggingEnabled	518
12.4.15 StartDebugServer	518
<b>13 Reference for unit 'dbugmsg'</b>	<b>519</b>
13.1 Used units	519
13.2 Overview	519
13.3 Constants, types and variables	519
13.3.1 Constants	519
13.3.2 Types	520
13.4 Procedures and functions	520
13.4.1 DebugMessageName	520
13.4.2 ReadDebugMessageFromStream	520
13.4.3 WriteDebugMessageToStream	520
13.5 TDebugMessage	521
<b>14 Reference for unit 'eventlog'</b>	<b>522</b>
14.1 Used units	522
14.2 Overview	522
14.3 Constants, types and variables	522

14.3.1	Resource strings	522
14.3.2	Types	523
14.4	ELogError	523
14.4.1	Description	523
14.5	TEventLog	524
14.5.1	Description	524
14.5.2	Method overview	524
14.5.3	Property overview	524
14.5.4	TEventLog.Destroy	524
14.5.5	TEventLog.EventTypeToString	525
14.5.6	TEventLog.RegisterMessageFile	525
14.5.7	TEventLog.UnRegisterMessageFile	526
14.5.8	TEventLog.Pause	526
14.5.9	TEventLog.Resume	526
14.5.10	TEventLog.Log	526
14.5.11	TEventLog.Warning	527
14.5.12	TEventLog.Error	527
14.5.13	TEventLog.Debug	527
14.5.14	TEventLog.Info	528
14.5.15	TEventLog.AppendContent	528
14.5.16	TEventLog.Identification	528
14.5.17	TEventLog.LogType	528
14.5.18	TEventLog.Active	529
14.5.19	TEventLog.RaiseExceptionOnError	529
14.5.20	TEventLog.DefaultEventType	529
14.5.21	TEventLog.FileName	529
14.5.22	TEventLog.TimeStampFormat	530
14.5.23	TEventLog.CustomLogType	530
14.5.24	TEventLog.EventIDOffset	530
14.5.25	TEventLog.OnGetCustomCategory	531
14.5.26	TEventLog.OnGetCustomEventID	531
14.5.27	TEventLog.OnGetCustomEvent	531
14.5.28	TEventLog.Paused	532
<b>15</b>	<b>Reference for unit 'ezcgi'</b>	<b>533</b>
15.1	Used units	533
15.2	Overview	533
15.3	Constants, types and variables	533
15.3.1	Constants	533
15.4	ECGIException	533

15.4.1	Description	533
15.5	TEZcgi	534
15.5.1	Description	534
15.5.2	Method overview	534
15.5.3	Property overview	534
15.5.4	TEZcgi.Create	534
15.5.5	TEZcgi.Destroy	534
15.5.6	TEZcgi.Run	535
15.5.7	TEZcgi.WriteContent	535
15.5.8	TEZcgi.PutLine	535
15.5.9	TEZcgi.GetValue	536
15.5.10	TEZcgi.DoPost	536
15.5.11	TEZcgi.DoGet	536
15.5.12	TEZcgi.Values	536
15.5.13	TEZcgi.Names	537
15.5.14	TEZcgi.Variables	537
15.5.15	TEZcgi.VariableCount	538
15.5.16	TEZcgi.Name	538
15.5.17	TEZcgi.Email	538
<b>16</b>	<b>Reference for unit 'fpjson'</b>	<b>539</b>
16.1	Used units	539
16.2	Overview	539
16.3	Constants, types and variables	541
16.3.1	Constants	541
16.3.2	Types	541
16.4	Procedures and functions	545
16.4.1	CreateJSON	545
16.4.2	CreateJSONArray	545
16.4.3	CreateJSONObject	545
16.4.4	GetJSONInstanceType	546
16.4.5	JSONStringToString	546
16.4.6	JSONTypeName	546
16.4.7	SetJSONInstanceType	546
16.4.8	StringToJSONString	547
16.5	TJSONEnum	547
16.6	EJSON	547
16.6.1	Description	547
16.7	TBaseJSONEnumerator	548
16.7.1	Description	548

16.7.2	Method overview	548
16.7.3	Property overview	548
16.7.4	TBaseJSONEnumerator.GetCurrent	548
16.7.5	TBaseJSONEnumerator.MoveNext	548
16.7.6	TBaseJSONEnumerator.Current	548
16.8	TJSONArray	549
16.8.1	Description	549
16.8.2	Method overview	549
16.8.3	Property overview	549
16.8.4	TJSONArray.Create	549
16.8.5	TJSONArray.Destroy	550
16.8.6	TJSONArray.JSONType	550
16.8.7	TJSONArray.Clone	550
16.8.8	TJSONArray.Iterate	551
16.8.9	TJSONArray.IndexOf	551
16.8.10	TJSONArray.GetEnumerator	551
16.8.11	TJSONArray.Clear	551
16.8.12	TJSONArray.Add	552
16.8.13	TJSONArray.Delete	552
16.8.14	TJSONArray.Exchange	552
16.8.15	TJSONArray.Extract	552
16.8.16	TJSONArray.Insert	553
16.8.17	TJSONArray.Move	553
16.8.18	TJSONArray.Remove	553
16.8.19	TJSONArray.Sort	554
16.8.20	TJSONArray.Items	554
16.8.21	TJSONArray.Types	554
16.8.22	TJSONArray.Nulls	554
16.8.23	TJSONArray.Integers	555
16.8.24	TJSONArray.Strings	555
16.8.25	TJSONArray.Floats	556
16.8.26	TJSONArray.Booleans	556
16.8.27	TJSONArray.Arrays	556
16.8.28	TJSONArray.Objects	557
16.9	TJSONBoolean	557
16.9.1	Description	557
16.9.2	Method overview	557
16.9.3	TJSONBoolean.Create	558
16.9.4	TJSONBoolean.JSONType	558
16.9.5	TJSONBoolean.Clear	558



16.9.6 TJSONBoolean.Clone . . . . .	558
16.10 TJSONData . . . . .	559
16.10.1 Description . . . . .	559
16.10.2 Method overview . . . . .	559
16.10.3 Property overview . . . . .	559
16.10.4 TJSONData.JSONType . . . . .	559
16.10.5 TJSONData.Create . . . . .	560
16.10.6 TJSONData.Clear . . . . .	560
16.10.7 TJSONData.DumpJSON . . . . .	560
16.10.8 TJSONData.GetEnumerator . . . . .	561
16.10.9 TJSONData.FindPath . . . . .	561
16.10.10 TJSONData.GetPath . . . . .	563
16.10.11 TJSONData.Clone . . . . .	563
16.10.12 TJSONData.FormatJSON . . . . .	563
16.10.13 TJSONData.CompressedJSON . . . . .	564
16.10.14 TJSONData.Count . . . . .	564
16.10.15 TJSONData.Items . . . . .	564
16.10.16 TJSONData.Value . . . . .	565
16.10.17 TJSONData.AsString . . . . .	565
16.10.18 TJSONData.AsFloat . . . . .	565
16.10.19 TJSONData.AsInteger . . . . .	566
16.10.20 TJSONData.AsBoolean . . . . .	566
16.10.21 TJSONData.IsNull . . . . .	566
16.10.22 TJSONData.AsJSON . . . . .	567
16.11 TJSONFloatNumber . . . . .	567
16.11.1 Description . . . . .	567
16.11.2 Method overview . . . . .	567
16.11.3 TJSONFloatNumber.Create . . . . .	567
16.11.4 TJSONFloatNumber.NumberType . . . . .	567
16.11.5 TJSONFloatNumber.Clear . . . . .	568
16.11.6 TJSONFloatNumber.Clone . . . . .	568
16.12 TJSONIntegerNumber . . . . .	568
16.12.1 Description . . . . .	568
16.12.2 Method overview . . . . .	568
16.12.3 TJSONIntegerNumber.Create . . . . .	568
16.12.4 TJSONIntegerNumber.NumberType . . . . .	569
16.12.5 TJSONIntegerNumber.Clear . . . . .	569
16.12.6 TJSONIntegerNumber.Clone . . . . .	569
16.13 TJSONNull . . . . .	569
16.13.1 Description . . . . .	569

16.13.2 Method overview . . . . .	569
16.13.3 TJSONNull.JSONType . . . . .	570
16.13.4 TJSONNull.Clear . . . . .	570
16.13.5 TJSONNull.Clone . . . . .	570
16.14 TJSONNumber . . . . .	570
16.14.1 Description . . . . .	570
16.14.2 Method overview . . . . .	570
16.14.3 TJSONNumber.JSONType . . . . .	571
16.14.4 TJSONNumber.NumberType . . . . .	571
16.15 TJSONObject . . . . .	571
16.15.1 Description . . . . .	571
16.15.2 Method overview . . . . .	571
16.15.3 Property overview . . . . .	572
16.15.4 TJSONObject.Create . . . . .	572
16.15.5 TJSONObject.Destroy . . . . .	573
16.15.6 TJSONObject.JSONType . . . . .	573
16.15.7 TJSONObject.Clone . . . . .	573
16.15.8 TJSONObject.GetEnumerator . . . . .	573
16.15.9 TJSONObject.Iterate . . . . .	574
16.15.10 TJSONObject.IndexOf . . . . .	574
16.15.11 TJSONObject.IndexOfName . . . . .	574
16.15.12 TJSONObject.Find . . . . .	574
16.15.13 TJSONObject.Get . . . . .	575
16.15.14 TJSONObject.Clear . . . . .	575
16.15.15 TJSONObject.Add . . . . .	576
16.15.16 TJSONObject.Delete . . . . .	576
16.15.17 TJSONObject.Remove . . . . .	576
16.15.18 TJSONObject.UnquotedMemberNames . . . . .	577
16.15.19 TJSONObject.Names . . . . .	577
16.15.20 TJSONObject.Elements . . . . .	577
16.15.21 TJSONObject.Types . . . . .	578
16.15.22 TJSONObject.Nulls . . . . .	578
16.15.23 TJSONObject.Floats . . . . .	578
16.15.24 TJSONObject.Integers . . . . .	578
16.15.25 TJSONObject.Strings . . . . .	579
16.15.26 TJSONObject.Booleans . . . . .	579
16.15.27 TJSONObject.Arrays . . . . .	579
16.15.28 TJSONObject.Objects . . . . .	580
16.16 TJJSONString . . . . .	580
16.16.1 Description . . . . .	580

16.16.2 Method overview . . . . .	580
16.16.3 TJSONString.Create . . . . .	580
16.16.4 TJSONString.JSONType . . . . .	581
16.16.5 TJSONString.Clear . . . . .	581
16.16.6 TJSONString.Clone . . . . .	581
<b>17 Reference for unit 'fpmimeTypes' . . . . .</b>	<b>582</b>
17.1 Used units . . . . .	582
17.2 Overview . . . . .	582
17.3 Procedures and functions . . . . .	582
17.3.1 MimeTypes . . . . .	582
17.4 TFPMimeType . . . . .	583
17.4.1 Description . . . . .	583
17.4.2 Method overview . . . . .	583
17.4.3 TFPMimeType.Create . . . . .	583
17.4.4 TFPMimeType.Destroy . . . . .	583
17.4.5 TFPMimeType.Clear . . . . .	584
17.4.6 TFPMimeType.LoadKnownTypes . . . . .	584
17.4.7 TFPMimeType.GetNextExtension . . . . .	584
17.4.8 TFPMimeType.LoadFromStream . . . . .	584
17.4.9 TFPMimeType.LoadFromFile . . . . .	585
17.4.10 TFPMimeType.AddType . . . . .	585
17.4.11 TFPMimeType.GetMimeExtensions . . . . .	585
17.4.12 TFPMimeType.GetMimeType . . . . .	585
17.4.13 TFPMimeType.GetKnownMimeType . . . . .	586
17.4.14 TFPMimeType.GetKnownExtensions . . . . .	586
17.5 TMimeType . . . . .	586
17.5.1 Description . . . . .	586
17.5.2 Method overview . . . . .	586
17.5.3 Property overview . . . . .	586
17.5.4 TMimeType.Create . . . . .	587
17.5.5 TMimeType.MergeExtensions . . . . .	587
17.5.6 TMimeType.MimeType . . . . .	587
17.5.7 TMimeType.Extensions . . . . .	587
<b>18 Reference for unit 'fptimer' . . . . .</b>	<b>588</b>
18.1 Used units . . . . .	588
18.2 Overview . . . . .	588
18.3 Constants, types and variables . . . . .	588
18.3.1 Types . . . . .	588
18.3.2 Variables . . . . .	588

18.4	TFPCustomTimer	589
18.4.1	Description	589
18.4.2	Method overview	589
18.4.3	TFPCustomTimer.Create	589
18.4.4	TFPCustomTimer.Destroy	589
18.4.5	TFPCustomTimer.StartTimer	589
18.4.6	TFPCustomTimer.StopTimer	590
18.5	TFPTimer	590
18.5.1	Description	590
18.5.2	Property overview	590
18.5.3	TFPTimer.Enabled	590
18.5.4	TFPTimer.Interval	591
18.5.5	TFPTimer.UseTimerThread	591
18.5.6	TFPTimer.OnTimer	591
18.5.7	TFPTimer.OnStartTimer	591
18.5.8	TFPTimer.OnStopTimer	591
18.6	TFPTimerDriver	592
18.6.1	Description	592
18.6.2	Method overview	592
18.6.3	Property overview	592
18.6.4	TFPTimerDriver.Create	592
18.6.5	TFPTimerDriver.StartTimer	592
18.6.6	TFPTimerDriver.StopTimer	593
18.6.7	TFPTimerDriver.Timer	593
18.6.8	TFPTimerDriver.TimerStarted	593
<b>19</b>	<b>Reference for unit 'gettext'</b>	<b>594</b>
19.1	Used units	594
19.2	Overview	594
19.3	Constants, types and variables	594
19.3.1	Constants	594
19.3.2	Types	594
19.4	Procedures and functions	595
19.4.1	GetLanguageIDs	595
19.4.2	TranslateResourceStrings	595
19.4.3	TranslateUnitResourceStrings	596
19.5	TMOFileHeader	596
19.6	TMOStringInfo	596
19.7	EMOFileError	596
19.7.1	Description	596

19.8	TMOFile	597
19.8.1	Description	597
19.8.2	Method overview	597
19.8.3	TMOFile.Create	597
19.8.4	TMOFile.Destroy	597
19.8.5	TMOFile.Translate	597
<b>20</b>	<b>Reference for unit 'IBConnection'</b>	<b>599</b>
20.1	Used units	599
20.2	Constants, types and variables	599
20.2.1	Constants	599
20.2.2	Types	599
20.3	TDatabaseInfo	599
20.4	EIBDatabaseError	600
20.4.1	Description	600
20.4.2	Property overview	600
20.4.3	EIBDatabaseError.GDSErrorCode	600
20.5	TIBConnection	600
20.5.1	Description	600
20.5.2	Method overview	601
20.5.3	Property overview	602
20.5.4	TIBConnection.Create	602
20.5.5	TIBConnection.GetConnectionInfo	602
20.5.6	TIBConnection.CreateDB	602
20.5.7	TIBConnection.DropDB	603
20.5.8	TIBConnection.BlobSegmentSize	603
20.5.9	TIBConnection.ODSMajorVersion	603
20.5.10	TIBConnection.DatabaseName	603
20.5.11	TIBConnection.Dialect	604
20.5.12	TIBConnection.CheckTransactionParams	604
20.5.13	TIBConnection.KeepConnection	604
20.5.14	TIBConnection.LoginPrompt	604
20.5.15	TIBConnection.Params	605
20.5.16	TIBConnection.OnLogin	605
20.5.17	TIBConnection.Port	605
20.5.18	TIBConnection.UseConnectionCharSetIfNone	605
20.5.19	TIBConnection.WireCompression	605
20.6	TIBConnectionDef	606
20.6.1	Description	606
20.6.2	Method overview	606

20.6.3	TIBConnectionDef.TypeName	606
20.6.4	TIBConnectionDef.ConnectionClass	606
20.6.5	TIBConnectionDef.Description	606
20.6.6	TIBConnectionDef.DefaultLibraryName	607
20.6.7	TIBConnectionDef.LoadFunction	607
20.6.8	TIBConnectionDef.UnLoadFunction	607
20.6.9	TIBConnectionDef.LoadedLibraryName	607
20.7	TIBCursor	607
20.7.1	Description	607
20.8	TIBTrans	607
20.8.1	Description	607
<b>21</b>	<b>Reference for unit 'idea'</b>	<b>608</b>
21.1	Used units	608
21.2	Overview	608
21.3	Constants, types and variables	608
21.3.1	Constants	608
21.3.2	Types	609
21.4	Procedures and functions	609
21.4.1	CipherIdea	609
21.4.2	DeKeyIdea	609
21.4.3	EnKeyIdea	610
21.5	EIDEAError	610
21.5.1	Description	610
21.6	TIDEADeCryptStream	610
21.6.1	Description	610
21.6.2	Method overview	610
21.6.3	TIDEADeCryptStream.Create	611
21.6.4	TIDEADeCryptStream.Read	611
21.6.5	TIDEADeCryptStream.Seek	611
21.7	TIDEAEncryptStream	612
21.7.1	Description	612
21.7.2	Method overview	612
21.7.3	TIDEAEncryptStream.Create	612
21.7.4	TIDEAEncryptStream.Destroy	612
21.7.5	TIDEAEncryptStream.Write	613
21.7.6	TIDEAEncryptStream.Seek	613
21.7.7	TIDEAEncryptStream.Flush	613
21.8	TIDEAStream	613
21.8.1	Description	613

21.8.2	Method overview	614
21.8.3	Property overview	614
21.8.4	TIDEAStream.Create	614
21.8.5	TIDEAStream.Key	614
<b>22</b>	<b>Reference for unit 'inicol'</b>	<b>615</b>
22.1	Used units	615
22.2	Overview	615
22.3	Constants, types and variables	615
22.3.1	Constants	615
22.4	EIniCol	615
22.4.1	Description	615
22.5	TIniCollection	616
22.5.1	Description	616
22.5.2	Method overview	616
22.5.3	Property overview	616
22.5.4	TIniCollection.Load	616
22.5.5	TIniCollection.Save	616
22.5.6	TIniCollection.SaveToIni	617
22.5.7	TIniCollection.SaveToFile	617
22.5.8	TIniCollection.LoadFromIni	618
22.5.9	TIniCollection.LoadFromFile	618
22.5.10	TIniCollection.Prefix	618
22.5.11	TIniCollection.SectionPrefix	619
22.5.12	TIniCollection.FileName	619
22.5.13	TIniCollection.GlobalSection	619
22.6	TIniCollectionItem	619
22.6.1	Description	619
22.6.2	Method overview	620
22.6.3	Property overview	620
22.6.4	TIniCollectionItem.SaveToIni	620
22.6.5	TIniCollectionItem.LoadFromIni	620
22.6.6	TIniCollectionItem.SaveToFile	620
22.6.7	TIniCollectionItem.LoadFromFile	621
22.6.8	TIniCollectionItem.SectionName	621
22.7	TNamedIniCollection	621
22.7.1	Description	621
22.7.2	Method overview	621
22.7.3	Property overview	622
22.7.4	TNamedIniCollection.IndexOfUserData	622

22.7.5	TNamedIniCollection.IndexOfName	622
22.7.6	TNamedIniCollection.FindByName	622
22.7.7	TNamedIniCollection.FindByUserData	623
22.7.8	TNamedIniCollection.NamedItems	623
22.8	TNamedIniCollectionItem	623
22.8.1	Description	623
22.8.2	Property overview	623
22.8.3	TNamedIniCollectionItem.UserData	623
22.8.4	TNamedIniCollectionItem.Name	624
<b>23</b>	<b>Reference for unit 'IniFiles'</b>	<b>625</b>
23.1	Used units	625
23.2	Overview	625
23.3	Constants, types and variables	625
23.3.1	Types	625
23.4	TCustomIniFile	627
23.4.1	Description	627
23.4.2	Method overview	627
23.4.3	Property overview	628
23.4.4	TCustomIniFile.Create	628
23.4.5	TCustomIniFile.Destroy	629
23.4.6	TCustomIniFile.SetBoolStringValue	629
23.4.7	TCustomIniFile.SectionExists	629
23.4.8	TCustomIniFile.ReadString	629
23.4.9	TCustomIniFile.WriteString	630
23.4.10	TCustomIniFile.ReadInteger	630
23.4.11	TCustomIniFile.WriteInteger	630
23.4.12	TCustomIniFile.ReadInt64	630
23.4.13	TCustomIniFile.WriteInt64	631
23.4.14	TCustomIniFile.ReadBool	631
23.4.15	TCustomIniFile.WriteBool	631
23.4.16	TCustomIniFile.ReadDate	632
23.4.17	TCustomIniFile.ReadDateTime	632
23.4.18	TCustomIniFile.ReadFloat	632
23.4.19	TCustomIniFile.ReadTime	632
23.4.20	TCustomIniFile.ReadBinaryStream	633
23.4.21	TCustomIniFile.WriteDate	633
23.4.22	TCustomIniFile.WriteDateTime	633
23.4.23	TCustomIniFile.WriteFloat	634
23.4.24	TCustomIniFile.WriteTime	634



---

23.4.25	<a href="#">TCustomIniFile.WriteBinaryStream</a>	634
23.4.26	<a href="#">TCustomIniFile.ReadSection</a>	634
23.4.27	<a href="#">TCustomIniFile.ReadSections</a>	635
23.4.28	<a href="#">TCustomIniFile.ReadSectionValues</a>	635
23.4.29	<a href="#">TCustomIniFile.EraseSection</a>	635
23.4.30	<a href="#">TCustomIniFile.DeleteKey</a>	636
23.4.31	<a href="#">TCustomIniFile.UpdateFile</a>	636
23.4.32	<a href="#">TCustomIniFile.ValueExists</a>	636
23.4.33	<a href="#">TCustomIniFile.Encoding</a>	636
23.4.34	<a href="#">TCustomIniFile.FileName</a>	637
23.4.35	<a href="#">TCustomIniFile.Options</a>	637
23.4.36	<a href="#">TCustomIniFile.EscapeLineFeeds</a>	637
23.4.37	<a href="#">TCustomIniFile.CaseSensitive</a>	637
23.4.38	<a href="#">TCustomIniFile.StripQuotes</a>	638
23.4.39	<a href="#">TCustomIniFile.FormatSettingsActive</a>	638
23.4.40	<a href="#">TCustomIniFile.BoolTrueStrings</a>	638
23.4.41	<a href="#">TCustomIniFile.BoolFalseStrings</a>	639
23.4.42	<a href="#">TCustomIniFile.OwnsEncoding</a>	639
23.5	<a href="#">THashedStringList</a>	639
23.5.1	<a href="#">Description</a>	639
23.5.2	<a href="#">Method overview</a>	639
23.5.3	<a href="#">THashedStringList.Destroy</a>	639
23.5.4	<a href="#">THashedStringList.IndexOf</a>	640
23.5.5	<a href="#">THashedStringList.IndexOfName</a>	640
23.6	<a href="#">TIniFile</a>	640
23.6.1	<a href="#">Description</a>	640
23.6.2	<a href="#">Method overview</a>	641
23.6.3	<a href="#">Property overview</a>	641
23.6.4	<a href="#">TIniFile.Create</a>	641
23.6.5	<a href="#">TIniFile.Destroy</a>	641
23.6.6	<a href="#">TIniFile.ReadString</a>	642
23.6.7	<a href="#">TIniFile.WriteString</a>	642
23.6.8	<a href="#">TIniFile.ReadSection</a>	642
23.6.9	<a href="#">TIniFile.ReadSectionRaw</a>	642
23.6.10	<a href="#">TIniFile.ReadSections</a>	643
23.6.11	<a href="#">TIniFile.ReadSectionValues</a>	643
23.6.12	<a href="#">TIniFile.EraseSection</a>	643
23.6.13	<a href="#">TIniFile.DeleteKey</a>	643
23.6.14	<a href="#">TIniFile.UpdateFile</a>	644
23.6.15	<a href="#">TIniFile.Stream</a>	644

---

23.6.16	TIniFile.CacheUpdates	644
23.6.17	TIniFile.WriteBOM	645
23.7	TIniFileKey	645
23.7.1	Description	645
23.7.2	Method overview	645
23.7.3	Property overview	645
23.7.4	TIniFileKey.Create	645
23.7.5	TIniFileKey.Ident	645
23.7.6	TIniFileKey.Value	646
23.8	TIniFileKeyList	646
23.8.1	Description	646
23.8.2	Method overview	646
23.8.3	Property overview	646
23.8.4	TIniFileKeyList.Destroy	646
23.8.5	TIniFileKeyList.Clear	646
23.8.6	TIniFileKeyList.Items	647
23.9	TIniFileSection	647
23.9.1	Description	647
23.9.2	Method overview	647
23.9.3	Property overview	647
23.9.4	TIniFileSection.Empty	647
23.9.5	TIniFileSection.Create	648
23.9.6	TIniFileSection.Destroy	648
23.9.7	TIniFileSection.Name	648
23.9.8	TIniFileSection.KeyList	648
23.10	TIniFileSectionList	649
23.10.1	Description	649
23.10.2	Method overview	649
23.10.3	Property overview	649
23.10.4	TIniFileSectionList.Destroy	649
23.10.5	TIniFileSectionList.Clear	649
23.10.6	TIniFileSectionList.Items	649
23.11	TMemIniFile	650
23.11.1	Description	650
23.11.2	Method overview	650
23.11.3	TMemIniFile.Create	650
23.11.4	TMemIniFile.Clear	650
23.11.5	TMemIniFile.GetStrings	651
23.11.6	TMemIniFile.Rename	651
23.11.7	TMemIniFile.SetStrings	651

23.12 TStringHash . . . . .	651
23.12.1 Description . . . . .	651
23.12.2 Method overview . . . . .	652
23.12.3 Property overview . . . . .	652
23.12.4 TStringHash.Create . . . . .	652
23.12.5 TStringHash.Destroy . . . . .	652
23.12.6 TStringHash.Add . . . . .	652
23.12.7 TStringHash.Clear . . . . .	653
23.12.8 TStringHash.Modify . . . . .	653
23.12.9 TStringHash.Remove . . . . .	653
23.12.10 TStringHash.ValueOf . . . . .	653
23.12.11 TStringHash.AddReplacesExisting . . . . .	654
<b>24 Reference for unit 'iostream'</b>	<b>655</b>
24.1 Used units . . . . .	655
24.2 Overview . . . . .	655
24.3 Constants, types and variables . . . . .	655
24.3.1 Types . . . . .	655
24.4 EIOStreamError . . . . .	656
24.4.1 Description . . . . .	656
24.5 TIOStream . . . . .	656
24.5.1 Description . . . . .	656
24.5.2 Method overview . . . . .	656
24.5.3 TIOStream.Create . . . . .	656
24.5.4 TIOStream.Read . . . . .	656
24.5.5 TIOStream.Write . . . . .	657
24.5.6 TIOStream.Seek . . . . .	657
<b>25 Reference for unit 'libtar'</b>	<b>658</b>
25.1 Used units . . . . .	658
25.2 Overview . . . . .	658
25.3 Constants, types and variables . . . . .	658
25.3.1 Constants . . . . .	658
25.3.2 Types . . . . .	659
25.4 Procedures and functions . . . . .	660
25.4.1 ClearDirRec . . . . .	660
25.4.2 ConvertFilename . . . . .	660
25.4.3 FileTimeGMT . . . . .	661
25.4.4 PermissionString . . . . .	661
25.5 TTarDirRec . . . . .	661
25.6 TTarArchive . . . . .	662

25.6.1	Description	662
25.6.2	Method overview	662
25.6.3	TTarArchive.Create	662
25.6.4	TTarArchive.Destroy	662
25.6.5	TTarArchive.Reset	662
25.6.6	TTarArchive.FindNext	663
25.6.7	TTarArchive.ReadFile	663
25.6.8	TTarArchive.GetFilePos	663
25.6.9	TTarArchive.SetFilePos	664
25.7	TTarWriter	664
25.7.1	Description	664
25.7.2	Method overview	664
25.7.3	Property overview	664
25.7.4	TTarWriter.Create	664
25.7.5	TTarWriter.Destroy	665
25.7.6	TTarWriter.AddFile	665
25.7.7	TTarWriter.AddStream	665
25.7.8	TTarWriter.AddString	666
25.7.9	TTarWriter.AddDir	666
25.7.10	TTarWriter.AddSymbolicLink	666
25.7.11	TTarWriter.AddLink	667
25.7.12	TTarWriter.AddVolumeHeader	667
25.7.13	TTarWriter.Finalize	667
25.7.14	TTarWriter.Permissions	667
25.7.15	TTarWriter.UID	668
25.7.16	TTarWriter.GID	668
25.7.17	TTarWriter.UserName	668
25.7.18	TTarWriter.GroupName	668
25.7.19	TTarWriter.Mode	669
25.7.20	TTarWriter.Magic	669
<b>26</b>	<b>Reference for unit 'memds'</b>	<b>670</b>
26.1	Used units	670
26.2	Overview	670
26.3	Constants, types and variables	670
26.3.1	Constants	670
26.4	MDSError	671
26.4.1	Description	671
26.5	TMemDataset	672
26.5.1	Description	672

26.5.2	Method overview	673
26.5.3	Property overview	674
26.5.4	TMemDataset.Create	674
26.5.5	TMemDataset.Destroy	674
26.5.6	TMemDataset.BookmarkValid	675
26.5.7	TMemDataset.CompareBookmarks	675
26.5.8	TMemDataset.CreateBlobStream	675
26.5.9	TMemDataset.Locate	676
26.5.10	TMemDataset.Lookup	676
26.5.11	TMemDataset.CreateTable	676
26.5.12	TMemDataset.DataSize	677
26.5.13	TMemDataset.Clear	677
26.5.14	TMemDataset.SaveToFile	677
26.5.15	TMemDataset.SaveToStream	678
26.5.16	TMemDataset.LoadFromStream	678
26.5.17	TMemDataset.LoadFromFile	678
26.5.18	TMemDataset.CopyFromDataset	679
26.5.19	TMemDataset.FileModified	680
26.5.20	TMemDataset.Filter	680
26.5.21	TMemDataset.FileName	680
26.5.22	TMemDataset.Filtered	681
26.5.23	TMemDataset.Active	681
26.5.24	TMemDataset.FieldDefs	681
26.5.25	TMemDataset.BeforeOpen	681
26.5.26	TMemDataset.AfterOpen	681
26.5.27	TMemDataset.BeforeClose	682
26.5.28	TMemDataset.AfterClose	682
26.5.29	TMemDataset.BeforeInsert	682
26.5.30	TMemDataset.AfterInsert	682
26.5.31	TMemDataset.BeforeEdit	682
26.5.32	TMemDataset.AfterEdit	682
26.5.33	TMemDataset.BeforePost	683
26.5.34	TMemDataset.AfterPost	683
26.5.35	TMemDataset.BeforeCancel	683
26.5.36	TMemDataset.AfterCancel	683
26.5.37	TMemDataset.BeforeDelete	683
26.5.38	TMemDataset.AfterDelete	683
26.5.39	TMemDataset.BeforeScroll	684
26.5.40	TMemDataset.AfterScroll	684
26.5.41	TMemDataset.OnDeleteError	684

26.5.42 TMemDataset.OnEditError . . . . .	684
26.5.43 TMemDataset.OnNewRecord . . . . .	684
26.5.44 TMemDataset.OnPostError . . . . .	685
26.5.45 TMemDataset.OnFilterRecord . . . . .	685
<b>27 Reference for unit 'MSSQLConn'</b>	<b>686</b>
27.1 Used units . . . . .	686
27.2 Overview . . . . .	686
27.3 Constants, types and variables . . . . .	686
27.3.1 Variables . . . . .	686
27.4 EMSSQLDatabaseError . . . . .	686
27.4.1 Description . . . . .	686
27.4.2 Property overview . . . . .	686
27.4.3 EMSSQLDatabaseError.DBErrorCode . . . . .	687
27.5 TMSSQLConnection . . . . .	687
27.5.1 Description . . . . .	687
27.5.2 Method overview . . . . .	687
27.5.3 Property overview . . . . .	688
27.5.4 TMSSQLConnection.Create . . . . .	688
27.5.5 TMSSQLConnection.GetConnectionInfo . . . . .	688
27.5.6 TMSSQLConnection.CreateDB . . . . .	688
27.5.7 TMSSQLConnection.DropDB . . . . .	688
27.5.8 TMSSQLConnection.Password . . . . .	688
27.5.9 TMSSQLConnection.Transaction . . . . .	689
27.5.10 TMSSQLConnection.UserName . . . . .	689
27.5.11 TMSSQLConnection.CharSet . . . . .	689
27.5.12 TMSSQLConnection.HostName . . . . .	689
27.5.13 TMSSQLConnection.Connected . . . . .	689
27.5.14 TMSSQLConnection.Role . . . . .	690
27.5.15 TMSSQLConnection.DatabaseName . . . . .	690
27.5.16 TMSSQLConnection.KeepConnection . . . . .	690
27.5.17 TMSSQLConnection.LoginPrompt . . . . .	690
27.5.18 TMSSQLConnection.Params . . . . .	690
27.5.19 TMSSQLConnection.OnLogin . . . . .	690
27.6 TMSSQLConnectionDef . . . . .	691
27.6.1 Method overview . . . . .	691
27.6.2 TMSSQLConnectionDef.TypeName . . . . .	691
27.6.3 TMSSQLConnectionDef.ConnectionClass . . . . .	691
27.6.4 TMSSQLConnectionDef.Description . . . . .	691
27.6.5 TMSSQLConnectionDef.DefaultLibraryName . . . . .	691

27.6.6	TMSSQLConnectionDef.LoadFunction	691
27.6.7	TMSSQLConnectionDef.UnLoadFunction	691
27.6.8	TMSSQLConnectionDef.LoadedLibraryName	691
27.7	TSybaseConnection	692
27.7.1	Description	692
27.7.2	Method overview	692
27.7.3	TSybaseConnection.Create	692
27.8	TSybaseConnectionDef	692
27.8.1	Method overview	692
27.8.2	TSybaseConnectionDef.TypeName	692
27.8.3	TSybaseConnectionDef.ConnectionClass	692
27.8.4	TSybaseConnectionDef.Description	693
<b>28</b>	<b>Reference for unit 'nullstream'</b>	<b>694</b>
28.1	Used units	694
28.2	Overview	694
28.3	ENullStreamError	694
28.3.1	Description	694
28.4	TNullStream	694
28.4.1	Description	694
28.4.2	Method overview	695
28.4.3	TNullStream.Read	695
28.4.4	TNullStream.Write	695
28.4.5	TNullStream.Seek	695
28.4.6	TNullStream.Create	696
<b>29</b>	<b>Reference for unit 'Pipes'</b>	<b>697</b>
29.1	Used units	697
29.2	Overview	697
29.3	Constants, types and variables	697
29.3.1	Constants	697
29.4	Procedures and functions	697
29.4.1	CreatePipeHandles	697
29.4.2	CreatePipeStreams	698
29.5	EPipeCreation	698
29.5.1	Description	698
29.6	EPipeError	698
29.6.1	Description	698
29.7	EPipeSeek	698
29.7.1	Description	698
29.8	TInputPipeStream	699

29.8.1	Description	699
29.8.2	Method overview	699
29.8.3	Property overview	699
29.8.4	TInputPipeStream.Destroy	699
29.8.5	TInputPipeStream.Write	699
29.8.6	TInputPipeStream.Seek	700
29.8.7	TInputPipeStream.Read	700
29.8.8	TInputPipeStream.NumBytesAvailable	700
29.9	TOutputPipeStream	701
29.9.1	Description	701
29.9.2	Method overview	701
29.9.3	TOutputPipeStream.Destroy	701
29.9.4	TOutputPipeStream.Seek	701
29.9.5	TOutputPipeStream.Read	701
<b>30</b>	<b>Reference for unit 'pooledmm'</b>	<b>702</b>
30.1	Used units	702
30.2	Overview	702
30.3	Constants, types and variables	702
30.3.1	Types	702
30.4	TPooledMemManagerItem	702
30.5	TNonFreePooledMemManager	703
30.5.1	Description	703
30.5.2	Method overview	703
30.5.3	Property overview	703
30.5.4	TNonFreePooledMemManager.Clear	703
30.5.5	TNonFreePooledMemManager.Create	703
30.5.6	TNonFreePooledMemManager.Destroy	704
30.5.7	TNonFreePooledMemManager.NewItem	704
30.5.8	TNonFreePooledMemManager.EnumerateItems	704
30.5.9	TNonFreePooledMemManager.ItemSize	704
30.6	TPooledMemManager	705
30.6.1	Description	705
30.6.2	Method overview	705
30.6.3	Property overview	705
30.6.4	TPooledMemManager.Clear	705
30.6.5	TPooledMemManager.Create	705
30.6.6	TPooledMemManager.Destroy	706
30.6.7	TPooledMemManager.MinimumFreeCount	706
30.6.8	TPooledMemManager.MaximumFreeCountRatio	706



30.6.9 TPooledMemManager.Count . . . . .	706
30.6.10 TPooledMemManager.FreeCount . . . . .	707
30.6.11 TPooledMemManager.AllocatedCount . . . . .	707
30.6.12 TPooledMemManager.FreedCount . . . . .	707
<b>31 Reference for unit 'process'</b>	<b>708</b>
31.1 Used units . . . . .	708
31.2 Overview . . . . .	708
31.3 Constants, types and variables . . . . .	708
31.3.1 Types . . . . .	708
31.3.2 Variables . . . . .	711
31.4 Procedures and functions . . . . .	712
31.4.1 CommandToList . . . . .	712
31.4.2 RunCommand . . . . .	712
31.4.3 RunCommandIndir . . . . .	712
31.5 EProcess . . . . .	713
31.5.1 Description . . . . .	713
31.6 TPROCESS . . . . .	713
31.6.1 Description . . . . .	713
31.6.2 Method overview . . . . .	714
31.6.3 Property overview . . . . .	715
31.6.4 TPROCESS.Create . . . . .	715
31.6.5 TPROCESS.Destroy . . . . .	716
31.6.6 TPROCESS.Execute . . . . .	716
31.6.7 TPROCESS.CloseInput . . . . .	716
31.6.8 TPROCESS.CloseOutput . . . . .	717
31.6.9 TPROCESS.CloseStderr . . . . .	717
31.6.10 TPROCESS.Resume . . . . .	717
31.6.11 TPROCESS.Suspend . . . . .	717
31.6.12 TPROCESS.Terminate . . . . .	718
31.6.13 TPROCESS.WaitOnExit . . . . .	718
31.6.14 TPROCESS.ReadInputStream . . . . .	718
31.6.15 TPROCESS.RunCommandLoop . . . . .	719
31.6.16 TPROCESS.WindowRect . . . . .	719
31.6.17 TPROCESS.Handle . . . . .	719
31.6.18 TPROCESS.ProcessHandle . . . . .	720
31.6.19 TPROCESS.ThreadHandle . . . . .	720
31.6.20 TPROCESS.ProcessID . . . . .	720
31.6.21 TPROCESS.ThreadID . . . . .	720
31.6.22 TPROCESS.Input . . . . .	721

31.6.23 TPROCESS.Output . . . . .	721
31.6.24 TPROCESS.Stderr . . . . .	721
31.6.25 TPROCESS.ExitStatus . . . . .	722
31.6.26 TPROCESS.ExitCode . . . . .	722
31.6.27 TPROCESS.InheritHandles . . . . .	722
31.6.28 TPROCESS.OnRunCommandEvent . . . . .	723
31.6.29 TPROCESS.RunCommandSleepTime . . . . .	723
31.6.30 TPROCESS.PipeBufferSize . . . . .	723
31.6.31 TPROCESS.Active . . . . .	723
31.6.32 TPROCESS.ApplicationName . . . . .	724
31.6.33 TPROCESS.CommandLine . . . . .	724
31.6.34 TPROCESS.Executable . . . . .	724
31.6.35 TPROCESS.Parameters . . . . .	725
31.6.36 TPROCESS.ConsoleTitle . . . . .	726
31.6.37 TPROCESS.CurrentDirectory . . . . .	726
31.6.38 TPROCESS.Desktop . . . . .	726
31.6.39 TPROCESS.Environment . . . . .	726
31.6.40 TPROCESS.Options . . . . .	727
31.6.41 TPROCESS.Priority . . . . .	727
31.6.42 TPROCESS.StartupOptions . . . . .	728
31.6.43 TPROCESS.Running . . . . .	729
31.6.44 TPROCESS.ShowWindow . . . . .	729
31.6.45 TPROCESS.WindowColumns . . . . .	729
31.6.46 TPROCESS.WindowHeight . . . . .	730
31.6.47 TPROCESS.WindowLeft . . . . .	730
31.6.48 TPROCESS.WindowRows . . . . .	730
31.6.49 TPROCESS.WindowTop . . . . .	731
31.6.50 TPROCESS.WindowWidth . . . . .	731
31.6.51 TPROCESS.FillAttribute . . . . .	731
31.6.52 TPROCESS.XTermProgram . . . . .	732
<b>32 Reference for unit 'RttiUtils'</b>	<b>733</b>
32.1 Used units . . . . .	733
32.2 Overview . . . . .	733
32.3 Constants, types and variables . . . . .	733
32.3.1 Constants . . . . .	733
32.3.2 Types . . . . .	733
32.3.3 Variables . . . . .	734
32.4 Procedures and functions . . . . .	735
32.4.1 CreateStoredItem . . . . .	735

32.4.2	ParseStoredItem	735
32.4.3	UpdateStoredList	735
32.5	TPropInfoList	736
32.5.1	Description	736
32.5.2	Method overview	736
32.5.3	Property overview	736
32.5.4	TPropInfoList.Create	736
32.5.5	TPropInfoList.Destroy	736
32.5.6	TPropInfoList.Contains	737
32.5.7	TPropInfoList.Find	737
32.5.8	TPropInfoList.Delete	737
32.5.9	TPropInfoList.Intersect	737
32.5.10	TPropInfoList.Count	738
32.5.11	TPropInfoList.Items	738
32.6	TPropsStorage	738
32.6.1	Description	738
32.6.2	Method overview	738
32.6.3	Property overview	739
32.6.4	TPropsStorage.StoreAnyProperty	739
32.6.5	TPropsStorage.LoadAnyProperty	739
32.6.6	TPropsStorage.StoreProperties	739
32.6.7	TPropsStorage.LoadProperties	740
32.6.8	TPropsStorage.LoadObjectsProps	740
32.6.9	TPropsStorage.StoreObjectsProps	740
32.6.10	TPropsStorage.Options	741
32.6.11	TPropsStorage.AObject	741
32.6.12	TPropsStorage.Prefix	742
32.6.13	TPropsStorage.Section	742
32.6.14	TPropsStorage.OnReadString	742
32.6.15	TPropsStorage.OnWriteString	742
32.6.16	TPropsStorage.OnEraseSection	743
<b>33</b>	<b>Reference for unit 'simpleipc'</b>	<b>744</b>
33.1	Used units	744
33.2	Overview	744
33.3	Constants, types and variables	745
33.3.1	Resource strings	745
33.3.2	Constants	745
33.3.3	Types	745
33.3.4	Variables	746

33.4	TMsgHeader	747
33.5	EIPCErrors	747
33.5.1	Description	747
33.6	TIPCClientComm	747
33.6.1	Description	747
33.6.2	Method overview	748
33.6.3	Property overview	748
33.6.4	TIPCClientComm.Create	748
33.6.5	TIPCClientComm.Connect	748
33.6.6	TIPCClientComm.Disconnect	748
33.6.7	TIPCClientComm.ServerRunning	749
33.6.8	TIPCClientComm.SendMessage	749
33.6.9	TIPCClientComm.Owner	749
33.7	TIPCServerComm	750
33.7.1	Description	750
33.7.2	Method overview	750
33.7.3	Property overview	750
33.7.4	TIPCServerComm.Create	750
33.7.5	TIPCServerComm.StartServer	750
33.7.6	TIPCServerComm.StopServer	751
33.7.7	TIPCServerComm.PeekMessage	751
33.7.8	TIPCServerComm.ReadMessage	751
33.7.9	TIPCServerComm.Owner	752
33.7.10	TIPCServerComm.InstanceID	752
33.8	TIPCServerMsg	752
33.8.1	Description	752
33.8.2	Method overview	752
33.8.3	Property overview	752
33.8.4	TIPCServerMsg.Create	753
33.8.5	TIPCServerMsg.Destroy	753
33.8.6	TIPCServerMsg.Stream	753
33.8.7	TIPCServerMsg.MsgType	753
33.8.8	TIPCServerMsg.OwnsStream	754
33.8.9	TIPCServerMsg.StringMessage	754
33.9	TIPCServerMsgQueue	754
33.9.1	Description	754
33.9.2	Method overview	754
33.9.3	Property overview	754
33.9.4	TIPCServerMsgQueue.Create	755
33.9.5	TIPCServerMsgQueue.Destroy	755

33.9.6	TIPCServerMsgQueue.Clear	755
33.9.7	TIPCServerMsgQueue.Push	755
33.9.8	TIPCServerMsgQueue.Pop	756
33.9.9	TIPCServerMsgQueue.Count	756
33.9.10	TIPCServerMsgQueue.MaxCount	756
33.9.11	TIPCServerMsgQueue.MaxAction	756
33.10	TSimpleIPC	757
33.10.1	Description	757
33.10.2	Property overview	757
33.10.3	TSimpleIPC.Active	757
33.10.4	TSimpleIPC.ServerID	757
33.11	TSimpleIPCClient	758
33.11.1	Description	758
33.11.2	Method overview	758
33.11.3	Property overview	758
33.11.4	TSimpleIPCClient.Create	758
33.11.5	TSimpleIPCClient.Destroy	758
33.11.6	TSimpleIPCClient.Connect	759
33.11.7	TSimpleIPCClient.Disconnect	759
33.11.8	TSimpleIPCClient.ServerRunning	759
33.11.9	TSimpleIPCClient.SendMessage	760
33.11.10	TSimpleIPCClient.SendStringMessage	760
33.11.11	TSimpleIPCClient.SendStringMessageFmt	760
33.11.12	TSimpleIPCClient.ServerInstance	760
33.12	TSimpleIPCServer	761
33.12.1	Description	761
33.12.2	Method overview	761
33.12.3	Property overview	761
33.12.4	TSimpleIPCServer.Create	762
33.12.5	TSimpleIPCServer.Destroy	762
33.12.6	TSimpleIPCServer.StartServer	762
33.12.7	TSimpleIPCServer.StopServer	763
33.12.8	TSimpleIPCServer.PeekMessage	763
33.12.9	TSimpleIPCServer.ReadMessage	763
33.12.10	TSimpleIPCServer.GetMessageData	763
33.12.11	TSimpleIPCServer.StringMessage	764
33.12.12	TSimpleIPCServer.Message	764
33.12.13	TSimpleIPCServer.MsgType	764
33.12.14	TSimpleIPCServer.MsgData	764
33.12.15	TSimpleIPCServer.InstanceID	765

33.12.16	SimpleIPCServer.ThreadExecuting	765
33.12.17	SimpleIPCServer.ThreadError	765
33.12.18	SimpleIPCServer.Global	765
33.12.19	SimpleIPCServer.OnMessage	766
33.12.20	SimpleIPCServer.OnMessageQueued	766
33.12.21	SimpleIPCServer.OnMessageError	766
33.12.22	SimpleIPCServer.OnThreadError	766
33.12.23	SimpleIPCServer.MaxQueue	767
33.12.24	SimpleIPCServer.MaxAction	767
33.12.25	SimpleIPCServer.Threaded	767
33.12.26	SimpleIPCServer.ThreadTimeout	768
33.12.27	SimpleIPCServer.SynchronizeEvents	768
<b>34</b>	<b>Reference for unit 'SQLDB'</b>	<b>769</b>
34.1	Used units	769
34.2	Overview	769
34.3	Using SQLDB to access databases	770
34.4	Using the universal TSQLConnector type	772
34.5	Retrieving Schema Information	773
34.6	Automatic generation of update SQL statements	773
34.7	Using parameters	774
34.8	Constants, types and variables	775
34.8.1	Constants	775
34.8.2	Types	778
34.8.3	Variables	782
34.9	Procedures and functions	782
34.9.1	GetConnectionDef	782
34.9.2	GetConnectionList	783
34.9.3	RegisterConnection	783
34.9.4	UnRegisterConnection	783
34.10	TSQLStatementInfo	784
34.11	ESQLDatabaseError	784
34.11.1	Description	784
34.11.2	Method overview	784
34.11.3	ESQLDatabaseError.CreateFmt	784
34.12	TConnectionDef	784
34.12.1	Description	784
34.12.2	Method overview	785
34.12.3	TConnectionDef.TypeName	785
34.12.4	TConnectionDef.ConnectionClass	785

34.12.5 TConnectionDef.Description . . . . .	786
34.12.6 TConnectionDef.DefaultLibraryName . . . . .	786
34.12.7 TConnectionDef.LoadFunction . . . . .	786
34.12.8 TConnectionDef.UnLoadFunction . . . . .	786
34.12.9 TConnectionDef.LoadedLibraryName . . . . .	787
34.12.10 TConnectionDef.ApplyParams . . . . .	787
34.13 TCustomSQLQuery . . . . .	787
34.13.1 Description . . . . .	787
34.13.2 Method overview . . . . .	788
34.13.3 Property overview . . . . .	788
34.13.4 TCustomSQLQuery.Create . . . . .	788
34.13.5 TCustomSQLQuery.Destroy . . . . .	788
34.13.6 TCustomSQLQuery.Prepare . . . . .	788
34.13.7 TCustomSQLQuery.UnPrepare . . . . .	789
34.13.8 TCustomSQLQuery.ExecSQL . . . . .	789
34.13.9 TCustomSQLQuery.SetSchemaInfo . . . . .	790
34.13.10 TCustomSQLQuery.RowsAffected . . . . .	790
34.13.11 TCustomSQLQuery.ParamByName . . . . .	790
34.13.12 TCustomSQLQuery.ApplyUpdates . . . . .	791
34.13.13 TCustomSQLQuery.Post . . . . .	791
34.13.14 TCustomSQLQuery.Delete . . . . .	791
34.13.15 TCustomSQLQuery.Prepared . . . . .	791
34.13.16 TCustomSQLQuery.SQLConnection . . . . .	792
34.13.17 TCustomSQLQuery.SQLTransaction . . . . .	792
34.14 TCustomSQLStatement . . . . .	792
34.14.1 Description . . . . .	792
34.14.2 Method overview . . . . .	793
34.14.3 Property overview . . . . .	793
34.14.4 TCustomSQLStatement.Create . . . . .	793
34.14.5 TCustomSQLStatement.Destroy . . . . .	793
34.14.6 TCustomSQLStatement.Prepare . . . . .	793
34.14.7 TCustomSQLStatement.Execute . . . . .	794
34.14.8 TCustomSQLStatement.Unprepare . . . . .	794
34.14.9 TCustomSQLStatement.ParamByName . . . . .	794
34.14.10 TCustomSQLStatement.RowsAffected . . . . .	795
34.14.11 TCustomSQLStatement.Prepared . . . . .	795
34.15 TServerIndexDefs . . . . .	795
34.15.1 Description . . . . .	795
34.15.2 Method overview . . . . .	795
34.15.3 TServerIndexDefs.Create . . . . .	795

34.15.4 TServerIndexDefs.Update . . . . .	796
34.16 TSQLConnection . . . . .	796
34.16.1 Description . . . . .	796
34.16.2 Method overview . . . . .	796
34.16.3 Property overview . . . . .	797
34.16.4 TSQLConnection.Create . . . . .	797
34.16.5 TSQLConnection.Destroy . . . . .	797
34.16.6 TSQLConnection.StartTransaction . . . . .	798
34.16.7 TSQLConnection.EndTransaction . . . . .	798
34.16.8 TSQLConnection.ExecuteDirect . . . . .	798
34.16.9 TSQLConnection.GetObjectNames . . . . .	799
34.16.10 TSQLConnection.GetTableNames . . . . .	799
34.16.11 TSQLConnection.GetProcedureNames . . . . .	799
34.16.12 TSQLConnection.GetFieldNames . . . . .	799
34.16.13 TSQLConnection.GetSchemaNames . . . . .	800
34.16.14 TSQLConnection.GetSequenceNames . . . . .	800
34.16.15 TSQLConnection.GetConnectionInfo . . . . .	800
34.16.16 TSQLConnection.GetStatementInfo . . . . .	800
34.16.17 TSQLConnection.CreateDB . . . . .	801
34.16.18 TSQLConnection.DropDB . . . . .	801
34.16.19 TSQLConnection.GetNextValue . . . . .	801
34.16.20 TSQLConnection.ConnOptions . . . . .	802
34.16.21 TSQLConnection.Handle . . . . .	802
34.16.22 TSQLConnection.FieldNameQuoteChars . . . . .	802
34.16.23 TSQLConnection.Password . . . . .	802
34.16.24 TSQLConnection.Transaction . . . . .	803
34.16.25 TSQLConnection.UserName . . . . .	803
34.16.26 TSQLConnection.CharSet . . . . .	803
34.16.27 TSQLConnection.HostName . . . . .	804
34.16.28 TSQLConnection.OnLog . . . . .	804
34.16.29 TSQLConnection.LogEvents . . . . .	804
34.16.30 TSQLConnection.Options . . . . .	805
34.16.31 TSQLConnection.Role . . . . .	805
34.16.32 TSQLConnection.Connected . . . . .	805
34.16.33 TSQLConnection.DatabaseName . . . . .	806
34.16.34 TSQLConnection.KeepConnection . . . . .	806
34.16.35 TSQLConnection.LoginPrompt . . . . .	806
34.16.36 TSQLConnection.Params . . . . .	807
34.16.37 TSQLConnection.OnLogin . . . . .	807
34.17 TSQLConnector . . . . .	807



34.17.1 Description . . . . .	807
34.17.2 Property overview . . . . .	807
34.17.3 TSQLConnector.ConnectorType . . . . .	808
34.18 TSQLCursor . . . . .	808
34.18.1 Description . . . . .	808
34.19 TSQLDBFieldDef . . . . .	808
34.19.1 Description . . . . .	808
34.19.2 Property overview . . . . .	808
34.19.3 TSQLDBFieldDef.SQLDBData . . . . .	808
34.20 TSQLDBFieldDefs . . . . .	809
34.20.1 Description . . . . .	809
34.21 TSQLDBParam . . . . .	809
34.21.1 Description . . . . .	809
34.21.2 Property overview . . . . .	809
34.21.3 TSQLDBParam.FieldDef . . . . .	809
34.21.4 TSQLDBParam.SQLDBData . . . . .	809
34.22 TSQLDBParams . . . . .	810
34.22.1 Description . . . . .	810
34.23 TSQLHandle . . . . .	810
34.23.1 Description . . . . .	810
34.24 TSQLQuery . . . . .	810
34.24.1 Description . . . . .	810
34.24.2 Property overview . . . . .	812
34.24.3 TSQLQuery.SchemaType . . . . .	813
34.24.4 TSQLQuery.StatementType . . . . .	813
34.24.5 TSQLQuery.MaxIndexesCount . . . . .	813
34.24.6 TSQLQuery.FieldDefs . . . . .	813
34.24.7 TSQLQuery.Active . . . . .	814
34.24.8 TSQLQuery.AutoCalcFields . . . . .	814
34.24.9 TSQLQuery.Filter . . . . .	814
34.24.10 TSQLQuery.Filtered . . . . .	814
34.24.11 TSQLQuery.AfterCancel . . . . .	814
34.24.12 TSQLQuery.AfterClose . . . . .	814
34.24.13 TSQLQuery.AfterDelete . . . . .	814
34.24.14 TSQLQuery.AfterEdit . . . . .	815
34.24.15 TSQLQuery.AfterInsert . . . . .	815
34.24.16 TSQLQuery.AfterOpen . . . . .	815
34.24.17 TSQLQuery.AfterPost . . . . .	815
34.24.18 TSQLQuery.AfterRefresh . . . . .	815
34.24.19 TSQLQuery.AfterScroll . . . . .	815

34.24.20	TSQLQuery.BeforeCancel	815
34.24.21	TSQLQuery.BeforeClose	816
34.24.22	TSQLQuery.BeforeDelete	816
34.24.23	TSQLQuery.BeforeEdit	816
34.24.24	TSQLQuery.BeforeInsert	816
34.24.25	TSQLQuery.BeforeOpen	816
34.24.26	TSQLQuery.BeforePost	816
34.24.27	TSQLQuery.BeforeRefresh	816
34.24.28	TSQLQuery.BeforeScroll	817
34.24.29	TSQLQuery.OnCalcFields	817
34.24.30	TSQLQuery.OnDeleteError	817
34.24.31	TSQLQuery.OnEditError	817
34.24.32	TSQLQuery.OnFilterRecord	817
34.24.33	TSQLQuery.OnNewRecord	817
34.24.34	TSQLQuery.OnPostError	817
34.24.35	TSQLQuery.Database	818
34.24.36	TSQLQuery.Transaction	818
34.24.37	TSQLQuery.ReadOnly	818
34.24.38	TSQLQuery.SQL	818
34.24.39	TSQLQuery.InsertSQL	819
34.24.40	TSQLQuery.UpdateSQL	819
34.24.41	TSQLQuery.DeleteSQL	820
34.24.42	TSQLQuery.RefreshSQL	820
34.24.43	TSQLQuery.IndexDefs	820
34.24.44	TSQLQuery.Options	821
34.24.45	TSQLQuery.Params	821
34.24.46	TSQLQuery.ParamCheck	822
34.24.47	TSQLQuery.ParseSQL	822
34.24.48	TSQLQuery.UpdateMode	822
34.24.49	TSQLQuery.UsePrimaryKeyAsKey	823
34.24.50	TSQLQuery.DataSource	823
34.24.51	TSQLQuery.Sequence	823
34.24.52	TSQLQuery.ServerFilter	824
34.24.53	TSQLQuery.ServerFiltered	824
34.24.54	TSQLQuery.ServerIndexDefs	824
34.25	TSQLScript	825
34.25.1	Description	825
34.25.2	Method overview	825
34.25.3	Property overview	825
34.25.4	TSQLScript.Create	825

34.25.5 TSQLScript.Destroy	826
34.25.6 TSQLScript.Execute	826
34.25.7 TSQLScript.ExecuteScript	826
34.25.8 TSQLScript.Aborted	826
34.25.9 TSQLScript.Line	827
34.25.10 TSQLScript.DataBase	827
34.25.11 TSQLScript.Transaction	827
34.25.12 TSQLScript.OnDirective	827
34.25.13 TSQLScript.AutoCommit	828
34.25.14 TSQLScript.UseDollarString	828
34.25.15 TSQLScript.DollarStrings	828
34.25.16 TSQLScript.Directives	829
34.25.17 TSQLScript.Defines	829
34.25.18 TSQLScript.Script	829
34.25.19 TSQLScript.Terminator	830
34.25.20 TSQLScript.CommentsinSQL	830
34.25.21 TSQLScript.UseSetTerm	830
34.25.22 TSQLScript.UseCommit	831
34.25.23 TSQLScript.UseDefines	831
34.25.24 TSQLScript.OnException	832
34.26 TSQLSequence	832
34.26.1 Description	832
34.26.2 Method overview	832
34.26.3 Property overview	832
34.26.4 TSQLSequence.Create	832
34.26.5 TSQLSequence.Assign	833
34.26.6 TSQLSequence.Apply	833
34.26.7 TSQLSequence.GetNextValue	833
34.26.8 TSQLSequence.FieldName	833
34.26.9 TSQLSequence.SequenceName	834
34.26.10 TSQLSequence.IncrementBy	834
34.26.11 TSQLSequence.ApplyEvent	834
34.27 TSQLStatement	834
34.27.1 Description	834
34.27.2 Property overview	835
34.27.3 TSQLStatement.Database	835
34.27.4 TSQLStatement.DataSource	835
34.27.5 TSQLStatement.ParamCheck	835
34.27.6 TSQLStatement.Params	836
34.27.7 TSQLStatement.ParseSQL	836

34.27.8 TSQLStatement.SQL	836
34.27.9 TSQLStatement.Transaction	837
34.28 TSQLTransaction	837
34.28.1 Description	837
34.28.2 Method overview	837
34.28.3 Property overview	837
34.28.4 TSQLTransaction.Create	837
34.28.5 TSQLTransaction.Destroy	838
34.28.6 TSQLTransaction.Commit	838
34.28.7 TSQLTransaction.CommitRetaining	838
34.28.8 TSQLTransaction.Rollback	839
34.28.9 TSQLTransaction.RollbackRetaining	839
34.28.10 TSQLTransaction.StartTransaction	839
34.28.11 TSQLTransaction.EndTransaction	840
34.28.12 TSQLTransaction.Handle	840
34.28.13 TSQLTransaction.SQLConnection	840
34.28.14 TSQLTransaction.Action	840
34.28.15 TSQLTransaction.Database	841
34.28.16 TSQLTransaction.Params	841
34.28.17 TSQLTransaction.Options	841
<b>35 Reference for unit 'SQLTypes'</b>	<b>842</b>
35.1 Used units	842
35.2 Constants, types and variables	842
35.2.1 Types	842
35.3 TSqLObjectIdentifier	844
35.3.1 Description	844
35.3.2 Method overview	844
35.3.3 Property overview	844
35.3.4 TSqLObjectIdentifier.Create	844
35.3.5 TSqLObjectIdentifier.FullName	845
35.3.6 TSqLObjectIdentifier.SchemaName	845
35.3.7 TSqLObjectIdentifier.ObjectName	845
35.4 TSqLObjectIdentifierList	845
35.4.1 Description	845
35.4.2 Method overview	846
35.4.3 Property overview	846
35.4.4 TSqLObjectIdentifierList.AddIdentifier	846
35.4.5 TSqLObjectIdentifierList.Identifiers	846
<b>36 Reference for unit 'streamcoll'</b>	<b>847</b>

36.1	Used units	847
36.2	Overview	847
36.3	Procedures and functions	847
36.3.1	ColReadBoolean	847
36.3.2	ColReadCurrency	848
36.3.3	ColReadDateTime	848
36.3.4	ColReadFloat	848
36.3.5	ColReadInteger	848
36.3.6	ColReadString	849
36.3.7	ColWriteBoolean	849
36.3.8	ColWriteCurrency	849
36.3.9	ColWriteDateTime	849
36.3.10	ColWriteFloat	850
36.3.11	ColWriteInteger	850
36.3.12	ColWriteString	850
36.4	EStreamColl	850
36.4.1	Description	850
36.5	TStreamCollection	850
36.5.1	Description	850
36.5.2	Method overview	851
36.5.3	Property overview	851
36.5.4	TStreamCollection.LoadFromStream	851
36.5.5	TStreamCollection.SaveToStream	851
36.5.6	TStreamCollection.Streaming	851
36.6	TStreamCollectionItem	852
36.6.1	Description	852
<b>37</b>	<b>Reference for unit 'streamex'</b>	<b>853</b>
37.1	Used units	853
37.2	Overview	853
37.3	Constants, types and variables	853
37.3.1	Constants	853
37.4	TBidirBinaryObjectReader	854
37.4.1	Description	854
37.4.2	Property overview	854
37.4.3	TBidirBinaryObjectReader.Position	854
37.5	TBidirBinaryObjectWriter	854
37.5.1	Description	854
37.5.2	Property overview	854
37.5.3	TBidirBinaryObjectWriter.Position	854

37.6	TDelphiReader	855
37.6.1	Description	855
37.6.2	Method overview	855
37.6.3	Property overview	855
37.6.4	TDelphiReader.GetDriver	855
37.6.5	TDelphiReader.ReadStr	855
37.6.6	TDelphiReader.Read	855
37.6.7	TDelphiReader.Position	856
37.7	TDelphiWriter	856
37.7.1	Description	856
37.7.2	Method overview	856
37.7.3	Property overview	856
37.7.4	TDelphiWriter.GetDriver	856
37.7.5	TDelphiWriter.FlushBuffer	857
37.7.6	TDelphiWriter.Write	857
37.7.7	TDelphiWriter.WriteStr	857
37.7.8	TDelphiWriter.WriteValue	857
37.7.9	TDelphiWriter.Position	857
37.8	TFileReader	858
37.8.1	Description	858
37.8.2	Method overview	858
37.8.3	TFileReader.Create	858
37.8.4	TFileReader.Destroy	858
37.8.5	TFileReader.Reset	859
37.8.6	TFileReader.Close	859
37.8.7	TFileReader.ReadLine	859
37.9	TStreamHelper	859
37.9.1	Description	859
37.9.2	Method overview	860
37.9.3	TStreamHelper.ReadWordLE	860
37.9.4	TStreamHelper.ReadDWordLE	860
37.9.5	TStreamHelper.ReadQWordLE	861
37.9.6	TStreamHelper.WriteWordLE	861
37.9.7	TStreamHelper.WriteDWordLE	861
37.9.8	TStreamHelper.WriteQWordLE	861
37.9.9	TStreamHelper.ReadWordBE	862
37.9.10	TStreamHelper.ReadDWordBE	862
37.9.11	TStreamHelper.ReadQWordBE	862
37.9.12	TStreamHelper.WriteWordBE	862
37.9.13	TStreamHelper.WriteDWordBE	863

37.9.14 TStreamHelper.WriteQWordBE . . . . .	863
37.9.15 TStreamHelper.ReadSingle . . . . .	863
37.9.16 TStreamHelper.ReadDouble . . . . .	863
37.9.17 TStreamHelper.WriteSingle . . . . .	864
37.9.18 TStreamHelper.WriteDouble . . . . .	864
37.9.19 TStreamHelper.ReadByte . . . . .	864
37.9.20 TStreamHelper.ReadWord . . . . .	864
37.9.21 TStreamHelper.ReadDWord . . . . .	864
37.9.22 TStreamHelper.ReadQWord . . . . .	864
37.9.23 TStreamHelper.WriteByte . . . . .	865
37.9.24 TStreamHelper.WriteWord . . . . .	865
37.9.25 TStreamHelper.WriteDWord . . . . .	865
37.9.26 TStreamHelper.WriteQWord . . . . .	865
37.10 TStreamReader . . . . .	865
37.10.1 Description . . . . .	865
37.10.2 Method overview . . . . .	865
37.10.3 Property overview . . . . .	865
37.10.4 TStreamReader.Create . . . . .	865
37.10.5 TStreamReader.Destroy . . . . .	866
37.10.6 TStreamReader.Reset . . . . .	866
37.10.7 TStreamReader.Close . . . . .	866
37.10.8 TStreamReader.ReadLine . . . . .	867
37.10.9 TStreamReader.BaseStream . . . . .	867
37.10.10 TStreamReader.OwnsStream . . . . .	867
37.11 TStringReader . . . . .	867
37.11.1 Description . . . . .	867
37.11.2 Method overview . . . . .	868
37.11.3 TStringReader.Create . . . . .	868
37.11.4 TStringReader.Destroy . . . . .	868
37.11.5 TStringReader.Reset . . . . .	868
37.11.6 TStringReader.Close . . . . .	869
37.11.7 TStringReader.ReadLine . . . . .	869
37.12 TTextReader . . . . .	869
37.12.1 Description . . . . .	869
37.12.2 Method overview . . . . .	869
37.12.3 Property overview . . . . .	869
37.12.4 TTextReader.Create . . . . .	870
37.12.5 TTextReader.Reset . . . . .	870
37.12.6 TTextReader.Close . . . . .	870
37.12.7 TTextReader.ReadLine . . . . .	870

37.12.8 TTextReader.Eof . . . . .	871
37.13 TWindowedStream . . . . .	871
37.13.1 Description . . . . .	871
37.13.2 Method overview . . . . .	871
37.13.3 TWindowedStream.Create . . . . .	871
37.13.4 TWindowedStream.Destroy . . . . .	872
37.13.5 TWindowedStream.Read . . . . .	872
37.13.6 TWindowedStream.Write . . . . .	872
37.13.7 TWindowedStream.Seek . . . . .	872
<b>38 Reference for unit 'StreamIO' . . . . .</b>	<b>874</b>
38.1 Used units . . . . .	874
38.2 Overview . . . . .	874
38.3 Procedures and functions . . . . .	874
38.3.1 AssignStream . . . . .	874
38.3.2 GetStream . . . . .	875
<b>39 Reference for unit 'syncobjs' . . . . .</b>	<b>876</b>
39.1 Used units . . . . .	876
39.2 Overview . . . . .	876
39.3 Constants, types and variables . . . . .	876
39.3.1 Constants . . . . .	876
39.3.2 Types . . . . .	876
39.4 ELockException . . . . .	877
39.4.1 Description . . . . .	877
39.5 ELockRecursionException . . . . .	877
39.5.1 Description . . . . .	877
39.6 ESynObjectException . . . . .	877
39.6.1 Description . . . . .	877
39.7 TCriticalSection . . . . .	877
39.7.1 Description . . . . .	877
39.7.2 Method overview . . . . .	878
39.7.3 TCriticalSection.Acquire . . . . .	878
39.7.4 TCriticalSection.Release . . . . .	878
39.7.5 TCriticalSection.Enter . . . . .	879
39.7.6 TCriticalSection.TryEnter . . . . .	879
39.7.7 TCriticalSection.Leave . . . . .	879
39.7.8 TCriticalSection.Create . . . . .	879
39.7.9 TCriticalSection.Destroy . . . . .	880
39.8 TEventObject . . . . .	880
39.8.1 Description . . . . .	880



39.8.2	Method overview	880
39.8.3	Property overview	880
39.8.4	TEventObject.Create	880
39.8.5	TEventObject.destroy	881
39.8.6	TEventObject.ResetEvent	881
39.8.7	TEventObject.SetEvent	881
39.8.8	TEventObject.WaitFor	881
39.8.9	TEventObject.ManualReset	882
39.9	THandleObject	882
39.9.1	Description	882
39.9.2	Method overview	882
39.9.3	Property overview	882
39.9.4	THandleObject.destroy	882
39.9.5	THandleObject.Handle	883
39.9.6	THandleObject.LastError	883
39.10	TSimpleEvent	883
39.10.1	Description	883
39.10.2	Method overview	883
39.10.3	TSimpleEvent.Create	883
39.11	TSynchroObject	884
39.11.1	Description	884
39.11.2	Method overview	884
39.11.3	TSynchroObject.Acquire	884
39.11.4	TSynchroObject.Release	884
<b>40</b>	<b>Reference for unit 'URIParser'</b>	<b>885</b>
40.1	Overview	885
40.2	Constants, types and variables	885
40.2.1	Types	885
40.3	Procedures and functions	885
40.3.1	EncodeURI	885
40.3.2	FilenameToURI	885
40.3.3	IsAbsoluteURI	886
40.3.4	ParseURI	886
40.3.5	ResolveRelativeURI	886
40.3.6	URIToFilename	887
40.4	TURI	887
<b>41</b>	<b>Reference for unit 'Zipper'</b>	<b>888</b>
41.1	Used units	888
41.2	Overview	888

41.3	Constants, types and variables	888
41.3.1	Constants	888
41.3.2	Types	891
41.4	Central_File_Header_Type	892
41.5	CodeRec	893
41.6	End_of_Central_Dir_Type	893
41.7	Extensible_Data_Field_Header_Type	893
41.8	Local_File_Header_Type	893
41.9	Zip64_End_of_Central_Dir_Locator_type	894
41.10	Zip64_End_of_Central_Dir_type	894
41.11	Zip64_Extended_Info_Field_Type	894
41.12	EZipError	895
41.12.1	Description	895
41.13	TCompressor	895
41.13.1	Description	895
41.13.2	Method overview	895
41.13.3	Property overview	895
41.13.4	TCompressor.Create	895
41.13.5	TCompressor.Compress	895
41.13.6	TCompressor.ZipID	896
41.13.7	TCompressor.ZipVersionReqd	896
41.13.8	TCompressor.ZipBitFlag	896
41.13.9	TCompressor.Terminate	896
41.13.10	TCompressor.BufferSize	896
41.13.11	TCompressor.OnPercent	896
41.13.12	TCompressor.OnProgress	896
41.13.13	TCompressor.Crc32Val	897
41.13.14	TCompressor.Terminated	897
41.14	TDeCompressor	897
41.14.1	Description	897
41.14.2	Method overview	897
41.14.3	Property overview	897
41.14.4	TDeCompressor.Create	897
41.14.5	TDeCompressor.DeCompress	898
41.14.6	TDeCompressor.Terminate	898
41.14.7	TDeCompressor.ZipID	898
41.14.8	TDeCompressor.BufferSize	898
41.14.9	TDeCompressor.OnPercent	898
41.14.10	TDeCompressor.OnProgress	898
41.14.11	TDeCompressor.OnProgressEx	898

41.14.12 TDeCompressor.Crc32Val	899
41.14.13 TDeCompressor.Terminated	899
41.15 TDeflater	899
41.15.1 Description	899
41.15.2 Method overview	899
41.15.3 Property overview	899
41.15.4 TDeflater.Create	899
41.15.5 TDeflater.Compress	899
41.15.6 TDeflater.ZipID	900
41.15.7 TDeflater.ZipVersionReqd	900
41.15.8 TDeflater.ZipBitFlag	900
41.15.9 TDeflater.CompressionLevel	900
41.16 TFullZipFileEntries	900
41.16.1 Property overview	900
41.16.2 TFullZipFileEntries.FullEntries	900
41.17 TFullZipFileEntry	900
41.17.1 Property overview	900
41.17.2 TFullZipFileEntry.BitFlags	901
41.17.3 TFullZipFileEntry.CompressMethod	901
41.17.4 TFullZipFileEntry.CompressedSize	901
41.17.5 TFullZipFileEntry.CRC32	901
41.18 TInflater	901
41.18.1 Description	901
41.18.2 Method overview	901
41.18.3 TInflater.Create	901
41.18.4 TInflater.DeCompress	902
41.18.5 TInflater.ZipID	902
41.19 TShrinker	902
41.19.1 Description	902
41.19.2 Method overview	902
41.19.3 TShrinker.Create	902
41.19.4 TShrinker.Destroy	902
41.19.5 TShrinker.Compress	902
41.19.6 TShrinker.ZipID	902
41.19.7 TShrinker.ZipVersionReqd	903
41.19.8 TShrinker.ZipBitFlag	903
41.20 TUnZipper	903
41.20.1 Method overview	903
41.20.2 Property overview	903
41.20.3 TUnZipper.Create	904

41.20.4 TUnZipper.Destroy	904
41.20.5 TUnZipper.UnZipAllFiles	904
41.20.6 TUnZipper.UnZipFile	904
41.20.7 TUnZipper.UnZipFiles	904
41.20.8 TUnZipper.Unzip	905
41.20.9 TUnZipper.Clear	905
41.20.10 TUnZipper.Examine	905
41.20.11 TUnZipper.Terminate	905
41.20.12 TUnZipper.BufferSize	905
41.20.13 TUnZipper.OnOpenInputStream	905
41.20.14 TUnZipper.OnCloseInputStream	906
41.20.15 TUnZipper.OnCreateStream	906
41.20.16 TUnZipper.OnDoneStream	906
41.20.17 TUnZipper.OnPercent	906
41.20.18 TUnZipper.OnProgress	906
41.20.19 TUnZipper.OnProgressEx	906
41.20.20 TUnZipper.OnStartFile	906
41.20.21 TUnZipper.OnEndFile	907
41.20.22 TUnZipper.FileName	907
41.20.23 TUnZipper.OutputPath	907
41.20.24 TUnZipper.FileComment	907
41.20.25 TUnZipper.Files	907
41.20.26 TUnZipper.Entries	907
41.20.27 TUnZipper.UseUTF8	908
41.20.28 TUnZipper.Flat	908
41.20.29 TUnZipper.Terminated	908
41.21 TZipFileEntries	908
41.21.1 Description	908
41.21.2 Method overview	908
41.21.3 Property overview	908
41.21.4 TZipFileEntries.AddFileEntry	908
41.21.5 TZipFileEntries.AddFileEntries	909
41.21.6 TZipFileEntries.Entries	909
41.22 TZipFileEntry	909
41.22.1 Method overview	909
41.22.2 Property overview	909
41.22.3 TZipFileEntry.Create	909
41.22.4 TZipFileEntry.IsDirectory	909
41.22.5 TZipFileEntry.IsLink	910
41.22.6 TZipFileEntry.Assign	910

41.22.7 TZipFileEntry.Stream . . . . .	910
41.22.8 TZipFileEntry.ArchiveFileName . . . . .	910
41.22.9 TZipFileEntry.UTF8ArchiveFileName . . . . .	910
41.22.10 TZipFileEntry.DiskFileName . . . . .	910
41.22.11 TZipFileEntry.UTF8DiskFileName . . . . .	910
41.22.12 TZipFileEntry.Size . . . . .	911
41.22.13 TZipFileEntry.DateTime . . . . .	911
41.22.14 TZipFileEntry.OS . . . . .	911
41.22.15 TZipFileEntry.Attributes . . . . .	911
41.22.16 TZipFileEntry.CompressionLevel . . . . .	911
41.23 TZipper . . . . .	912
41.23.1 Method overview . . . . .	912
41.23.2 Property overview . . . . .	912
41.23.3 TZipper.Create . . . . .	912
41.23.4 TZipper.Destroy . . . . .	912
41.23.5 TZipper.ZipAllFiles . . . . .	912
41.23.6 TZipper.SaveToFile . . . . .	913
41.23.7 TZipper.SaveToStream . . . . .	913
41.23.8 TZipper.ZipFile . . . . .	913
41.23.9 TZipper.ZipFiles . . . . .	913
41.23.10 TZipper.Zip . . . . .	913
41.23.11 TZipper.Clear . . . . .	914
41.23.12 TZipper.Terminate . . . . .	914
41.23.13 TZipper.BufferSize . . . . .	914
41.23.14 TZipper.OnPercent . . . . .	914
41.23.15 TZipper.OnProgress . . . . .	914
41.23.16 TZipper.OnStartFile . . . . .	914
41.23.17 TZipper.OnEndFile . . . . .	915
41.23.18 TZipper.FileName . . . . .	915
41.23.19 TZipper.FileComment . . . . .	915
41.23.20 TZipper.Files . . . . .	915
41.23.21 TZipper.InMemSize . . . . .	915
41.23.22 TZipper.Entries . . . . .	915
41.23.23 TZipper.Terminated . . . . .	915
41.23.24 TZipper.UseLanguageEncoding . . . . .	916
<b>42 Reference for unit 'ZStream'</b>	<b>917</b>
42.1 Used units . . . . .	917
42.2 Overview . . . . .	917
42.3 Constants, types and variables . . . . .	917

42.3.1	Types	917
42.4	Ecompressionerror	918
42.4.1	Description	918
42.5	Edecompressionerror	918
42.5.1	Description	918
42.6	Egzfileerror	918
42.6.1	Description	918
42.7	Ezliberror	918
42.7.1	Description	918
42.8	Tcompressionstream	918
42.8.1	Description	918
42.8.2	Method overview	919
42.8.3	Property overview	919
42.8.4	Tcompressionstream.create	919
42.8.5	Tcompressionstream.destroy	919
42.8.6	Tcompressionstream.write	919
42.8.7	Tcompressionstream.flush	920
42.8.8	Tcompressionstream.get_compressionrate	920
42.8.9	Tcompressionstream.OnProgress	920
42.9	Tcustomzlibstream	920
42.9.1	Description	920
42.9.2	Method overview	921
42.9.3	Tcustomzlibstream.create	921
42.9.4	Tcustomzlibstream.destroy	921
42.10	Tdecompressionstream	921
42.10.1	Description	921
42.10.2	Method overview	921
42.10.3	Property overview	922
42.10.4	Tdecompressionstream.create	922
42.10.5	Tdecompressionstream.destroy	922
42.10.6	Tdecompressionstream.read	922
42.10.7	Tdecompressionstream.Seek	923
42.10.8	Tdecompressionstream.get_compressionrate	923
42.10.9	Tdecompressionstream.OnProgress	923
42.11	TGZFileStream	924
42.11.1	Description	924
42.11.2	Method overview	924
42.11.3	TGZFileStream.create	924
42.11.4	TGZFileStream.read	924
42.11.5	TGZFileStream.write	925

<a href="#">42.11.6 TGZFileStream.seek</a> . . . . .	925
<a href="#">42.11.7 TGZFileStream.destroy</a> . . . . .	925

## 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 'ascii85'

### 1.1 Used units

Table 1.1: Used units by unit 'ascii85'

Name	Page
Classes	??
sysutils	??

### 1.2 Overview

The `ascii85` provides an ASCII 85 or base 85 decoding algorithm. It is class and stream based: the `TASCII85DecoderStream` ([89](#)) stream can be used to decode any stream with ASCII85 encoded data.

Currently, no ASCII85 encoder stream is available.

It's usage and purpose is similar to the IDEA ([608](#)) or base64 ([117](#)) units.

### 1.3 Constants, types and variables

#### 1.3.1 Types

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

Table 1.2: Enumeration values for type TASCII85State

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

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

## 1.4 TASCII85DecoderStream

### 1.4.1 Description

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

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

### 1.4.2 Method overview

Page	Method	Description
<a href="#">90</a>	Close	Close decoder
<a href="#">90</a>	ClosedP	Check if the state is correct
<a href="#">89</a>	Create	Create new ASCII 85 decoder stream
<a href="#">90</a>	Decode	Decode source byte
<a href="#">90</a>	Destroy	Clean up instance
<a href="#">91</a>	Read	Read data from stream
<a href="#">91</a>	Seek	Set stream position

### 1.4.3 Property overview

Page	Properties	Access	Description
<a href="#">91</a>	BExpectBoundary	rw	Expect character

### 1.4.4 TASCII85DecoderStream.Create

Synopsis: Create new ASCII 85 decoder stream

Declaration: constructor Create(aStream: TStream)

Visibility: published

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

See also: TASCII85DecoderStream.Destroy ([90](#))

### 1.4.5 TASCII85DecoderStream.Decode

Synopsis: Decode source byte

Declaration: `procedure Decode(aInput: Byte)`

Visibility: published

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

See also: `TASCII85DecoderStream.Close` ([90](#))

### 1.4.6 TASCII85DecoderStream.Close

Synopsis: Close decoder

Declaration: `procedure Close`

Visibility: published

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

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

See also: `TASCII85DecoderStream.ClosedP` ([90](#)), `TASCII85DecoderStream.Read` ([91](#)), `TASCII85DecoderStream.Destroy` ([90](#))

### 1.4.7 TASCII85DecoderStream.ClosedP

Synopsis: Check if the state is correct

Declaration: `function ClosedP : Boolean`

Visibility: published

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

See also: `TASCII85DecoderStream.Close` ([90](#)), `TASCII85DecoderStream.BExpectBoundary` ([91](#))

### 1.4.8 TASCII85DecoderStream.Destroy

Synopsis: Clean up instance

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` closes the input stream using `Close` ([90](#)) and cleans up the `TASCII85DecoderStream` instance from memory.

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

See also: `TASCII85DecoderStream.Close` ([90](#))

### 1.4.9 TASCII85DecoderStream.Read

Synopsis: Read data from stream

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

Visibility: public

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

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

### 1.4.10 TASCII85DecoderStream.Seek

Synopsis: Set stream position

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

Visibility: public

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

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

See also: `TASCII85DecoderStream.Read` ([91](#))

### 1.4.11 TASCII85DecoderStream.BExpectBoundary

Synopsis: Expect character

Declaration: `Property BExpectBoundary : Boolean`

Visibility: published

Access: Read,Write

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

See also: `ClosedP` ([90](#))

## 1.5 TASCII85EncoderStream

### 1.5.1 Description

`TASCII85EncoderStream` is the counterpart to the `TASCII85DecoderStream` ([89](#)) decoder stream: what `TASCII85EncoderStream` encodes, can be decoded by `TASCII85DecoderStream` ([89](#)).

The encoder stream works using a destination stream: whatever data is written to the encoder stream is encoded and written to the destination stream. The stream must be passed on in the constructor.

Note that all encoded data is only written to the destination stream when the encoder stream is destroyed.

See also: `TASCII85EncoderStream.create` ([92](#)), `TASCII85DecoderStream` ([89](#))

### 1.5.2 Method overview

Page	Method	Description
<a href="#">92</a>	Create	Create a new instance of <code>TASCII85EncoderStream</code>
<a href="#">92</a>	Destroy	Flushed the data to the output stream and cleans up the encoder instance.
<a href="#">92</a>	Write	Write data encoded to the destination stream

### 1.5.3 Property overview

Page	Properties	Access	Description
<a href="#">93</a>	Boundary	r	Is a boundary delineator written before and after the data
<a href="#">93</a>	Width	r	Width of the lines written to the data stream

### 1.5.4 TASCII85EncoderStream.Create

Synopsis: Create a new instance of `TASCII85EncoderStream`

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

Visibility: `public`

Description: `Create` creates a new instance of `TASCII85EncoderStream`. It stores `ADest` as the destination stream for the encoded data. The `Width` parameter indicates the width of the lines that are written by the encoder: after this amount of characters, a linefeed is put in the data stream. If `ABoundary` is `True` then a boundary delineator is written to the stream before and after the data.

See also: `TASCII85EncoderStream` ([91](#)), `Width` ([93](#)), `Boundary` ([93](#))

### 1.5.5 TASCII85EncoderStream.Destroy

Synopsis: Flushed the data to the output stream and cleans up the encoder instance.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` writes the data remaining in the internal buffer to the destination stream (possibly followed by a boundary delineator) and then destroys the encoder instance.

See also: `TASCII85EncoderStream.Write` ([92](#)), `TASCII85EncoderStream.Boundary` ([93](#))

### 1.5.6 TASCII85EncoderStream.Write

Synopsis: Write data encoded to the destination stream

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

Visibility: `public`

Description: `Write` encodes the `aCount` bytes of data in `aBuffer` and writes the encoded data to the destination stream.

Not all data is written immediately to the destination stream. Only after the encoding stream is destroyed will the destination stream contain the full data.

See also: `TASCII85EncoderStream.Destroy` ([92](#))

### 1.5.7 TASCII85EncoderStream.Width

Synopsis: Width of the lines written to the data stream

Declaration: `Property Width : Integer`

Visibility: `public`

Access: `Read`

Description: `Width` is the width of the lines of encoded data written to the stream. After `Width` lines, a line ending will be written to the stream. The value is passed to the constructor and cannot be changed afterwards.

See also: [Boundary \(93\)](#), [Create \(92\)](#)

### 1.5.8 TASCII85EncoderStream.Boundary

Synopsis: Is a boundary delineator written before and after the data

Declaration: `Property Boundary : Boolean`

Visibility: `public`

Access: `Read`

Description: `Boundary` indicates whether the stream will write a boundary delineator before and after the encoded data. It is passed to the constructor and cannot be changed.

See also: [Width \(93\)](#), [Create \(92\)](#)

## 1.6 TASCII85RingBuffer

### 1.6.1 Description

`TASCII85RingBuffer` is an internal buffer class: it maintains a memory buffer of 1Kb, for faster reading of the stream. It should not be necessary to instantiate an instance of this class, the `TASCII85DecoderStream` ([89](#)) decoder stream will create an instance of this class automatically.

See also: `TASCII85DecoderStream` ([89](#))

### 1.6.2 Method overview

Page	Method	Description
<a href="#">94</a>	<code>Read</code>	Read data from the internal buffer
<a href="#">94</a>	<code>Write</code>	Write data to the internal buffer

### 1.6.3 Property overview

Page	Properties	Access	Description
<a href="#">94</a>	<code>FillCount</code>	<code>r</code>	Number of bytes in buffer
<a href="#">94</a>	<code>Size</code>	<code>r</code>	Size of buffer

### 1.6.4 TASCII85RingBuffer.Write

Synopsis: Write data to the internal buffer

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

Visibility: published

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

See also: `TASCII85RingBuffer.FillCount` ([94](#)), `TASCII85RingBuffer.Read` ([94](#)), `TASCII85RingBuffer.Size` ([94](#))

### 1.6.5 TASCII85RingBuffer.Read

Synopsis: Read data from the internal buffer

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

Visibility: published

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

See also: `TASCII85RingBuffer.FillCount` ([94](#)), `TASCII85RingBuffer.Write` ([94](#)), `TASCII85RingBuffer.Size` ([94](#))

### 1.6.6 TASCII85RingBuffer.FillCount

Synopsis: Number of bytes in buffer

Declaration: `Property FillCount : Cardinal`

Visibility: published

Access: Read

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

See also: `TASCII85RingBuffer.Write` ([94](#)), `TASCII85RingBuffer.Read` ([94](#)), `TASCII85RingBuffer.Size` ([94](#))

### 1.6.7 TASCII85RingBuffer.Size

Synopsis: Size of buffer

Declaration: `Property Size : Cardinal`

Visibility: published

Access: Read

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

See also: `TASCII85RingBuffer.FillCount` ([94](#))

## Chapter 2

# Reference for unit 'AVL\_Tree'

### 2.1 Used units

Table 2.1: Used units by unit 'AVL\_Tree'

Name	Page
Classes	??
sysutils	??

### 2.2 Overview

The `avl_tree` unit implements a general-purpose AVL (balanced) tree class: the `TAVLTree` (96) class and it's associated data node class `TAVLTreeNode` (110).

### 2.3 Constants, types and variables

#### 2.3.1 Types

`PAVLTreeNode` = `^TAVLTreeNode`

Pointer to `TAVLTreeNode`

`TAVLTreeClass` = `Class of TAVLTree`

`TAVLTreeClass` is the class of `TAVLTree` (96).

`TAVLTreeNodeClass` = `Class of TAVLTreeNode`

`TAVLTreeNodeClass` is the class of `TAVLTreeNode` (110). It is the type of the `TAVLTree.NodeClass` (109) property and determines what class of nodes will be created by the tree.

`TObjectSortCompare` = `function(Tree: TAVLTree; Data1: Pointer;  
Data2: Pointer) : Integer of object`



`TObjectSortCompare` is the prototype for the `TAVLTree.OnObjectCompare` (109) property. When assigned, it is used to sort the elements in the tree. It provides more information than the standard `TListSortCompare` handler used in `TAVLTree.OnCompare` (109): it also passes the tree to the sort mechanism.

### 2.3.2 Variables

`NodeMemManager` : `TAVLTreeNodeMemManager`

`NodeMemManager` is the default node manager for a new instance of `TAVLTree`.

## 2.4 TAVLTree

### 2.4.1 Description

`TAVLTree` maintains a balanced AVL tree. The tree consists of `TAVLTreeNode` (110) 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` (109) property.

See also: `TAVLTreeNode` (110)

## 2.4.2 Method overview

Page	Method	Description
99	Add	Add a new node to the tree
100	AddAscendingSequence	
103	Assign	Assign another tree
101	Clear	Clears the tree
103	Compare	Compare 2 nodes
108	ConsistencyCheck	Check the consistency of the tree
98	Create	Create a new instance of TAVLTree
98	CreateObjectCompare	Create an instance of the tree with extended compare method
100	Delete	Delete a node from the tree
98	Destroy	Destroy the TAVLTree instance
99	DisposeNode	Dispose of a node outside of the tree
102	Equals	Check if two trees are equal
103	Find	Find a data item in the tree.
105	FindHighest	Find the highest (rightmost) node in the tree.
103	FindKey	Find a data item in the tree using alternate compare mechanism
106	FindLeftMost	Find the node most left to a specified data node
106	FindLeftMostKey	Find the node most left to a specified key node
107	FindLeftMostSameKey	Find the node most left to a specified node with the same data
105	FindLowest	Find the lowest (leftmost) node in the tree.
105	FindNearest	Find the node closest to the data in the tree
104	FindNearestKey	Find nearest key for a data pointer
105	FindPointer	Search for a data pointer
104	FindPrecessor	
106	FindRightMost	Find the node most right to a specified node
106	FindRightMostKey	Find the node most right to a specified key node
107	FindRightMostSameKey	Find the node most right of a specified node with the same data
104	FindSuccessor	Find successor to node
102	FreeAndClear	Clears the tree and frees nodes
102	FreeAndDelete	Delete a node from the tree and destroy it
107	GetEnumerator	Get an enumerator for the tree.
107	GetEnumeratorHighToLow	Return an enumerator that enumerates the tree in reversed order
102	IsEqual	Check whether 2 tree instances are equal.
101	MoveDataLeftMost	Move data to the nearest left element
101	MoveDataRightMost	Move data to the nearest right element
99	NewNode	Create a new tree node
108	NodeToReportStr	Create a textual dump of the tree
100	Remove	Remove a data item from the list.
101	RemovePointer	Remove a pointer item from the list.
108	ReportAsString	Return the tree report as a string
98	SetNodeManager	Set the node instance manager to use
108	WriteReportToStream	Write the contents of the tree consistency check to the stream

### 2.4.3 Property overview

Page	Properties	Access	Description
<a href="#">110</a>	Count	r	Number of nodes in the tree.
<a href="#">109</a>	NodeClass	rw	Node class to create
<a href="#">109</a>	OnCompare	rw	Compare function used when comparing nodes
<a href="#">109</a>	OnObjectCompare	rw	Compare handler
<a href="#">109</a>	Root	r	Root node of the tree

### 2.4.4 TAVLTree.Create

Synopsis: Create a new instance of TAVLTree

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

Visibility: public

Description: `Create` initializes a new instance of TAVLTree ([96](#)). An alternate `OnCompare` ([109](#)) can be provided: the default `OnCompare` method compares the 2 data pointers of a node.

See also: `OnCompare` ([109](#))

### 2.4.5 TAVLTree.CreateObjectCompare

Synopsis: Create an instance of the tree with extended compare method

Declaration: `constructor CreateObjectCompare`  
`(const OnCompareMethod: TObjectSortCompare)`

Visibility: public

Description: `CreateObjectCompare` is an alternative constructor that accepts a `TObjectSortCompare` ([96](#)) compare function instead of a regular `TListSortCompare` compare function. The compare function can still be set in the `TAVLTree.OnObjectCompare` ([109](#)) property.

See also: `TAVLTree.OnObjectCompare` ([109](#))

### 2.4.6 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` ([98](#)), `TAVLTree.Clean` ([96](#))

### 2.4.7 TAVLTree.SetNodeManager

Synopsis: Set the node instance manager to use

Declaration: `procedure SetNodeManager(NewMgr: TBaseAVLTreeNodeManager;`  
`AutoFree: Boolean)`

Visibility: public

**Description:** `SetNodeManager` sets the node manager instance used by the tree to `newmgr`. It should be called before any nodes are added to the tree. The `TAVLTree` instance will not destroy the `nodemanager`, thus the same instance of the tree node manager can be used to manager the nodes of multiple `TAVLTree` instances.

By default, a single instance of `TAVLTreeNodeMemManager` (113) is used to manage the nodes of all `TAVLTree` instances.

See also: `TBaseAVLTreeNodeManager` (115), `TAVLTreeNodeMemManager` (113)

### 2.4.8 TAVLTree.NewNode

Synopsis: Create a new tree node

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

Visibility: public

**Description:** `NewNode` creates a new node, but does not insert it in the tree. It will use the node manager if that is set. If it is not set then the `TAVLTree.NodeClass` (109) class is used to create a new node.

See also: `TAVLTree.NodeClass` (109), `TAVLTree.Add` (99), `TAVLTree.DisposeNode` (99)

### 2.4.9 TAVLTree.DisposeNode

Synopsis: Dispose of a node outside of the tree

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

Visibility: public

**Description:** `DisposeNode` disposes of a node outside of the tree. If the node manager is set, the node is returned to the manager, otherwise it is freed. Do not use this on a node that is still in the tree.

**Errors:** If use on a node in the tree, no error will happen, but the tree will no longer be correct and access violations may happen later on.

See also: `TAVLTree.NewNode` (99)

### 2.4.10 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` (100), `TAVLTree.Remove` (100)

### 2.4.11 TAVLTree.AddAscendingSequence

Synopsis:

Declaration: `function AddAscendingSequence(Data: Pointer; LastAdded: TAVLTreeNode;  
var Successor: TAVLTreeNode) : TAVLTreeNode`

Visibility: public

Description: `AddAscendingSequence` is an optimized version of `Add` (99) for quickly adding an ascending sequence of nodes. It adds `Data` between `LastAdded` and `Successor` as a state and skips searching for an insert position. For nodes with same value the order of the sequence is kept.

It can be used as follows:

```
LastNode:=nil; // TAVLTreeNode
Successor:=nil; // TAVLTreeNode
for i:=1 to 1000 do
  LastNode:=Tree.AddAscendingSequence(TItem.Create(i), LastNode, Successor);
```

If `LastAdded` is `Nil` a regular add is performed.

Errors: If the nodes are not in ascending order, the tree will not be consistent.

See also: `TAVLTree.Add` (99)

### 2.4.12 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` (113) 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` (100), `TAVLTree.RemovePointer` (101), `TAVLTree.Clear` (101)

### 2.4.13 TAVLTree.Remove

Synopsis: Remove a data item from the list.

Declaration: `function Remove(Data: Pointer) : Boolean`

Visibility: public

Description: `Remove` finds the node associated with `Data` using `find` (103) and, if found, deletes it from the tree. Only the first occurrence of `Data` will be removed.

See also: `TAVLTree.Delete` (100), `TAVLTree.RemovePointer` (101), `TAVLTree.Clear` (101), `TAVLTree.Find` (103)

#### 2.4.14 TAVLTree.RemovePointer

Synopsis: Remove a pointer item from the list.

Declaration: `function RemovePointer(Data: Pointer) : Boolean`

Visibility: `public`

Description: `Remove` uses `FindPointer` (105) 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` (100), `TAVLTree.Delete` (100), `TAVLTree.Clear` (101)

#### 2.4.15 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` (101)

#### 2.4.16 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` (101)

#### 2.4.17 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` (102) instead.

See also: `TAVLTree.FreeAndClear` (102), `TAVLTree.Delete` (100)

### 2.4.18 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` ([101](#)), `TAVLTree.Delete` ([100](#)), `TAVLTree.FreeAndDelete` ([102](#))

### 2.4.19 TAVLTree.FreeAndDelete

Synopsis: Delete a node from the tree and destroy it

Declaration: `procedure FreeAndDelete(ANode: TAVLTreeNode); Virtual`

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` ([101](#)), `TAVLTree.Delete` ([100](#)), `TAVLTree.FreeAndClear` ([102](#))

### 2.4.20 TAVLTree.Equals

Synopsis: Check if two trees are equal

Declaration: `function Equals(Obj: TObject) : Boolean; Override`

Visibility: `public`

Description: `Equals` checks, when passed an `TAVLTree`, whether the tree is equal (using `TAVLTree.IsEqual` ([102](#)), comparing keys only). If another type of object is passed, the inherited `IsEqual` is called.

Errors: None.

See also: `TAVLTree.IsEqual` ([102](#))

### 2.4.21 TAVLTree.IsEqual

Synopsis: Check whether 2 tree instances are equal.

Declaration: `function IsEqual(aTree: TAVLTree; CheckDataPointer: Boolean) : Boolean`

Visibility: `public`

Description: `IsEqual` checks the current tree with `aTree` and checks whether the two trees contain the same data in the same order and whether they use the same compare methods, and node class. If `CheckDataPointer` is `True`, only the data pointers are compared, not the keys.

Errors: None.

See also: `TAVLTree.Equals` ([102](#))

### 2.4.22 TAVLTree.Assign

Synopsis: Assign another tree

Declaration: `procedure Assign(aTree: TAVLTree); Virtual`

Visibility: public

Description: `Assign` copies all data from `aTree` to the current tree if they are not equal. The current tree is cleared first. Note that the compare function(s) and class node are not copied, only the data.

Errors: If you pass nil, an exception is raised.

See also: `TAVLTree.IsEqual` (102)

### 2.4.23 TAVLTree.Compare

Synopsis: Compare 2 nodes

Declaration: `function Compare(Data1: Pointer; Data2: Pointer) : Integer`

Visibility: public

Description: `Compare` compares the keys from 2 data pointers. It uses the appropriate compare function `TAVLtree.OnCompare` (109) or `TAVLTree.OnObjectCompare` (109) to do so. The result is

- negative if the first key comes before the second
- 0 when the two keys are equal.
- positive if the second key comes before the first

See also: `TAVLTree.OnObjectCompare` (109), `TAVLtree.OnCompare` (109)

### 2.4.24 TAVLTree.Find

Synopsis: Find a data item in the tree.

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

Visibility: public

Description: `Find` uses the default `OnCompare` (109) 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` (105).

See also: `OnCompare` (109), `FindKey` (103)

### 2.4.25 TAVLTree.FindKey

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

Declaration: `function FindKey(Key: Pointer;  
const 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: `OnCompare` ([109](#)), `Find` ([103](#))

### 2.4.26 TAVLTree.FindNearestKey

**Synopsis:** Find nearest key for a data pointer

**Declaration:** `function FindNearestKey(Key: Pointer;  
const OnCompareKeyWithData: TListSortCompare)  
: TAVLTreeNode`

**Visibility:** public

**Description:** `FindNearestKey` attempts to find the nearest possible key in the tree using the `OnCompareKeyWithData` function. It returns the closest possible key in the tree.

**Errors:** None.

See also: `TAVLTree.FindKey` ([103](#))

### 2.4.27 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` ([104](#)), `TAVLTree.MoveDataRightMost` ([101](#))

### 2.4.28 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` ([104](#)), `TAVLTree.MoveDataLeftMost` ([101](#))

### 2.4.29 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: `FindHighest` ([105](#))

### 2.4.30 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: `FindLowest` ([105](#))

### 2.4.31 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: `FindHighest` ([105](#)), `FindLowest` ([105](#)), `Find` ([103](#)), `FindKey` ([103](#))

### 2.4.32 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` ([103](#)), where a custom compare function can be used.

The default `OnCompare` ([109](#)) compares the data pointers, so the default `Find` will return the same node as `FindPointer`

See also: `TAVLTree.Find` ([103](#)), `TAVLTree.FindKey` ([103](#))

### 2.4.33 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` (106), `TAVLTree.FindLeftMostKey` (106), `TAVLTree.FindRightMostKey` (106)

### 2.4.34 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` (106), `TAVLTree.FindLeftMostKey` (106), `TAVLTree.FindRightMostKey` (106)

### 2.4.35 TAVLTree.FindLeftMostKey

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

Declaration: `function FindLeftMostKey (Key: Pointer;  
const 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` (106), `TAVLTree.FindRightMost` (106), `TAVLTree.FindRightMostKey` (106)

### 2.4.36 TAVLTree.FindRightMostKey

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

Declaration: `function FindRightMostKey (Key: Pointer;  
const 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` (106), `TAVLTree.FindRightMost` (106), `TAVLTree.FindLeftMostKey` (106)

### 2.4.37 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` (106), `TAVLTree.FindLeftMostKey` (106), `TAVLTree.FindRightMostSameKey` (107)

### 2.4.38 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` (106), `TAVLTree.FindRightMostKey` (106), `TAVLTree.FindLeftMostSameKey` (107)

### 2.4.39 TAVLTree.GetEnumerator

Synopsis: Get an enumerator for the tree.

Declaration: `function GetEnumerator : TAVLTreeNodeEnumerator`

Visibility: public

Description: `GetEnumerator` returns an instance of the standard tree node enumerator `TAVLTreeNodeEnumerator` (111).

See also: `TAVLTreeNodeEnumerator` (111)

### 2.4.40 TAVLTree.GetEnumeratorHighToLow

Synopsis: Return an enumerator that enumerates the tree in reversed order

Declaration: `function GetEnumeratorHighToLow : TAVLTreeNodeEnumerator`

Visibility: public

Description: `GetEnumeratorHighToLow` returns an enumerator that traverses the tree in reversed order.

See also: `TAVLTree.GetEnumerator` (107)

### 2.4.41 TAVLTree.ConsistencyCheck

Synopsis: Check the consistency of the tree

Declaration: `procedure ConsistencyCheck; Virtual`

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` ([108](#))

### 2.4.42 TAVLTree.WriteReportToStream

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

Declaration: `procedure WriteReportToStream(s: TStream)`

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` ([108](#))

### 2.4.43 TAVLTree.NodeToReportStr

Synopsis: Create a textual dump of the tree

Declaration: `function NodeToReportStr(aNode: TAVLTreeNode) : string; Virtual`

Visibility: `public`

Description: `NodeToReportStr` creates a textual representation of a node. It is called by `TAVLTree.ReportAsString` ([108](#)) for debugging purposes. It prints the data pointer as a hex value. Override this to create a human-readable representation of the data.

See also: `TAVLTree.ReportAsString` ([108](#))

### 2.4.44 TAVLTree.ReportAsString

Synopsis: Return the tree report as a string

Declaration: `function ReportAsString : string`

Visibility: `public`

Description: `ReportAsString` calls `WriteReportToStream` ([108](#)) and returns the stream data as a string.

See also: `TAVLTree.WriteReportToStream` ([108](#))

### 2.4.45 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` (98)

### 2.4.46 TAVLTree.OnObjectCompare

Synopsis: Compare handler

Declaration: `Property OnObjectCompare : TObjectSortCompare`

Visibility: public

Access: Read,Write

Description: `OnObjectCompare` is used to compare nodes. It is only used if `TAVLTree.OnCompare` (109) is not set.

See also: `TAVLTree.OnCompare` (109)

### 2.4.47 TAVLTree.NodeClass

Synopsis: Node class to create

Declaration: `Property NodeClass : TAVLTreeNodeClass`

Visibility: public

Access: Read,Write

Description: `NodeClass` is the class of nodes to create when adding new nodes: `TAVLTree.NewNode` (99) will use this class when creating a new node. This can be set to a descendent class of `TAVLTreeNode` (110), but not if there are already nodes in the tree.

See also: `TAVLTreeNode` (110), `TAVLTree.NewNode` (99)

### 2.4.48 TAVLTree.Root

Synopsis: Root node of the tree

Declaration: `Property Root : TAVLTreeNode`

Visibility: public

Access: Read

Description: `Root` is the root node of the tree. It should not be set explicitly, only use the `Add` (99), `Delete` (100), `Remove` (100), `RemovePointer` (101), or `Clear` (101) methods to manipulate the items in the tree.

See also: `TAVLTree.Add` (99), `TAVLTree.Delete` (100), `TAVLTree.Remove` (100), `TAVLTree.RemovePointer` (101), `TAVLTree.Clear` (101)

### 2.4.49 TAVLTree.Count

Synopsis: Number of nodes in the tree.

Declaration: `Property Count : SizeInt`

Visibility: `public`

Access: `Read`

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

## 2.5 TAVLTreeNode

### 2.5.1 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.

See also: `TAVLTree` ([96](#)), `TAVLTreeNode.Data` ([??](#))

### 2.5.2 Method overview

Page	Method	Description
<a href="#">111</a>	<code>Clear</code>	Clears the node's data
<a href="#">111</a>	<code>ConsistencyCheck</code>	Check consistency of the node and below nodes.
<a href="#">111</a>	<code>GetCount</code>	Get the number of nodes
<a href="#">110</a>	<code>Precessor</code>	Preceding node in the tree
<a href="#">110</a>	<code>Successor</code>	Succeeding node in the tree
<a href="#">111</a>	<code>TreeDepth</code>	Level of the node in the tree below

### 2.5.3 TAVLTreeNode.Successor

Synopsis: Succeeding node in the tree

Declaration: `function Successor : TAVLTreeNode`

Visibility: `public`

Description: `Successor` calculates and return the succeeding (right) node in the tree. For the last node, `Nil` is returned.

See also: `TAVLTreeNode.Precessor` ([110](#))

### 2.5.4 TAVLTreeNode.Precessor

Synopsis: Preceding node in the tree

Declaration: `function Precessor : TAVLTreeNode`

Visibility: `public`

Description: `Precessor` calculates and return the preceding (left) node in the tree. For the first node, `Nil` is returned.

See also: `TAVLTreeNode.Successor` ([110](#))

### 2.5.5 TAVLTreeNode.Clear

Synopsis: Clears the node's data

Declaration: `procedure Clear`

Visibility: `public`

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

### 2.5.6 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: `Balance` (??)

### 2.5.7 TAVLTreeNode.ConsistencyCheck

Synopsis: Check consistency of the node and below nodes.

Declaration: `procedure ConsistencyCheck (Tree: TAVLTree); Virtual`

Visibility: `public`

Description: `ConsistencyCheck` checks whether the node and nodes below are consistent, i.e. the nodes are still ordered correctly: left nodes are before right nodes.

Errors: If an inconsistency is detected, an exception is raised.

### 2.5.8 TAVLTreeNode.GetCount

Synopsis: Get the number of nodes

Declaration: `function GetCount : SizeInt`

Visibility: `public`

Description: `GetCount` returns 1 plus the number of subnodes, if any.

Errors: None.

## 2.6 TAVLTreeNodeEnumerator

### 2.6.1 Description

`TAVLTreeNodeEnumerator` is a class which implements the enumerator interface for the `TAVL-Tree` (96). It enumerates all the nodes in the tree.

See also: `TAVLTree` (96)



### 2.6.2 Method overview

Page	Method	Description
<a href="#">112</a>	Create	Create a new instance of TAVLTreeNodeEnumerator
<a href="#">112</a>	GetEnumerator	Returns the enumerator
<a href="#">112</a>	MoveNext	Move to next node in the tree.

### 2.6.3 Property overview

Page	Properties	Access	Description
<a href="#">112</a>	Current	r	Current node in the tree
<a href="#">113</a>	LowToHigh	r	Should the enumerator return nodes from low to high or high to low

### 2.6.4 TAVLTreeNodeEnumerator.Create

Synopsis: Create a new instance of TAVLTreeNodeEnumerator

Declaration: `constructor Create(Tree: TAVLTree; aLowToHigh: Boolean)`

Visibility: public

Description: `Create` creates a new instance of `TAVLTreeNodeEnumerator` and saves the `Tree` argument for later use in the enumerator.

### 2.6.5 TAVLTreeNodeEnumerator.GetEnumerator

Synopsis: Returns the enumerator

Declaration: `function GetEnumerator : TAVLTreeNodeEnumerator`

Visibility: public

Description: `GetEnumerator` returns `Self`..

### 2.6.6 TAVLTreeNodeEnumerator.MoveNext

Synopsis: Move to next node in the tree.

Declaration: `function MoveNext : Boolean`

Visibility: public

Description: `MoveNext` will return the lowest node in the tree to start with, and for all other calls returns the successor node of the current node with `TAVLTree.FindSuccessor` ([104](#)).

See also: `TAVLTree.FindSuccessor` ([104](#))

### 2.6.7 TAVLTreeNodeEnumerator.Current

Synopsis: Current node in the tree

Declaration: `Property Current : TAVLTreeNode`

Visibility: public

Access: Read

Description: `Current` is the current node in the enumeration.

See also: `TAVLTreeNodeEnumerator.MoveNext` ([112](#))

## 2.6.8 `TAVLTreeNodeEnumerator.LowToHigh`

Synopsis: Should the enumerator return nodes from low to high or high to low

Declaration: `Property LowToHigh : Boolean`

Visibility: `public`

Access: `Read`

Description: `LowToHigh` determines whether the tree is walked from low to high or high to low. It's value is set in the constructor and cannot be changed while enumerating the tree nodes.

See also: `TAVLTreeNodeEnumerator.Create` ([112](#))

## 2.7 `TAVLTreeNodeMemManager`

### 2.7.1 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.

See also: `TAVLTreeNode` ([110](#)), `TAVLTree` ([96](#))

### 2.7.2 Method overview

Page	Method	Description
<a href="#">114</a>	<code>Clear</code>	Frees all unused nodes
<a href="#">114</a>	<code>Create</code>	Create a new instance of <code>TAVLTreeNodeMemManager</code>
<a href="#">114</a>	<code>Destroy</code>	
<a href="#">113</a>	<code>DisposeNode</code>	Return a node to the free list
<a href="#">114</a>	<code>NewNode</code>	Create a new <code>TAVLTreeNode</code> instance

### 2.7.3 Property overview

Page	Properties	Access	Description
<a href="#">115</a>	<code>Count</code>	<code>r</code>	Number of nodes in the list.
<a href="#">115</a>	<code>MaximumFreeNodeRatio</code>	<code>rw</code>	Maximum amount of free nodes in the list
<a href="#">115</a>	<code>MinimumFreeNode</code>	<code>rw</code>	Minimum amount of free nodes to be kept.

### 2.7.4 `TAVLTreeNodeMemManager.DisposeNode`

Synopsis: Return a node to the free list

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

Visibility: `public`

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

See also: `TAVLTreeNodeMemManager.NewNode` ([114](#))

### 2.7.5 `TAVLTreeNodeMemManager.NewNode`

**Synopsis:** Create a new `TAVLTreeNode` instance

**Declaration:** `function NewNode : TAVLTreeNode; Override`

**Visibility:** `public`

**Description:** `NewNode` returns a new `TAVLTreeNode` ([110](#)) instance. If there is a node in the free list, it is returned. If no more free nodes are present, a new node is created.

See also: `TAVLTreeNodeMemManager.DisposeNode` ([113](#))

### 2.7.6 `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` ([115](#)), `TAVLTreeNodeMemManager.MaximumFreeNodeRatio` ([115](#))

### 2.7.7 `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` ([114](#))

### 2.7.8 `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` ([114](#))

## 2.7.9 TAVLTreeNodeMemManager.MinimumFreeNode

Synopsis: Minimum amount of free nodes to be kept.

Declaration: `Property MinimumFreeNode : SizeInt`

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` (115)

## 2.7.10 TAVLTreeNodeMemManager.MaximumFreeNodeRatio

Synopsis: Maximum amount of free nodes in the list

Declaration: `Property MaximumFreeNodeRatio : SizeInt`

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` (115), `TAVLTreeNodeMemManager.MinimumFreeNode` (115)

## 2.7.11 TAVLTreeNodeMemManager.Count

Synopsis: Number of nodes in the list.

Declaration: `Property Count : SizeInt`

Visibility: `public`

Access: `Read`

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

See also: `TAVLTreeNodeMemManager.MinimumFreeNode` (115), `TAVLTreeNodeMemManager.MaximumFreeNodeRatio` (115)

## 2.8 TBaseAVLTreeNodeManager

### 2.8.1 Description

`TBaseAVLTreeNodeManager` is an abstract class from which a descendent can be created that manages creating and disposing of tree nodes (instances of `TAVLTreeNode` (110)) for a `TAVLTree` (96) tree instance. No instance of this class should be created, it is a purely abstract class. The default descendant of this class used by an `TAVLTree` instance is `TAVLTreeNodeMemManager` (113).

The `TAVLTree.SetNodeManager` (98) method can be used to set the node manager that a `TAVLTree` instance should use.

See also: `TAVLTreeNodeMemManager` (113), `TAVLTree.SetNodeManager` (98), `TAVLTreeNode` (110)

### 2.8.2 Method overview

Page	Method	Description
<a href="#">116</a>	<code>DisposeNode</code>	Called when the AVL tree no longer needs node
<a href="#">116</a>	<code>NewNode</code>	Called when the AVL tree needs a new node

### 2.8.3 TBaseAVLTreeNodeManager.DisposeNode

Synopsis: Called when the AVL tree no longer needs node

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

Visibility: `public`

Description: `DisposeNode` is called by `TAVLTree` ([96](#)) when it no longer needs a `TAVLTreeNode` ([110](#)) instance. The manager may decide to re-use the instance for later use instead of destroying it.

See also: `TBaseAVLTreeNodeManager.NewNode` ([116](#)), `TAVLTree.Delete` ([100](#)), `TAVLTreeNode` ([110](#))

### 2.8.4 TBaseAVLTreeNodeManager.NewNode

Synopsis: Called when the AVL tree needs a new node

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

Visibility: `public`

Description: `NewNode` is called by `TAVLTree` ([96](#)) when it needs a new node in `TAVLTree.Add` ([99](#)). It must be implemented by descendants to return a new `TAVLTreeNode` ([110](#)) instance.

See also: `TBaseAVLTreeNodeManager.DisposeNode` ([116](#)), `TAVLTree.Add` ([99](#)), `TAVLTreeNode` ([110](#))

## Chapter 3

# Reference for unit 'base64'

### 3.1 Used units

Table 3.1: Used units by unit 'base64'

Name	Page
Classes	??
sysutils	??

### 3.2 Overview

`base64` implements base64 encoding (as used for instance in MIME encoding) based on streams. It implements 2 streams which encode or decode anything written or read from it. The source or the destination of the encoded data is another stream. 2 classes are implemented for this: `TBase64EncodingStream` ([121](#)) for encoding, and `TBase64DecodingStream` ([118](#)) for decoding.

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

### 3.3 Constants, types and variables

#### 3.3.1 Types

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

Table 3.2: Enumeration values for type `TBase64DecodingMode`

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

`TBase64DecodingMode` determines the decoding algorithm used by `TBase64DecodingStream` ([118](#)). 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` (118) exception is raised.

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

## 3.4 Procedures and functions

### 3.4.1 DecodeStringBase64

Synopsis: Decodes a Base64 encoded string and returns the decoded data as a string.

Declaration: `function DecodeStringBase64(const s: string; strict: Boolean) : string`

Visibility: default

Description: `DecodeStringBase64` decodes the string `s` (containing Base 64 encoded data) returns the decoded data as a string. It uses a `TBase64DecodingStream` (118) to do this. The `Strict` parameter is passed on to the constructor as `bdmStrict` or `bdmMIME`

See also: `DecodeStringBase64` (118), `TBase64DecodingStream` (118)

### 3.4.2 EncodeStringBase64

Synopsis: Encode a string with Base64 encoding and return the result as a string.

Declaration: `function EncodeStringBase64(const s: string) : string`

Visibility: default

Description: `EncodeStringBase64` encodes the string `s` using Base 64 encoding and returns the result. It uses a `TBase64EncodingStream` (121) to do this.

See also: `DecodeStringBase64` (118), `TBase64EncodingStream` (121)

## 3.5 EBase64DecodingException

### 3.5.1 Description

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

## 3.6 TBase64DecodingStream

### 3.6.1 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.

See also: `TBase64EncodingStream` (121)

### 3.6.2 Method overview

Page	Method	Description
<a href="#">119</a>	Create	Create a new instance of the <code>TBase64DecodingStream</code> class
<a href="#">119</a>	Read	Read and decrypt data from the source stream
<a href="#">119</a>	Reset	Reset the stream
<a href="#">120</a>	Seek	Set stream position.

### 3.6.3 Property overview

Page	Properties	Access	Description
<a href="#">120</a>	EOF	r	
<a href="#">120</a>	Mode	rw	Decoding mode

### 3.6.4 TBase64DecodingStream.Create

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

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

Visibility: public

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

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

See also: `TBase64EncodingStream.Create` ([121](#)), `TBase64DecodingMode` ([117](#))

### 3.6.5 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` ([120](#)), `TBase64DecodingStream.Read` ([119](#))

### 3.6.6 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` ([118](#)), `TBase64DecodingStream.Seek` ([120](#)), `TStream.Read` ([??](#))

### 3.6.7 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` ([119](#)), `TBase64DecodingStream.Write` ([118](#)), `TBase64EncodingStream.Seek` ([122](#)), `TStream.Seek` ([??](#))

### 3.6.8 TBase64DecodingStream.EOF

**Synopsis:**

**Declaration:** `Property EOF : Boolean`

**Visibility:** public

**Access:** Read

**Description:**

### 3.6.9 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` ([118](#))

## 3.7 TBase64EncodingStream

### 3.7.1 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.

By the nature of base64 encoding, when a buffer is written to the stream, the output stream does not yet contain all output: input must be a multiple of 3. In order to be sure that the output contains all encoded bytes, the Flush ([121](#)) method can be used. The destructor will automatically call Flush, so all data is written to the destination stream when the decodes is destroyed.

See also: TBase64DecodingStream ([118](#))

### 3.7.2 Method overview

Page	Method	Description
<a href="#">121</a>	Destroy	Remove a TBase64EncodingStream instance from memory
<a href="#">121</a>	Flush	Flush the remaining bytes to the output stream.
<a href="#">122</a>	Seek	Position the stream
<a href="#">122</a>	Write	Write data to the stream.

### 3.7.3 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 ([121](#))

### 3.7.4 TBase64EncodingStream.Flush

Synopsis: Flush the remaining bytes to the output stream.

Declaration: `function Flush : Boolean`

Visibility: `public`

Description: Flush writes the remaining bytes from the internal encoding buffer to the output stream and pads the output with "=" signs. It returns True if padding was necessary, and False if not.

See also: TBase64EncodingStream.Destroy ([121](#))

### 3.7.5 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` ([122](#)), `TBase64EncodingStream.Read` ([121](#)), `TBase64DecodingStream.Write` ([118](#)), `TStream.Write` ([??](#))

### 3.7.6 TBase64EncodingStream.Seek

Synopsis: Position the stream

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

Visibility: `public`

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

Errors: An `EStreamError` error is raised.

See also: `TBase64EncodingStream.Read` ([121](#)), `TBase64EncodingStream.Write` ([122](#)), `TStream.Seek` ([??](#))

## Chapter 4

# Reference for unit 'BlowFish'

### 4.1 Used units

Table 4.1: Used units by unit 'BlowFish'

Name	Page
Classes	??
sysutils	??

### 4.2 Overview

The BlowFish implements a class TBlowFish ([124](#)) to handle Blowfish encryption/decryption of memory buffers, and 2 TStream (??) descendents TBlowFishDeCryptStream ([125](#)) which decrypts any data that is read from it on the fly, as well as TBlowFishEnCryptStream ([126](#)) which encrypts the data that is written to it on the fly.

### 4.3 Constants, types and variables

#### 4.3.1 Constants

`BFRounds = 16`

Number of rounds in Blowfish encryption.

#### 4.3.2 Types

`PBlowFishKey = ^TBlowFishKey`

PBlowFishKey is a simple pointer to a TBlowFishKey ([124](#)) array.

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

`TBFBlock` is the basic data structure used by the encrypting/decrypting routines in `TBlowFish` (124), `TBlowFishDeCryptStream` (125) and `TBlowFishEnCryptStream` (126). It is the basic encryption/decryption block for all encrypting/decrypting: all encrypting/decrypting happens on a `TBFBlock` structure.

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

`TBlowFishKey` is a data structure which keeps the encryption or decryption key for the `TBlowFish` (124), `TBlowFishDeCryptStream` (125) and `TBlowFishEnCryptStream` (126) classes. It should be filled with the encryption key and passed to the constructor of one of these classes.

## 4.4 EBlowFishError

### 4.4.1 Description

`EBlowFishError` is used by the `TBlowFishStream` (128), `TBlowFishEncryptStream` (126) and `TBlowFishDecryptStream` (125) classes to report errors.

See also: `TBlowFishStream` (128), `TBlowFishEncryptStream` (126), `TBlowFishDecryptStream` (125)

## 4.5 TBlowFish

### 4.5.1 Description

`TBlowFish` is a simple class that can be used to encrypt/decrypt a single `TBFBlock` (123) data block with the `Encrypt` (125) and `Decrypt` (125) calls. It is used internally by the `TBlowFishEnCryptStream` (126) and `TBlowFishDeCryptStream` (125) classes to encrypt or decrypt the actual data.

See also: `TBlowFishEnCryptStream` (126), `TBlowFishDeCryptStream` (125)

### 4.5.2 Method overview

Page	Method	Description
124	Create	Create a new instance of the <code>TBlowFish</code> class
125	Decrypt	Decrypt a block
125	Encrypt	Encrypt a block

### 4.5.3 TBlowFish.Create

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

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

**Visibility:** public

**Description:** `Create` initializes a new instance of the `TBlowFish` class: it stores the key `Key` in the internal data structures so it can be used in later calls to `Encrypt` (125) and `Decrypt` (125).

See also: `Encrypt` (125), `Decrypt` (125)

#### 4.5.4 TBlowFish.Encrypt

Synopsis: Encrypt a block

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

Visibility: public

Description: `Encrypt` encrypts the data in `Block` (always 8 bytes) using the key (124) specified when the `TBlowFish` instance was created.

See also: `TBlowFishKey` (124), `Decrypt` (125), `Create` (124)

#### 4.5.5 TBlowFish.Decrypt

Synopsis: Decrypt a block

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

Visibility: public

Description: `ncrypt` decrypts the data in `Block` (always 8 bytes) using the key (124) specified when the `TBlowFish` instance was created. The data must have been encrypted with the same key and the `Encrypt` (125) call.

See also: `TBlowFishKey` (124), `Encrypt` (125), `Create` (124)

### 4.6 TBlowFishDeCryptStream

#### 4.6.1 Description

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

This is a read-only stream: it is seekable only in a forward direction, and data can only be read from it, writing is not possible. For writing data so it is encrypted, the `TBlowFishEncryptStream` (126) stream must be used.

See also: `Create` (128), `TBlowFishEncryptStream` (126)

#### 4.6.2 Method overview

Page	Method	Description
<a href="#">125</a>	<code>Create</code>	
<a href="#">126</a>	<code>Read</code>	Read data from the stream
<a href="#">126</a>	<code>Seek</code>	Set the stream position.

#### 4.6.3 TBlowFishDeCryptStream.Create

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

Visibility: public

#### 4.6.4 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: Create (128), TBlowFishEncryptStream (126)

#### 4.6.5 TBlowFishDecryptStream.Seek

Synopsis: Set the stream position.

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; 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 (124) exception will be raised.

See also: Read (126), EBlowFishError (124)

### 4.7 TBlowFishEncryptStream

#### 4.7.1 Description

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

This is a write-only stream: it is not seekable, and data can only be written to it, reading is not possible. For reading encrypted data, the TBlowFishDecryptStream (125) stream must be used.

See also: Create (128), TBlowFishDecryptStream (125)

#### 4.7.2 Method overview

Page	Method	Description
<a href="#">127</a>	Destroy	Free the TBlowFishEncryptStream
<a href="#">127</a>	Flush	Flush the encryption buffer
<a href="#">127</a>	Seek	Set the position in the stream
<a href="#">127</a>	Write	Write data to the stream

### 4.7.3 TBlowFishEncryptStream.Destroy

Synopsis: Free the TBlowFishEncryptStream

Declaration: destructor Destroy; Override

Visibility: public

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

See also: Flush ([127](#)), Create ([128](#))

### 4.7.4 TBlowFishEncryptStream.Write

Synopsis: Write data to the stream

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

Visibility: public

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

Data is always written 4 bytes at a time, since this is the amount of bytes required by the Blowfish algorithm. If no multiple of 4 was written to the destination stream, the Flush ([127](#)) mechanism can be used to write the remaining bytes.

See also: TBlowFishEncryptStream.Read ([126](#))

### 4.7.5 TBlowFishEncryptStream.Seek

Synopsis: Set the position in the stream

Declaration: function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64  
; 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 ([124](#)) exception.

See also: TBlowFishEncryptStream.Write ([127](#))

### 4.7.6 TBlowFishEncryptStream.Flush

Synopsis: Flush the encryption buffer

Declaration: procedure Flush

Visibility: public



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

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

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

See also: Write (127), TBFBlock (123)

## 4.8 TBlowFishStream

### 4.8.1 Description

TBlowFishStream is an abstract class which is used as a parent class for TBlowFishEncryptStream (126) and TBlowFishDecryptStream (125). It simply provides a constructor and storage for a TBlowFish (124) 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`.

See also: [TBlowFishEncryptStream \(126\)](#), [TBlowFishDecryptStream \(125\)](#), [TBlowFish \(124\)](#)

### 4.8.2 Method overview

Page	Method	Description
<a href="#">128</a>	Create	Create a new instance of the <code>TBlowFishStream</code> class
<a href="#">129</a>	Destroy	Destroy the <code>TBlowFishStream</code> instance.

### 4.8.3 Property overview

Page	Properties	Access	Description
<a href="#">129</a>	BlowFish	r	Blowfish instance used when encrypting/decrypting

#### 4.8.4 TBlowFishStream.Create

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

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

Visibility: public

**Description:** Create initializes a new instance of TBlowFishStream, and creates an internal instance of TBlowFish (124) using AKey and AKeySize. The Dest stream is stored so the descendent classes can refer to it.

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

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

See also: [TBlowFishEncryptStream \(126\)](#), [TBlowFishDecryptStream \(125\)](#), [TBlowFish \(124\)](#)

#### 4.8.5 TBlowFishStream.Destroy

Synopsis: Destroy the TBlowFishStream instance.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: Destroy cleans up the internal TBlowFish (124) instance.

See also: Create (128), TBlowFish (124)

#### 4.8.6 TBlowFishStream.BlowFish

Synopsis: Blowfish instance used when encrypting/decrypting

Declaration: `Property BlowFish : TBlowFish`

Visibility: `public`

Access: `Read`

Description: BlowFish is the TBlowFish (124) 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 (126) and TBlowFishDecryptStream (125).

See also: TBlowFishEncryptStream (126), TBlowFishDecryptStream (125), TBlowFish (124)

## Chapter 5

# Reference for unit 'BufDataset'

### 5.1 Used units

Table 5.1: Used units by unit 'BufDataset'

Name	Page
bufdataset_parser	??
Classes	??
DB	<a href="#">316</a>
sysutils	??

### 5.2 Overview

Implements a database-aware buffered dataset.

Original Authors: Joost van der Sluis and members of the Free Pascal development team.

### 5.3 Constants, types and variables

#### 5.3.1 Types

`PBlobBuffer = ^TBlobBuffer`

`PBlobBuffer` is a pointer to the `TBlobBuffer` type. `PBlobBuffer` is used to allocate and access the buffers for Blob fields (Memo or Graphic field types). `PBlobBuffer` is used in `TBufBlobField`, `TBufBlobStream`, and in `TBufDataset` methods which perform record buffer management.

`PBufBlobField = ^TBufBlobField`

Pointer to a `TBufBlobField` type

`PBufBookmark = ^TBufBookmark`

Pointer to the `TBufBookmark` type

PBufRecLinkItem = ^TBufRecLinkItem

Pointer to a TBufRecLinkItem type

```
TCompareFunc = function(subValue: pointer; aValue: pointer;
    size: Integer; options: TLocateOptions) :
    Int64
```

TCompareFunc is an Int64 function type used to compare pointers using the specified locate option values. TCompareFunc is used in the implementation of the TDBCompareRec type.

TDataPacketFormat = (dfBinary, dfXML, dfXMLUTF8, dfAny, dfDefault)

Table 5.2: Enumeration values for type TDataPacketFormat

Value	Explanation
dfAny	Data packets can be in any supported format (detected by the packet reader)
dfBinary	Data packets are in binary format
dfDefault	
dfXML	Data packets are in XML format
dfXMLUTF8	Data packets are in UTF-8-encoded XML format

Indicates the format used for data packets in TBufDataset

TDatapacketReaderClass = Class of TDatapacketReader

Class type used to create data packet readers for TBufDataset

Array = Array of TDBCompareRec

TDBCompareStruct is an array of TDBCompareRec type used to implement a structure to compare values for a group of fields in TBufDataset. TDBCompareStruct is used in the implementation of TBufIndex and its descendent classes.

Array = Array of TRecUpdateBuffer

TRecordsUpdateBuffer is an array of TRecUpdateBuffer type used to implement an array of update buffers for records in TBufDataset.

```
TResolverErrorEvent = procedure(Sender: TObject;
    DataSet: TCustomBufDataset;
    E: EUpdateError;
    UpdateKind
    : TUpdateKind;
    var Response: TResolverResponse
    )
    of object
```

`TResolverErrorEvent` is an object procedure which defines an event handler signalled when an error occurs while updating a record in `TBufDataset`. `TResolverErrorEvent` is the type used for the `TCustomBufDataset.OnUpdateError` property. Applications should create a routine which uses the signature for the type to handle the event notification. The routine should update the `Response` argument to indicate the action taken for the event notification.

`TRowState = Set of TRowStateValue`

`TRowState` is a set type used to store values from the `TRowStateValue` enumeration. `TRowState` is used to indicate the state for records loaded and saved using the data packet handler in `TBufDataset`.

`TRowStateValue = (rsvOriginal, rsvDeleted, rsvInserted, rsvUpdated, rsvDetailUpdates)`

Table 5.3: Enumeration values for type `TRowStateValue`

Value	Explanation
<code>rsvDeleted</code>	Buffer is for a record that will be deleted
<code>rsvDetailUpdates</code>	Buffer is for updates to a detail (child) dataset
<code>rsvInserted</code>	Buffer is for a record that will be inserted
<code>rsvOriginal</code>	Buffer contains the original value(s)
<code>rsvUpdated</code>	Buffer is for a record that will be updated

`TRowStateValue` is an enumeration type with values that indicate the state for field values in the row. Values in the enumeration indicate how fields in an update buffer differ from the fields in the original record data. Values from `TRowStateValue` are stored in the `TRowState` type, and used in data packet readers for `TBufDataset` (`TDataPacketReader`).

## 5.4 Procedures and functions

### 5.4.1 RegisterDatapacketReader

**Synopsis:** Registers a data packet reader using a specific format for `TBufDataset`

**Declaration:** `procedure RegisterDatapacketReader`  
`(ADatapacketReaderClass: TDatapacketReaderClass;`  
`AFormat: TDataPacketFormat)`

**Visibility:** default

**Description:** `RegisterDatapacketReader` is a procedure which registers a class type needed to create data packet readers for a specific format. `ADatapacketReaderClass` is the class type used to create new instances of `TDataPacketReader`. `AFormat` is a `TDataPacketFormat` enumeration value that indicates the format supported for data packets in the reader class.

`RegisterDatapacketReader` accesses a unit variable used to store an array of registered data packet reader classes and their associated formats. `RegisterDatapacketReader` increments the size of the array variable, and stores the values in `ADatapacketReaderClass` and `AFormat` in the `TDatapacketReaderRegistration` type. It does not perform any checks for duplicate entries, so the procedure should be called exactly once for the class type and format combination.

`RegisterDatapacketReader` is normally used in the initialization section of a unit that implements a `TDataPacketReader` descendant for a specific storage format, such as: `xmldatapacketreader.pp`.

See also: TDataPacketReader ([171](#)), TDataPacketFormat ([131](#)), TXMLDatapacketReader (??)

## 5.5 TBlobBuffer

```
TBlobBuffer = record
  FieldNo : Integer;
  OrgBufID : Integer;
  Buffer : pointer;
  Size : PtrInt;
end
```

TBlobBuffer is a record type used to represent a buffer allocated for a Blob field. Pointers to TBlobBuffer (PBlobBuffer) are used in the implementation of TBufBlobStream and TBufDataset.

## 5.6 TBufBlobField

```
TBufBlobField = record
  ConnBlobBuffer : Array[0..11] of Byte;
  BlobBuffer : PBlobBuffer;
end
```

Implements a Blob field for TBufDataset

## 5.7 TBufBookmark

```
TBufBookmark = record
  BookmarkData : PBufRecLinkItem;
  BookmarkInt
  : Integer;
  BookmarkFlag : TBookmarkFlag;
end
```

TBufBookmark is a record type used to implement a Bookmark for TBufDataset.

## 5.8 TBufRecLinkItem

```
TBufRecLinkItem = record
  prior : PBufRecLinkItem;
  next : PBufRecLinkItem
  ;
end
```

Stores bookmarks to the previous and next records in a linked list

## 5.9 TDBCompareRec

```
TDBCompareRec = record
```

```

CompareFunc : TCompareFunc;
Off : PtrInt;
;
NullBOff : PtrInt;
FieldInd : LongInt;
Size : Integer;
Options
: TLocateOptions;
Desc : Boolean;
end

```

TDBCompareRec is a record type which implements a structure used to compare buffers for fields in TBufDataset. TDBCompareRec is used in the implementation of the TDBCompareStruct type.

## 5.10 TRecUpdateBuffer

```

TRecUpdateBuffer = record
  UpdateKind : TUpdateKind;
  BookmarkData
  : TBufBookmark;
  NextBookmarkData : TBufBookmark;
  OldValuesBuffer
  : TRecordBuffer;
end

```

Implements a buffer used for record updates in TBufDataset

## 5.11 TArrayBufIndex

### 5.11.1 Description

TArrayBufIndex is a TBufIndex descendant that implements an index using an array of record buffers. TArrayBufIndex uses an internal member with an array of pointers to the TRecordBuffer data type used for record buffers in the index. TArrayBufIndex provides support for Bookmarks in the dataset using TBufBookmark.

See also: TBufIndex ([145](#)), TRecordBuffer ([130](#)), TBufBookmark ([133](#))

**5.11.2 Method overview**

Page	Method	Description
<a href="#">138</a>	AddRecord	
<a href="#">138</a>	BeginUpdate	
<a href="#">137</a>	CanScrollForward	
<a href="#">135</a>	Create	
<a href="#">137</a>	DoScrollForward	
<a href="#">138</a>	EndUpdate	
<a href="#">135</a>	GetCurrent	
<a href="#">137</a>	GotoBookmark	
<a href="#">137</a>	InitialiseIndex	
<a href="#">137</a>	InitialiseSpareRecord	
<a href="#">138</a>	InsertRecordBeforeCurrentRecord	
<a href="#">138</a>	ReleaseSpareRecord	
<a href="#">138</a>	RemoveRecordFromIndex	
<a href="#">136</a>	RestoreCurrentRecord	
<a href="#">135</a>	ScrollBackward	
<a href="#">136</a>	ScrollFirst	
<a href="#">135</a>	ScrollForward	
<a href="#">136</a>	ScrollLast	
<a href="#">136</a>	SetToFirstRecord	
<a href="#">136</a>	SetToLastRecord	
<a href="#">137</a>	StoreCurrentRecIntoBookmark	
<a href="#">136</a>	StoreCurrentRecord	
<a href="#">137</a>	StoreSpareRecIntoBookmark	

**5.11.3 TArrayBufIndex.Create**

Synopsis:

Declaration: `constructor Create(const ADataset: TCustomBufDataset);` Override

Visibility: public

**5.11.4 TArrayBufIndex.ScrollBackward**

Synopsis:

Declaration: `function ScrollBackward : TGetResult;` Override

Visibility: public

**5.11.5 TArrayBufIndex.ScrollForward**

Synopsis:

Declaration: `function ScrollForward : TGetResult;` Override

Visibility: public

**5.11.6 TArrayBufIndex.GetCurrent**

Synopsis:



Declaration: function GetCurrent : TGetResult; Override

Visibility: public

### **5.11.7 TArrayBuflIndex.ScrollFirst**

Synopsis:

Declaration: function ScrollFirst : TGetResult; Override

Visibility: public

### **5.11.8 TArrayBuflIndex.ScrollLast**

Synopsis:

Declaration: procedure ScrollLast; Override

Visibility: public

### **5.11.9 TArrayBuflIndex.SetToFirstRecord**

Synopsis:

Declaration: procedure SetToFirstRecord; Override

Visibility: public

### **5.11.10 TArrayBuflIndex.SetToLastRecord**

Synopsis:

Declaration: procedure SetToLastRecord; Override

Visibility: public

### **5.11.11 TArrayBuflIndex.StoreCurrentRecord**

Synopsis:

Declaration: procedure StoreCurrentRecord; Override

Visibility: public

### **5.11.12 TArrayBuflIndex.RestoreCurrentRecord**

Synopsis:

Declaration: procedure RestoreCurrentRecord; Override

Visibility: public

### **5.11.13 TArrayBufIndex.CanScrollForward**

Synopsis:

Declaration: `function CanScrollForward : Boolean; Override`

Visibility: `public`

### **5.11.14 TArrayBufIndex.DoScrollForward**

Synopsis:

Declaration: `procedure DoScrollForward; Override`

Visibility: `public`

### **5.11.15 TArrayBufIndex.StoreCurrentRecIntoBookmark**

Synopsis:

Declaration: `procedure StoreCurrentRecIntoBookmark(const ABookmark: PBufBookmark)  
; Override`

Visibility: `public`

### **5.11.16 TArrayBufIndex.StoreSpareRecIntoBookmark**

Synopsis:

Declaration: `procedure StoreSpareRecIntoBookmark(const ABookmark: PBufBookmark)  
; Override`

Visibility: `public`

### **5.11.17 TArrayBufIndex.GotoBookmark**

Synopsis:

Declaration: `procedure GotoBookmark(const ABookmark: PBufBookmark); Override`

Visibility: `public`

### **5.11.18 TArrayBufIndex.InitialiseIndex**

Synopsis:

Declaration: `procedure InitialiseIndex; Override`

Visibility: `public`

### **5.11.19 TArrayBufIndex.InitialiseSpareRecord**

Synopsis:

Declaration: `procedure InitialiseSpareRecord(const ASpareRecord: TRecordBuffer)  
; Override`

Visibility: `public`

### **5.11.20 TArrayBufIndex.ReleaseSpareRecord**

Synopsis:

Declaration: `procedure ReleaseSpareRecord; Override`

Visibility: `public`

### **5.11.21 TArrayBufIndex.BeginUpdate**

Synopsis:

Declaration: `procedure BeginUpdate; Override`

Visibility: `public`

### **5.11.22 TArrayBufIndex.AddRecord**

Synopsis:

Declaration: `procedure AddRecord; Override`

Visibility: `public`

### **5.11.23 TArrayBufIndex.InsertRecordBeforeCurrentRecord**

Synopsis:

Declaration: `procedure InsertRecordBeforeCurrentRecord(const ARecord: TRecordBuffer)  
; Override`

Visibility: `public`

### **5.11.24 TArrayBufIndex.RemoveRecordFromIndex**

Synopsis:

Declaration: `procedure RemoveRecordFromIndex(const ABookmark: TBufBookmark)  
; Override`

Visibility: `public`

### **5.11.25 TArrayBufIndex.EndUpdate**

Synopsis:

Declaration: `procedure EndUpdate; Override`

Visibility: `public`

## 5.12 TBufBlobStream

### 5.12.1 Description

TBufBlobStream is a TStream descendant which implements a Blob Stream for TBufDataset. TBufBlobStream has members which represent the Dataset and Field for the Blob, its Buffer, and modification flags. TBufBlobStream provides overridden methods used to perform Read, Write, and Seek operations using the buffer allocated for a Blob field.

TBufBlobStream instances are created in the TBufDataset.CreateBlobStream method.

See also: TCustomBufDataset.CreateBlobStream ([160](#))

### 5.12.2 Method overview

Page	Method	Description
<a href="#">139</a>	Create	Constructor for the class instance
<a href="#">139</a>	Destroy	Destructor for the class instance

### 5.12.3 TBufBlobStream.Create

Synopsis: Constructor for the class instance

Declaration: constructor Create(Field: TBlobField; Mode: TBlobStreamMode)

Visibility: public

Description: Constructor for the class instance

### 5.12.4 TBufBlobStream.Destroy

Synopsis: Destructor for the class instance

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy is the destructor for the class instance.

## 5.13 TBufDataset

### 5.13.1 Description

TBufDataset is a TCustomBufDataset descendant that implements a database-aware buffered dataset. TBufDataset is an in-memory dataset that can be used as a local stand-alone dataset, or it can be used as a local buffer for updates applied to a remote dataset. TBufDataset implements many features similar to those in TClientDataset in Delphi. However, it is **not** meant to be code or function compatible with TClientDataset. It is designed to provide management of the buffers used to access record data, and is used as the ancestor for classes like TSQLQuery and TRESTBufDataset.

As a TDBDataset descendent, it offers access to many of the database features supported in the FCL DB package. As a TDataset descendent, it also offers access to familiar navigation and data handling methods in the ancestor class. TBufDataset has features that allow local storage and retrieval of field definitions and record data. In addition, facilities are provided that allow local indexing of record data

in the dataset. There is a comprehensive parser/expression evaluator available that allows complete support for Filters in the dataset.

TBufDataset sets the visibility for properties and methods defined in the ancestor class.

Additional information about using TBufDataset can be found on the Lazarus Wiki at: [How to write in-memory database applications in Lazarus/FPC \(TBufDataset\)](#).

See also: TCustomBufDataset ([153](#)), TDataset ([371](#)), TDBDataset ([415](#))

### 5.13.2 Property overview

Page	Properties	Access	Description
<a href="#">141</a>	Active		
<a href="#">141</a>	AfterCancel		
<a href="#">142</a>	AfterClose		
<a href="#">142</a>	AfterDelete		
<a href="#">142</a>	AfterEdit		
<a href="#">142</a>	AfterInsert		
<a href="#">142</a>	AfterOpen		
<a href="#">142</a>	AfterPost		
<a href="#">142</a>	AfterScroll		
<a href="#">141</a>	AutoCalcFields		
<a href="#">143</a>	BeforeCancel		
<a href="#">143</a>	BeforeClose		
<a href="#">143</a>	BeforeDelete		
<a href="#">143</a>	BeforeEdit		
<a href="#">143</a>	BeforeInsert		
<a href="#">143</a>	BeforeOpen		
<a href="#">143</a>	BeforePost		
<a href="#">144</a>	BeforeScroll		
<a href="#">140</a>	FieldDefs		
<a href="#">141</a>	Filter		
<a href="#">141</a>	Filtered		
<a href="#">140</a>	MaxIndexesCount		
<a href="#">144</a>	OnCalcFields		
<a href="#">144</a>	OnDeleteError		
<a href="#">144</a>	OnEditError		
<a href="#">144</a>	OnFilterRecord		
<a href="#">144</a>	OnNewRecord		
<a href="#">144</a>	OnPostError		
<a href="#">141</a>	ReadOnly		

### 5.13.3 TBufDataset.MaxIndexesCount

Declaration: Property MaxIndexesCount :

Visibility: published

Access:

### 5.13.4 TBufDataset.FieldDefs

Declaration: Property FieldDefs :

Visibility: published

Access:

### **5.13.5 TBufDataset.Active**

Declaration: Property Active :

Visibility: published

Access:

### **5.13.6 TBufDataset.AutoCalcFields**

Declaration: Property AutoCalcFields :

Visibility: published

Access:

### **5.13.7 TBufDataset.Filter**

Declaration: Property Filter :

Visibility: published

Access:

### **5.13.8 TBufDataset.Filtered**

Declaration: Property Filtered :

Visibility: published

Access:

### **5.13.9 TBufDataset.ReadOnly**

Declaration: Property ReadOnly :

Visibility: published

Access:

### **5.13.10 TBufDataset.AfterCancel**

Declaration: Property AfterCancel :

Visibility: published

Access:

#### **5.13.11 TBufDataset.AfterClose**

Declaration: `Property AfterClose :`

Visibility: published

Access:

#### **5.13.12 TBufDataset.AfterDelete**

Declaration: `Property AfterDelete :`

Visibility: published

Access:

#### **5.13.13 TBufDataset.AfterEdit**

Declaration: `Property AfterEdit :`

Visibility: published

Access:

#### **5.13.14 TBufDataset.AfterInsert**

Declaration: `Property AfterInsert :`

Visibility: published

Access:

#### **5.13.15 TBufDataset.AfterOpen**

Declaration: `Property AfterOpen :`

Visibility: published

Access:

#### **5.13.16 TBufDataset.AfterPost**

Declaration: `Property AfterPost :`

Visibility: published

Access:

#### **5.13.17 TBufDataset.AfterScroll**

Declaration: `Property AfterScroll :`

Visibility: published

Access:

#### **5.13.18 TBufDataset.BeforeCancel**

Declaration: `Property BeforeCancel :`

Visibility: published

Access:

#### **5.13.19 TBufDataset.BeforeClose**

Declaration: `Property BeforeClose :`

Visibility: published

Access:

#### **5.13.20 TBufDataset.BeforeDelete**

Declaration: `Property BeforeDelete :`

Visibility: published

Access:

#### **5.13.21 TBufDataset.BeforeEdit**

Declaration: `Property BeforeEdit :`

Visibility: published

Access:

#### **5.13.22 TBufDataset.BeforeInsert**

Declaration: `Property BeforeInsert :`

Visibility: published

Access:

#### **5.13.23 TBufDataset.BeforeOpen**

Declaration: `Property BeforeOpen :`

Visibility: published

Access:

#### **5.13.24 TBufDataset.BeforePost**

Declaration: `Property BeforePost :`

Visibility: published

Access:



### 5.13.25 TBufDataset.BeforeScroll

Declaration: Property BeforeScroll :

Visibility: published

Access:

### 5.13.26 TBufDataset.OnCalcFields

Declaration: Property OnCalcFields :

Visibility: published

Access:

### 5.13.27 TBufDataset.OnDeleteError

Declaration: Property OnDeleteError :

Visibility: published

Access:

### 5.13.28 TBufDataset.OnEditError

Declaration: Property OnEditError :

Visibility: published

Access:

### 5.13.29 TBufDataset.OnFilterRecord

Declaration: Property OnFilterRecord :

Visibility: published

Access:

### 5.13.30 TBufDataset.OnNewRecord

Declaration: Property OnNewRecord :

Visibility: published

Access:

### 5.13.31 TBufDataset.OnPostError

Declaration: Property OnPostError :

Visibility: published

Access:

## 5.14 TBufIndex

### 5.14.1 Description

TBufIndex is a TObject descendant which defines the interface used to implement indexes in TBufDataset. TBufIndex provides access to the Dataset with field values for the index, and methods to perform record navigation and index maintenance.

TBufIndex contains many virtual and abstract methods that should be implemented in a descendent class which provides a specific index implementation, such as: TDoubleLinkedBufIndex, TUniDirectionalBufIndex, and TArrayBufIndex.

TBufIndex is the type used for the TBufDataset.BufferIndex property.

See also: TBufDataset.BufferIndex ([139](#)), TDoubleLinkedBufIndex ([173](#)), TUniDirectionalBufIndex ([181](#)), TArrayBufIndex ([134](#))

**5.14.2 Method overview**

Page	Method	Description
151	AddRecord	Adds a record to the index
150	BeginUpdate	
149	BookmarkValid	Indicates the specified Bookmark is valid
148	CanScrollForward	Indicates if the index can be scrolled forward
149	CompareBookmarks	Gets the relative order for the specified Bookmarks
147	Create	Constructor for the class instance
149	DoScrollForward	Implements actions need to scroll forward in the index
151	EndUpdate	
147	GetCurrent	Gets the record buffer for the current record in the dataset
148	GetRecord	Gets prior/next record relative to the specified bookmark
149	GotoBookmark	Moves the index position to the specified Bookmark
150	InitialiseIndex	Initializes the index
150	InitialiseSpareRecord	Initializes values in the spare record for the index
151	InsertRecordBeforeCurrentRecord	Inserts a record before the current record in the index order
151	OrderCurrentRecord	
150	ReleaseSpareRecord	Releases resources allocated to the spare record for the index
151	RemoveRecordFromIndex	Remove the record at the specified bookmark from the index
148	RestoreCurrentRecord	Restores the stored record buiffer to the current record
150	SameBookmarks	Indicates if the specified Bookmarks are for the same record
147	ScrollBackward	Moves to the prior record in the index
147	ScrollFirst	Moves to the first record in the index
147	ScrollForward	Moves to the next record in the index
147	ScrollLast	Moves to the last record in the index
148	SetToFirstRecord	Sets the index to the first record in the index order
148	SetToLastRecord	Sets the index to the last record in the index order
149	StoreCurrentRecIntoBookmark	
148	StoreCurrentRecord	Stores the record buffer for the current record
149	StoreSpareRecIntoBookmark	

**5.14.3 Property overview**

Page	Properties	Access	Description
153	BookmarkSize	r	Size (number of bytes) needed for Bookmarks in the index
152	CurrentBuffer	r	Current record buffer in the index
152	CurrentRecord	r	Current record in the index
152	IsInitialized	r	Indicates if the index has been initialized
153	RecNo	rw	Active record number in the index
152	SpareBuffer	r	Spare record buffer for the index
152	SpareRecord	r	Spare record for the index

#### 5.14.4 TBufIndex.Create

Synopsis: Constructor for the class instance

Declaration: `constructor Create(const ADataset: TCustomBufDataset); Virtual`

Visibility: public

Description: `Create` is the constructor for the class instance. `Create` calls the inherited constructor, and stores the `ADataset` parameter to the internal member used for the `TCustomBufDataset` class instance.

#### 5.14.5 TBufIndex.ScrollBackward

Synopsis: Moves to the prior record in the index

Declaration: `function ScrollBackward : TGetResult; Virtual; Abstract`

Visibility: public

Description: `ScrollBackward` is a `TGetResult` function used to scrolling to the previous record in the index order. `ScrollBackward` implements the behavior needed to support the `TDataset.MoveBy` method using the index order for the dataset.

`ScrollBackward` is an abstract virtual method method, and must be implemented in a descendent class. The return value is a `TGetResult` enumeration value that indicates the result for the scroll request.

See also: `TGetResult` ([326](#)), `TDataset.MoveBy` ([388](#))

#### 5.14.6 TBufIndex.ScrollForward

Synopsis: Moves to the next record in the index

Declaration: `function ScrollForward : TGetResult; Virtual; Abstract`

Visibility: public

#### 5.14.7 TBufIndex.GetCurrent

Synopsis: Gets the record buffer for the current record in the dataset

Declaration: `function GetCurrent : TGetResult; Virtual; Abstract`

Visibility: public

#### 5.14.8 TBufIndex.ScrollFirst

Synopsis: Moves to the first record in the index

Declaration: `function ScrollFirst : TGetResult; Virtual; Abstract`

Visibility: public

#### 5.14.9 TBufIndex.ScrollLast

Synopsis: Moves to the last record in the index

Declaration: `procedure ScrollLast; Virtual; Abstract`

Visibility: public

#### 5.14.10 TBufIndex.GetRecord

Synopsis: Gets prior/next record relative to the specified bookmark

Declaration: `function GetRecord(ABookmark: PBufBookmark; GetMode: TGetMode)  
: TGetResult; Virtual`

Visibility: public

Description: `GetRecord` is used to get the prior/next record relative to the specified bookmark. `GetRecord` is a virtual method that should be overridden in a descendent class; the implementation in `TBufIndex` simply returns the value `grError` as the return value for the method.

Please note that `GetRecord` should **not** change the current record in the dataset on exit.

#### 5.14.11 TBufIndex.SetToFirstRecord

Synopsis: Sets the index to the first record in the index order

Declaration: `procedure SetToFirstRecord; Virtual; Abstract`

Visibility: public

#### 5.14.12 TBufIndex.SetToLastRecord

Synopsis: Sets the index to the last record in the index order

Declaration: `procedure SetToLastRecord; Virtual; Abstract`

Visibility: public

#### 5.14.13 TBufIndex.StoreCurrentRecord

Synopsis: Stores the record buffer for the current record

Declaration: `procedure StoreCurrentRecord; Virtual; Abstract`

Visibility: public

#### 5.14.14 TBufIndex.RestoreCurrentRecord

Synopsis: Restores the stored record buiffer to the current record

Declaration: `procedure RestoreCurrentRecord; Virtual; Abstract`

Visibility: public

#### 5.14.15 TBufIndex.CanScrollForward

Synopsis: Indicates if the index can be scrolled forward

Declaration: `function CanScrollForward : Boolean; Virtual; Abstract`

Visibility: public

#### 5.14.16 TBufIndex.DoScrollForward

Synopsis: Implements actions need to scroll forward in the index

Declaration: `procedure DoScrollForward; Virtual; Abstract`

Visibility: public

#### 5.14.17 TBufIndex.StoreCurrentRecIntoBookmark

Synopsis:

Declaration: `procedure StoreCurrentRecIntoBookmark(const ABookmark: PBufBookmark)  
; Virtual; Abstract`

Visibility: public

#### 5.14.18 TBufIndex.StoreSpareRecIntoBookmark

Synopsis:

Declaration: `procedure StoreSpareRecIntoBookmark(const ABookmark: PBufBookmark)  
; Virtual; Abstract`

Visibility: public

#### 5.14.19 TBufIndex.GotoBookmark

Synopsis: Moves the index position to the specified Bookmark

Declaration: `procedure GotoBookmark(const ABookmark: PBufBookmark); Virtual  
; Abstract`

Visibility: public

#### 5.14.20 TBufIndex.BookmarkValid

Synopsis: Indicates the specified Bookmark is valid

Declaration: `function BookmarkValid(const ABookmark: PBufBookmark) : Boolean  
; Virtual`

Visibility: public

#### 5.14.21 TBufIndex.CompareBookmarks

Synopsis: Gets the relative order for the specified Bookmarks

Declaration: `function CompareBookmarks(const ABookmark1: PBufBookmark;  
const ABookmark2: PBufBookmark) : Integer  
; Virtual`

Visibility: public

#### 5.14.22 TBufIndex.SameBookmarks

Synopsis: Indicates if the specified Bookmarks are for the same record

Declaration: `function SameBookmarks(const ABookmark1: PBufBookmark;  
const ABookmark2: PBufBookmark) : Boolean  
; Virtual`

Visibility: public

#### 5.14.23 TBufIndex.InitialiseIndex

Synopsis: Initializes the index

Declaration: `procedure InitialiseIndex; Virtual; Abstract`

Visibility: public

Description: Initializes the index

#### 5.14.24 TBufIndex.InitialiseSpareRecord

Synopsis: Initializes values in the spare record for the index

Declaration: `procedure InitialiseSpareRecord(const ASpareRecord: TRecordBuffer)  
; Virtual; Abstract`

Visibility: public

Description: Initializes values in the spare record for the index

#### 5.14.25 TBufIndex.ReleaseSpareRecord

Synopsis: Releases resources allocated to the spare record for the index

Declaration: `procedure ReleaseSpareRecord; Virtual; Abstract`

Visibility: public

Description: Releases resources allocated to the spare record for the index

#### 5.14.26 TBufIndex.BeginUpdate

Synopsis:

Declaration: `procedure BeginUpdate; Virtual; Abstract`

Visibility: public

Description:

#### 5.14.27 TBufIndex.AddRecord

Synopsis: Adds a record to the index

Declaration: `procedure AddRecord; Virtual; Abstract`

Visibility: public

Description: Adds a record to the end of the index as the new last record (spare record). AddRecord is used in the GetNextPacket method.

#### 5.14.28 TBufIndex.InsertRecordBeforeCurrentRecord

Synopsis: Inserts a record before the current record in the index order

Declaration: `procedure InsertRecordBeforeCurrentRecord(const ARecord: TRecordBuffer)  
; Virtual; Abstract`

Visibility: public

Description: Inserts a record before the current record using the sort order for the active index.

#### 5.14.29 TBufIndex.RemoveRecordFromIndex

Synopsis: Remove the record at the specified bookmark from the index

Declaration: `procedure RemoveRecordFromIndex(const ABookmark: TBufBookmark); Virtual  
; Abstract`

Visibility: public

#### 5.14.30 TBufIndex.OrderCurrentRecord

Synopsis:

Declaration: `procedure OrderCurrentRecord; Virtual; Abstract`

Visibility: public

Description:

#### 5.14.31 TBufIndex.EndUpdate

Synopsis:

Declaration: `procedure EndUpdate; Virtual; Abstract`

Visibility: public

Description:



### **5.14.32 TBufIndex.SpareRecord**

Synopsis: Spare record for the index

Declaration: `Property SpareRecord : TRecordBuffer`

Visibility: public

Access: Read

Description: Spare record for the index

### **5.14.33 TBufIndex.SpareBuffer**

Synopsis: Spare record buffer for the index

Declaration: `Property SpareBuffer : TRecordBuffer`

Visibility: public

Access: Read

Description: Spare record buffer for the index

### **5.14.34 TBufIndex.CurrentRecord**

Synopsis: Current record in the index

Declaration: `Property CurrentRecord : TRecordBuffer`

Visibility: public

Access: Read

Description: Current record in the index

### **5.14.35 TBufIndex.CurrentBuffer**

Synopsis: Current record buffer in the index

Declaration: `Property CurrentBuffer : Pointer`

Visibility: public

Access: Read

Description: Current record buffer in the index

### **5.14.36 TBufIndex.IsInitialized**

Synopsis: Indicates if the index has been initialized

Declaration: `Property IsInitialized : Boolean`

Visibility: public

Access: Read

Description: Indicates if the index has been initialized

### 5.14.37 TBufIndex.BookmarkSize

Synopsis: Size (number of bytes) needed for Bookmarks in the index

Declaration: `Property BookmarkSize : Integer`

Visibility: `public`

Access: `Read`

Description: Size (number of bytes) needed for Bookmarks in the index

### 5.14.38 TBufIndex.RecNo

Synopsis: Active record number in the index

Declaration: `Property RecNo : LongInt`

Visibility: `public`

Access: `Read,Write`

Description: Active record number in the index

## 5.15 TCustomBufDataset

### 5.15.1 Description

`TCustomBufDataset` is a `TDBDataset` descendant that implements the ancestor class for a database-aware buffered dataset. `TCustomBufDataset` is an in-memory dataset that can be used as a local stand-alone dataset, or it can be used as a local buffer for updates applied to a remote dataset. `TCustomBufDataset` implements many features similar to those in `TClientDataset` in Delphi. However, it is **not** meant to be code or function compatible with `TClientDataset`. It is designed to provide management of the buffers used to access record data, and is used as the ancestor for classes like `TSQLQuery` and `TRESTBufDataset`.

As a `TDBDataset` descendant, it offers access to many of the database features supported in the FCL DB package. As a `TDataset` descendant, it also offers access to familiar navigation and data handling methods in the ancestor class. `TCustomBufDataset` has features that allow local storage and retrieval of field definitions and record data. In addition, facilities are provided that allow local indexing of record data in the dataset. There is a comprehensive parser/expression evaluator available that allows complete support for Filters in the dataset.

Do not create instances of `TCustomBufDataset`. Use the `TBufDataset` descendant instead.

Additional information about using `TBufDataset` can be found on the Lazarus Wiki at: [How to write in-memory database applications in Lazarus/FPC \(TBufDataset\)](#).

See also: `TBufDataset` ([139](#)), `TDBDataset` ([415](#)), `TDataset` ([371](#))

### 5.15.2 Method overview

Page	Method	Description
160	AddIndex	Adds an index definition to the dataset
156	ApplyUpdates	Applies pending updates to the dataset
165	BookmarkValid	Determines if the specified Bookmark is valid for the dataset
157	CancelUpdates	Cancels pending updates in the dataset
165	Clear	Clears the content in the dataset
161	ClearIndexes	Clears index storage in the dataset
165	CompareBookmarks	Gets the relative sort order for the specified Bookmarks
166	CopyFromDataset	Loads field definitions and optional data from another dataset
155	Create	Constructor for the class instance
160	CreateBlobStream	Creates a Blob stream for the specified field with the given permissions
164	CreateDataset	Creates the dataset using its field definitions or bound fields
157	Destroy	Destructor for the class instance
161	GetDatasetPacket	Builds a data packet representing the content in the buffered dataset
156	GetFieldData	
163	LoadFromFile	Loads the dataset from the specified file using the given format
162	LoadFromStream	Loads the dataset from the specified stream using the given data format
158	Locate	Locates the first record with fields having the specified values
158	Lookup	Gets values from the first record with fields that match the search condition
157	MergeChangeLog	Frees update buffers and Blob update buffer allocated for the dataset
157	RevertRecord	Reverts the current record to its original (un-edited) values
163	SaveToFile	Saves the dataset to the specified file using the given data format
162	SaveToStream	Saves the dataset to the specified stream using the given data format
161	SetDatasetPacket	Retrieves and applies the data packet with the content for the dataset
156	SetFieldData	
160	UpdateStatus	Gets the update status for the current record in the dataset

### 5.15.3 Property overview

Page	Properties	Access	Description
<a href="#">166</a>	ChangeCount	r	Number of pending changes for the dataset
<a href="#">167</a>	FileName	rw	File name on the local file system used to load or store the dataset
<a href="#">169</a>	IndexDefs	r	Index definitions for the dataset
<a href="#">170</a>	IndexFieldNames	rw	Field names included in the custom index
<a href="#">169</a>	IndexName	rw	Name of the selected index for the dataset
<a href="#">167</a>	ManualMergeChangeLog	rw	Indicates if the update change log can be manually merged
<a href="#">166</a>	MaxIndexesCount	rw	Maximum number of indexes available in the dataset
<a href="#">168</a>	OnUpdateError	rw	Event handler signalled when an error occurs while updating records
<a href="#">168</a>	PacketRecords	rw	Number of records allowed in a data packet handled by the packet reader
<a href="#">167</a>	ReadOnly	rw	Indicates if records can be added, deleted, or modified in the dataset
<a href="#">170</a>	UniDirectional	rw	Indicates if the dataset is for uni-directional navigation only

### 5.15.4 TCustomBufDataset.Create

Synopsis: Constructor for the class instance

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` is an overridden constructor for the class instance. `Create` calls the inherited constructor using the value in `AOwner` as the owner of the class instance. `Create` sets the default value for internal members used in the implementation of the buffered dataset, including:

- Parser for data packets (`Nil`)
- Manual MergeChangeLog handling (**False**)
- Default AutoInc field value (**-1**)
- Update Buffer count (**0**)
- Blob Buffer count (**0**)
- Blob Update Buffer count (**0**)

`Creates` sets the value for public and published properties, including:

- MaxIndexesCount (**2**)
- PacketRecords (**10**)

`Create` allocates resources needed for the `BufIndexDefs` and `BufIndexes` properties.

See also: `TCustomBufDataset.BufIndexDefs` ([153](#)), `TCustomBufDataset.BufIndexes` ([153](#)), `TCustomBufDataset.MaxIndexesCount` ([166](#)), `TCustomBufDataset.PacketRecords` ([168](#))

### 5.15.5 TCustomBufDataset.GetFieldData

Synopsis:

```
Declaration: function GetFieldData(Field: TField; Buffer: Pointer;
                                NativeFormat: Boolean) : Boolean; Override
            function GetFieldData(Field: TField; Buffer: Pointer) : Boolean
                                ; Override
```

Visibility: public

Description:

### 5.15.6 TCustomBufDataset.SetFieldData

Synopsis:

```
Declaration: procedure SetFieldData(Field: TField; Buffer: Pointer;
                                NativeFormat: Boolean); Override
            procedure SetFieldData(Field: TField; Buffer: Pointer); Override
```

Visibility: public

Description:

### 5.15.7 TCustomBufDataset.ApplyUpdates

Synopsis: Applies pending updates to the dataset

```
Declaration: procedure ApplyUpdates; Virtual; Overload
            procedure ApplyUpdates(MaxErrors: Integer); Virtual; Overload
```

Visibility: public

Description: `ApplyUpdates` is an overloaded procedure used to apply pending updates for the dataset.

`ApplyUpdates` calls `CheckBrowseMode` and gets a bookmark for the current record in the dataset. The bookmark is used to return the dataset its original record after updates are applied and the dataset has been refreshed.

`MaxErrors` is the threshold where errors encountered in the process cause it to be aborted. The value 0 (zero) indicates no errors are allowed during the apply updates process.

`ApplyUpdates` uses the internal update buffers allocated for the dataset to perform the actions required in the method. The `ApplyRecUpdate` method is called for each of the update buffers. An exception raised in `ApplyRecUpdate` is handled in the method. When the value in `MaxErrors` is exceeded, the process is aborted. Otherwise, the update buffer is skipped. If the `OnUpdateError` exception handler has been assigned, it is signalled using the required arguments and its `Response` value is captured in the method. If the process is to be aborted, an exception is raised to indicate the condition.

When an update buffer is successfully applied, resources allocated to its `OldValuesBuffer`, `Bookmark` data, and the update buffer itself are freed. If all pending updates are applied successfully, and `ManualMergeChangeLog` contains `False`, the `MergeChangeLog` method is called.

`ApplyUpdates` calls `Resync` to fresh the records displayed in the dataset prior to exiting from the method.

See also: `TDataset.CheckBrowseMode` ([378](#)), `TRecUpdateBuffer` ([134](#)), `TCustomBufDataset.ApplyRecUpdate` ([153](#)), `TCustomBufDataset.OnUpdateError` ([168](#)), `TCustomBufDataset.ManualMergeChangeLog` ([167](#)), `TCustomBufDataset.MergeChangeLog` ([157](#)), `TDataset.Resync` ([390](#))

### 5.15.8 TCustomBufDataset.MergeChangeLog

Synopsis: Frees update buffers and Blob update buffer allocated for the dataset

Declaration: `procedure MergeChangeLog`

Visibility: `public`

Description: `MergeChangeLog` is a procedure used to free update buffers and Blob update buffer allocated for the dataset. `MergeChangeLog` is called from the `ApplyUpdates` method.

See also: `TCustomBufDataset.ApplyUpdates` ([156](#))

### 5.15.9 TCustomBufDataset.RevertRecord

Synopsis: Reverts the current record to its original (un-edited) values

Declaration: `procedure RevertRecord`

Visibility: `public`

Description: `RevertRecord` is a procedure used to revert changes in the current record to their original (un-edited) values. `RevertRecord` calls the `CheckBrowseMode` method to ensure that the dataset is Active, and to perform event notifications for a change in dataset state.

`RevertRecord` checks for an update buffer in the current record, and when found calls the `CancelRecordUpdateBuffer` method and removes the update buffer. The `Resync` method is called to refresh the records in the dataset.

See also: `TDataset.CheckBrowseMode` ([378](#)), `TDataset.Active` ([400](#)), `TCustomBufDataset.CancelRecordUpdateBuffer` ([153](#)), `TCustomBufDataset.Resync` ([153](#))

### 5.15.10 TCustomBufDataset.CancelUpdates

Synopsis: Cancels pending updates in the dataset

Declaration: `procedure CancelUpdates; Virtual`

Visibility: `public`

Description: `CancelUpdates` is a procedure used to cancel pending updates to records in the dataset. `CancelUpdates` calls the `CheckBrowseMode` method to ensure that the dataset is Active, and to perform event notifications for a change in dataset state.

`CancelUpdates` checks for existing update buffers allocated in the dataset. When update buffers exist, the `CancelRecordUpdateBuffer` method is called for the bookmark in each of the update buffers.

`CancelUpdates` restores the record position when able, and calls `Resync` to refresh the records available in the dataset.

See also: `TDataset.CheckBrowseMode` ([378](#)), `TDataset.Active` ([400](#)), `TCustomBufDataset.CancelRecordUpdateBuffer` ([153](#)), `TCustomBufDataset.Resync` ([153](#))

### 5.15.11 TCustomBufDataset.Destroy

Synopsis: Destructor for the class instance

Declaration: `destructor Destroy; Override`

Visibility: `public`

**Description:** `Destroy` is the overridden destructor for the class instance. `Destroy` calls the `Close` method if the dataset contains `True` in the `Active` property.

`Destroy` frees resources allocated to the internal buffers for records, and `Blob` fields. The `ClearIndexes` method is called to free resources allocated for index storage mechanisms. `Destroy` calls the inherited destructor.

### 5.15.12 TCustomBufDataset.Locate

**Synopsis:** Locates the first record with fields having the specified values

**Declaration:** `function Locate(const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions) : Boolean; Override`

**Visibility:** `public`

**Description:** `Locate` is an overridden `Boolean` function used to locate the first record with fields that match the specified search values. `Locate` implements the method defined in the ancestor class.

`KeyFields` is a delimited list of field names searched in the method. The delimiter character is `'` (Semicolon).

`KeyValues` is a `Variant` type with the field values required for the specified `KeyFields`. When `KeyFields` contains a single field name, `KeyValues` can be represented using the same data type as the field value. When `KeyFields` contains more than one field name, it is a `Variant` array.

`Options` is a `TLocateOptions` parameter that indicates the locate options enabled in the method. Options can contain zero (`0`) or more values from the `TLocateOption` enumeration.

`Locate` calls the inherited method to ensure that the dataset supports bi-directional navigation. An `Exception` is raised if the `UniDirectional` property contains `True`. The value in the `Active` property is checked to ensure that the dataset has been opened. An `Exception` is raised if `Active` contains `False`. No actions are performed in the method when `IsEmpty` returns `True`.

`Locate` initializes an internal `TDBCompareStruct` instance that is used when searching field values using the `SearchFields` and `Options` arguments. The search is performed by using the `Filter` feature; the filter fields are set to the values in `KeyValues`. Each record in the dataset is visited and the `OnFilterRecord` event is signalled to determine if the record meets the `Filter` criteria. The search ends when a record is found that matches the search values.

The record position is changed to the marching record, or positioned at the end-of-file when no matching record is found.

The return value is `True` if a record was located that matched the search values.

**See also:** `TDataset.Locate` (387), `TCustomBufDataset.UniDirectional` (170), `TCustomBufDataset.Active` (153), `TCustomBufDataset.Filter` (153), `TCustomBufDataset.OnFilterRecord` (153), `TLocateOptions` (327), `TDBCompareStruct` (131)

### 5.15.13 TCustomBufDataset.Lookup

**Synopsis:** Gets values from the first record with fields that match the search condition

**Declaration:** `function Lookup(const KeyFields: string; const KeyValues: Variant; const ResultFields: string) : Variant; Override`

**Visibility:** `public`

**Description:** `Lookup` is an overridden `Variant` function used to get values from the first record with fields that match the specified search values. `Lookup` re-implements the method defined in the ancestor class. `Lookup` allows searching one or more fields for corresponding values, and returning a set of field values from the matching record.

`KeyFields` is a delimited list of field names searched in the method. The delimiter character is ';' (**Semicolon**). The following example would represent the fields **LASTNAME** and **FIRSTNAME**:

```
' LASTNAME; FIRSTNAME'
```

`KeyValues` is a `Variant` type with the field values required for the specified `KeyFields`. When `KeyFields` contains a single field name, `KeyValues` can be represented using the same data type as the field value. When `KeyFields` contains more than one field name, it is a `Variant` array. Using the previous values for `KeyFields`, the following would represent the values for **'JOHN SMITH'**:

```
VarArrayOf([ ' SMITH' , ' JOHN' ])
```

`ResultFields` contains one or more field names which determine the field values to include in the return value for the method. Like `KeyFields`, it is a delimited list of field names using ';' (**Semicolon**) as the delimiter character.

The return value is a `Variant` type that contains a value for each of the field names specified in the `ResultFields` argument. When a matching record is located, the return value is a `Variant` array. The return value is `Null` if a record is not found matching the specified `KeyValues` in its `KeyFields`.

```
var
  AVarValues: Variant;
  ABufDataset: TBufDataset;

// ...

AVarValues := ABufDataset.Lookup('lastname;firstname',
  VarArrayOf([' SMITH' , ' JOHN' ]), 'lastname;firstname;birthdate;emailaddress');

if not VarIsNull(AVarValues) then
  DoSomethingWith(AVarValues)
else
  WriteLn('No records match the search criteria');
```

`Lookup` does not change the record position in the dataset. A temporary `TBookmark` is used to record the current record in the dataset. The record position is restored prior to exiting from the method.

`Lookup` calls the `Locate` method to search records in the dataset using the `KeyFields` and `KeyValues` arguments. The `Locate` method may raise an `Exception` if the dataset is configured for `UniDirectional` access or when it is not `Active`.

See also: `TCustomBufDataset.Locate` ([158](#)), `TCustomBufDataset.Active` ([153](#)), `TCustomBufDataset.UniDirectional` ([170](#))



### 5.15.14 TCustomBufDataset.UpdateStatus

**Synopsis:** Gets the update status for the current record in the dataset

**Declaration:** `function UpdateStatus : TUpdateStatus; Override`

**Visibility:** public

**Description:** `UpdateStatus` is an overridden `TUpdateStatus` function used to get the update status for the current record in the dataset. `UpdateStatus` re-implements the method defined in the ancestor class. The return value is a member of the `TUpdateStatus` enumeration, and includes the following values:

**usUnmodifiedRecord** has not been modified

**usModifiedRecord** has been modified

**usInsertedRecords** was appended or inserted

**usDeletedRecord** has been deleted

`UpdateStatus` calls `GetActiveRecordUpdateBuffer` to get the update buffer for the `ActiveRecord`. When an update buffer exists for the record, its `UpdateKind` property is used as the return value for the method. The return value is `usUnmodified` when an update buffer does not exist for `ActiveRecord`.

See also: `TUpdateStatus` (130), `TCustomBufDataset.ActiveRecord` (153)

### 5.15.15 TCustomBufDataset.CreateBlobStream

**Synopsis:** Creates a Blob stream for the specified field with the given permissions

**Declaration:** `function CreateBlobStream(Field: TField; Mode: TBlobStreamMode)  
: TStream; Override`

**Visibility:** public

**Description:** Creates a Blob stream for the specified field with the given permissions

See also: `TDataset.CreateBlobStream` (380), `TField` (421), `TBlobStreamMode` (318)

### 5.15.16 TCustomBufDataset.AddIndex

**Synopsis:** Adds an index definition to the dataset

**Declaration:** `procedure AddIndex(const AName: string; const AFields: string;  
AOptions: TIndexOptions; const ADescFields: string;  
const ACaseInsFields: string); Virtual`

**Visibility:** public

**Description:** `AddIndex` is a procedure used to add an index definition to the dataset. `AddIndex` calls `CheckBiDirectional` to ensure that the `UniDirectional` property contains `False`. An exception is raised if `UniDirectional` contains `True`.

`AddIndex` checks for a valid value in the `AFields` argument. An Exception is raised when `AFields` is an empty string ("). In addition, an index added to an Active dataset cannot cause the index count to exceed the value in `MaxIndexesCount`. An Exception is raised if the index count would be larger than the value in `MaxIndexesCount`.

`AddIndex` creates a `TBufDatasetIndex` instance by calling `InternalAddIndex` using the arguments passed to the method. The new index is temporary when the `Active` property contains `True`; it will be discarded when the dataset is closed.

See also: `TCustomBufDataset.UniDirectional` ([170](#)), `TDataset.Active` ([400](#)), `TCustomBufDataset.InternalAddIndex` ([153](#)), `TBufDatasetIndex` ([130](#))

### 5.15.17 TCustomBufDataset.ClearIndexes

Synopsis: Clears index storage in the dataset

Declaration: `procedure ClearIndexes`

Visibility: `public`

Description: `ClearIndexes` is a procedure used to clear the storage for all indexes in the dataset. `ClearIndexes` uses the index definitions in the `BufIndexDefs` property and calls the `ClearIndex` method for each of the indexes.

`ClearIndexes` calls `CheckInactive` to ensure that the dataset has its `Active` property set to `False`. Indexes cannot be cleared if the dataset has been opened.

`ClearIndexes` is used when the value in the `UniDirectional` property is changed to `False`, and in the destructor for the class instance.

See also: `TCustomBufDataset.BufIndexDefs` ([153](#)), `TCustomBufDataset.CheckInactive` ([153](#)), `TCustomBufDataset.UniDirectional` ([170](#))

### 5.15.18 TCustomBufDataset.SetDatasetPacket

Synopsis: Retrieves and applies the data packet with the content for the dataset

Declaration: `procedure SetDatasetPacket (AReader: TDataPacketReader)`

Visibility: `public`

Description: `SetDatasetPacket` is a procedure used to retrieve and apply the data packet with the content for the dataset.

`AReader` is a `TDataPacketReader` instance that is used to read and apply the content for the buffered dataset. It is temporarily assigned to an internal member in the calls instance.

`SetDatasetPacket` calls the `Open` method to load the field definitions and record data for the dataset.

`SetDatasetPacket` is used in the implementation of the `LoadFromStream` method.

See also: `TDataset.Open` ([389](#)), `TCustomBufDataset.LoadFromStream` ([162](#)), `TDataPacketReader` ([171](#))

### 5.15.19 TCustomBufDataset.GetDatasetPacket

Synopsis: Builds a data packet representing the content in the buffered dataset

Declaration: `procedure GetDatasetPacket (AWriter: TDataPacketReader)`

Visibility: `public`

Description: `GetDatasetPacket` is a procedure used to build the data packet that represents the content in the buffered dataset. `GetDatasetPacket` ensures that record buffers and update buffers in the dataset are normalized and applied using the data packet handler (n.b. I won't call it a reader if it writes) in `AWriter`.

`GetDatasetPacket` sequentially processes records using the currently selected index for the dataset. Update buffers for a given record are consolidate in the record buffer, and the record buffer to stored

to the data packet handler. The current record position is not changed when the process has been completed.

GetDatasetPacket stores the current value for an auto-increment field used in the dataset in the AWriter argument. The internal data packet handler for the dataset is set to Nil when the method is completed.

GetDatasetPacket is used in the implementation of the SaveToStream method.

See also: TDataPacketReader ([171](#)), TCustomBufDataset.SaveToStream ([162](#))

### 5.15.20 TCustomBufDataset.LoadFromStream

Synopsis: Loads the dataset from the specified stream using the given data format

Declaration: `procedure LoadFromStream(AStream: TStream; Format: TDataPacketFormat)`

Visibility: public

Description: LoadFromStream is a procedure used to load field definitions and record data for the dataset from the specified stream. LoadFromStream is an overloaded variant of the method defined in the ancestor class and includes a parameter for the desired data packet format.

AStream is a TStream descendant with the values to load in the dataset. Format indicates the expected format for data packets in the stream. It is a variable argument that can be modified when a packet reader is located for the stream.

LoadFromStream calls CheckBiDirectional to ensure that the UniDirectional property in the dataset is set to False. An exception is raised when UniDirectional contains True. Index creation requires navigating in a non-default record order that would not be supported in a uni-directional dataset.

LoadFromStream creates a temporary TDataPacketReader class instance using the value specified in the Format argument. Format is a variable argument and can be modified when SetDatasetPacket is used to examine the packet reader and its stream. Calling SetDatasetPacket sets in motion a relatively complicated sequence of events that eventually get the field definitions and record data loaded into the dataset. The journey starts with Open.

LoadFromStream is used in the implementation of the LoadFromFile method.

See also: TCustomBufDataset.UniDirectional ([170](#)), TCustomBufDataset.SetDatasetPacket ([161](#)), TCustomBufDataset.LoadFromFile ([163](#)), TDataset.Open ([389](#)), TDataPacketFormat ([131](#))

### 5.15.21 TCustomBufDataset.SaveToStream

Synopsis: Saves the dataset to the specified stream using the given data format

Declaration: `procedure SaveToStream(AStream: TStream; Format: TDataPacketFormat)`

Visibility: public

Description: SaveToStream is a procedure used to store the content in the buffered dataset to a stream using the specified data format.

AStream is a TStream descendant where the field definitions and record data in the dataset will be stored.

Format is a TDataPacketFormat enumeration value that indicates the format used to stored dataset values into the stream. The default value is dfBinary, and causes the TFpcBinaryDatapacketReader to be used to write the content for the dataset. Use another value from the TDataPacketFormat enumeration to select a different registered data packet handler that handles the associated data format.

SaveToStream calls the CheckBiDirectional method to ensure that the dataset is opened for bi-directional record navigation. An Exception is raised if the UniDirectional property contains True. A temporary TDataPacketReader instance is created using the specified Format. SaveToStream calls the GetDatasetPacket method using the data packet handler to store the content in the buffered dataset to the stream.

SaveToStream is used in the implementation of the SaveToFile method.

See also: TCustomBufDataset.UniDirectional ([170](#)), TCustomBufDataset.SaveToFile ([163](#)), TDataPacketFormat ([131](#)), RegisterDatapacketReader ([132](#)), TDataPacketReaderClass ([131](#)), TDataPacketReader ([171](#))

### 5.15.22 TCustomBufDataset.LoadFromFile

Synopsis: Loads the dataset from the specified file using the given format

Declaration: `procedure LoadFromFile(AFileName: string; Format: TDataPacketFormat)`

Visibility: public

Description: LoadFromFile is a procedure used to load field definitions and record data from a file stored using a given data format.

AFileName is a String with the name of the file on the local file system. AFileName can contain optional path information, and should include the base name and extension for the file. For example:

```
ABufDataset.LoadFromFile('/usr/data/sample.bds');
```

or

```
ABufDataset.LoadFromFile('c:\usr\data\sample.bds');
```

The default value for AFileName is an empty string (""). When AFileName is an empty string, the value in the FileName property is used to load the content for the dataset. An Exception will be raised if both AFileName and FileName contain an empty String (""), or when the file name does not exist on the local file system.

Format is a TDataPacketFormat enumeration value that indicates the storage format used for the content in the file. The default value for Format is dfAny, and indicates that any registered data packet handler can be used to read the values in the file. Other values in the TDataPacketFormat enumeration indicate a specific registered data packet handler supporting a specific format should be used.

LoadFromFile creates a temporary TFileStream instance for the specified file name, and calls the LoadFromStream method to load the dataset from the stream using the specified file Format.

See also: TCustomBufDataset.FileName ([167](#)), TCustomBufDataset.LoadFromStream ([162](#))

### 5.15.23 TCustomBufDataset.SaveToFile

Synopsis: Saves the dataset to the specified file using the given data format

Declaration: `procedure SaveToFile(AFileName: string; Format: TDataPacketFormat)`

Visibility: public

**Description:** `SaveToFile` is a procedure used to save the content from the buffered dataset to the specified file name using a given data format.

`AFileName` is a String with the name of the file on the local file system. `AFileName` can contain optional path information, and should include the base name and extension for the file. For example:

```
ABufDataset.SaveToFile('/usr/data/sample.bds');
```

or

```
ABufDataset.SaveToFile('c:\usr\data\sample.bds');
```

The default value for `AFileName` is an empty string (''). When `AFileName` is omitted, the value in the `FileName` property will be used to save the content for the dataset. An Exception will be raised if both `AFileName` and `FileName` contain an empty String ('').

`Format` is a `TDataPacketFormat` enumeration value that indicates the storage format to use when writing the content for the file. The default value for `Format` is `dfBinary`, and indicates that the `TFpcBinaryDatapacketReader` should be used to write the values in the file. Other values in the `TDataPacketFormat` enumeration indicate a specific registered data packet handler supporting a specific format should be used.

`SaveToFile` creates a temporary `TFileStream` instance for the specified file name, and calls the `SaveToStream` method to store the dataset to the stream in the required `Format`.

See also: `TCustomBufDataset.FileName` ([167](#)), `TCustomBufDataset.SaveToStream` ([162](#))

#### 5.15.24 TCustomBufDataset.CreateDataset

**Synopsis:** Creates the dataset using its field definitions or bound fields

**Declaration:** `procedure CreateDataset`

**Visibility:** `public`

**Description:** `CreateDataset` is a procedure used to create the structure for a buffered dataset using the field definitions or bound fields defined in the class instance. `CreateDataset` calls `CheckInactive` to ensure that the dataset has not been opened. An exception is raised if the `Active` property contains `True`.

`CreateDataset` uses the `FieldDefs` and `Fields` properties to determine which property contains the structure for the dataset.

Preference is given to the `FieldDefs` property; it will be used even when bound fields have been created in the `Fields` property. The `CreateFields` method is called to create a bound field for each of the items in the `FieldDefs` collection.

The `Fields` property is used when there are no field definitions present in the `FieldDefs` property. The `InitFieldDefsFromFields` method is called to create a field definition in `FieldDefs` for each of the items in `Fields` collection. The `BindFields` method is called to link the items in `Fields` to the corresponding field definition in `FieldDefs`.

An exception is raised if both `FieldDefs` and `Fields` are empty.

`CreateDataset` sets the default value for an auto-increment field in the dataset to 1. `CreateDataset` also temporarily clears any value assigned to the `FileName` property; this is done to ensure that fields and record data in an existing file are not loaded when the dataset is Opened. The method is designed to create a dataset with field definitions but no record data. The value in `FileName` is restored after `Open` has been called in the method.

See also: [TDataSet.Active \(400\)](#), [TDataSet.FieldDefs \(395\)](#), [TDataSet.Fields \(398\)](#), [TDataSet.CreateFields \(371\)](#), [TDataSet.InitFieldDefsFromFields \(371\)](#), [TDataSet.BindFields \(371\)](#), [TDataSet.Open \(389\)](#), [TCustomBufDataset.FileName \(167\)](#)

### 5.15.25 TCustomBufDataset.Clear

Synopsis: Clears the content in the dataset

Declaration: `procedure Clear`

Visibility: `public`

Description: `Clear` is a procedure used to remove the content in the buffered dataset. `Clear` calls the `Close` method to ensure that the dataset has set its `Active` property to `False` and removed indexes and their storage. `Clear` removes field definition and bound fields in the `FieldDefs` and `Fields` properties.

See also: [TCustomBufDataset.Close \(153\)](#), [TDataSet.FieldDefs \(395\)](#), [TDataSet.Fields \(398\)](#)

### 5.15.26 TCustomBufDataset.BookmarkValid

Synopsis: Determines if the specified Bookmark is valid for the dataset

Declaration: `function BookmarkValid(ABookmark: TBookMark) : Boolean; Override`

Visibility: `public`

Description: `BookmarkValid` is an overridden `Boolean` function used to determine if the Bookmark specified in `ABookmark` is valid for the currently selected index in the dataset. `BookmarkValid` re-implements the method defined in the ancestor class.

`BookmarkValid` requires an index be selected and assigned to the `CurrentIndexBuf` property. The return value is `False` when no index is available to validate the bookmark. The return value is `True` when `ABookmark` represents a valid bookmark in the selected index.

See also: [TCustomBufDataset.CurrentIndexBuf \(153\)](#), [TBufIndex.BookmarkValid \(149\)](#)

### 5.15.27 TCustomBufDataset.CompareBookmarks

Synopsis: Gets the relative sort order for the specified Bookmarks

Declaration: `function CompareBookmarks(Bookmark1: TBookMark; Bookmark2: TBookMark) : LongInt; Override`

Visibility: `public`

Description: `CompareBookmarks` is an overridden `LongInt` function used to get the relative sort order for the specified Bookmarks. `CompareBookmarks` re-implements the method defined in the ancestor class.

The return value indicates the relative sort order for the compared bookmark values, and uses the following values and meanings:

**0** Compared bookmarks have the same value

**1** `Bookmark1` is not assigned, or `Bookmark1` occurs after `Bookmark2` in the index

**-1** `Bookmark2` is not assigned, or the current index has not been assigned (default value), or `Bookmark1` occurs before `Bookmark2` in the index

When the `CurentIndexBuf` property is assigned, its `CompareBookmarks` method is used to compare the bookmarks and get the return value for the method.

See also: [TCustomBufDataset.CurrentIndexBuf \(153\)](#), [TBufIndex \(145\)](#)

### 5.15.28 TCustomBufDataset.CopyFromDataset

Synopsis: Loads field definitions and optional data from another dataset

Declaration: `procedure CopyFromDataset (DataSet: TDataSet; CopyData: Boolean)`

Visibility: `public`

Description: `CopyFromDataset` is a procedure used to copy field definitions and optional record data for the dataset specified in the `DataSet` argument. `CopyData` indicates if record data is included in the copy operation. When `CopyData` contains `False`, only the field definitions in `DataSet` are copied to the buffered dataset.

`CopyFromDataset` calls the `Close` method prior to performing actions that clear field definitions and bound fields in the buffered dataset. A field definition is added to `FieldDefs` for each of the fields in `DataSet`. `CreateDataset` is called to bind `Fields` to the new field definitions, and to initialize index storage for the buffered dataset.

When `CopyData` contains `True`, record data from `DataSet` is copied to the class instance on a row-by-row and field-by-field basis. `Append` is called to add a new record. Internal lists with field definitions and bound fields are used to copy field values in the record (including `Blob` fields). Null field values are not copied in the method. The `DataType` for the field is used to read/write the field values. For blob fields, a temporary stream is used to read/write the field value. The `Post` method is called to store appended values in the buffered dataset. If an exception is raised, the `Cancel` method is called to clear the update and the exception is re-raised.

The record position in `DataSet` is restored to its original position prior to exiting from the method.

See also: `TDataSet.Close` (379), `TDataSet.Cancel` (378), `TDataSet.FieldDefs` (395)

### 5.15.29 TCustomBufDataset.ChangeCount

Synopsis: Number of pending changes for the dataset

Declaration: `Property ChangeCount : Integer`

Visibility: `public`

Access: `Read`

Description: `ChangeCount` is a read-only `Integer` property that indicates the number of pending changes in the update buffers allocated for the dataset. Update buffers are maintained in an internal `TRecord-sUpdateBuffer` member when methods that add, delete, or modify record data are called.

`ChangeCount` can be used in an application to determine if `ApplyUpdates`, `CancelUpdates`, or `RevertRecord` should be called prior to closing the dataset.

See also: `TCustomBufDataset.ApplyUpdates` (156), `TCustomBufDataset.CancelUpdates` (157), `TCustomBufDataset.RevertRecord` (157)

### 5.15.30 TCustomBufDataset.MaxIndexesCount

Synopsis: Maximum number of indexes available in the dataset

Declaration: `Property MaxIndexesCount : Integer`

Visibility: `public`

Access: `Read,Write`

**Description:** `MaxIndexesCount` is an `Integer` property which indicates the maximum number of indexes available in the buffered dataset. The default value for the property is 2; representing the automatically created default index ('`DEFAULT_ORDER`') and custom index ('`CUSTOM_ORDER`').

The value in `MaxIndexesCount` is assigned when the dataset is created, and updated when an index is added using the `AddIndex` method. Setting the value in the `MaxIndexesCount` property requires the dataset to be inactive (Active property contains `False`). An exception is raised if the dataset is Active, or the new value for the property is less than 2. You should not need to directly assign the value for the `MaxIndexesCount` property.

`MaxIndexesCount` is used in the `BufferOffset` method to determine the amount of space reserved for `TBufRecLinkItem` items in a record buffer for the dataset.

See also: `TDataset.Active` (400), `TCustomBufDataset.AddIndex` (160), `TCustomBufDataset.BufferOffset` (153)

### 5.15.31 TCustomBufDataset.ReadOnly

**Synopsis:** Indicates if records can be added, deleted, or modified in the dataset

**Declaration:** `Property ReadOnly : Boolean`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `ReadOnly` is a `Boolean` property which indicates if records can be added, deleted, or modified in the buffered dataset. The default value for the property is `False`.

`ReadOnly` is used, along with `UniDirectional`, when getting the value for the `CanModify` property. The dataset can be modified when both source properties contain the value `False`, which results in setting `CanModify` to `True`. Applications can set the value in `ReadOnly` to `False` to ensure that the dataset is not changeable regardless of the value in the `UniDirectional` property.

Use the items in the `FieldDefs` property to control whether individual fields defined for the dataset include the read-only field attribute. Use the items in the `Fields` property to determine if a field bound to a field definition has its `ReadOnly` property set.

See also: `TCustomBufDataset.UniDirectional` (170), `TDataset.CanModify` (393), `TDataset.FieldDefs` (395), `TDataset.Fields` (398)

### 5.15.32 TCustomBufDataset.ManualMergeChangeLog

**Synopsis:** Indicates if the update change log can be manually merged

**Declaration:** `Property ManualMergeChangeLog : Boolean`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `ManualMergeChangeLog` is a `Boolean` property which indicates if the update change log for the dataset can be manually merged. The default value for the property is `False`.

### 5.15.33 TCustomBufDataset.FileName

**Synopsis:** File name on the local file system used to load or store the dataset

**Declaration:** `Property FileName : TFileName`



Visibility: published

Access: Read,Write

**Description:** `FileName` is a `TFileName` property that contains a file name on the local file system used to load and/or store the content for the buffered dataset. `FileName` can contain optional path information needed to access the file, and must contain a valid file name and extension for the local file system. For example:

```
ABufDataset.FileName := '/usr/data/sample.bds';
```

or

```
ABufDataset.FileName := 'c:\usr\data\sample.bds';
```

The value in `FileName` is used in methods which load and/or save field definitions and record data for the dataset, such as `LoadFromFile` and `SaveToFile`. In these methods, `FileName` is used as the default value for an omitted file name argument in the method(s). `FileName` is used in the implementation of other methods such as: `InternalInitFieldDef`, `IntLoadFieldDefsFromFile`, and `InternalOpen`.

`FileName` is also used in the `DoBeforeClose` method called when the value in the `Active` property is changed from `True` to `False`. In this method, `FileName` is passed to `SaveToFile` as an argument prior to exiting from the method.

See also: `TCustomBufDataset.LoadFromFile` ([163](#)), `TCustomBufDataset.SaveToFile` ([163](#)), `TDataset.Open` ([389](#)), `TDataset.Close` ([379](#))

### 5.15.34 TCustomBufDataset.PacketRecords

**Synopsis:** Number of records allowed in a data packet handled by the packet reader

**Declaration:** `Property PacketRecords : Integer`

Visibility: published

Access: Read,Write

**Description:** `PacketRecords` is an `Integer` property that indicates the number of records allowed in a data packet handled by the packet reader. The default value for the property is 10, and is intended to minimize memory and network overhead when processing data packets for the dataset. The value in `PacketRecords` is used in the `FetchAll` and `GetNextPacket` methods.

Additional validation is performed when setting the value for `PacketRecords` to ensure that the new property value is not -1. An exception is raised when -1 is the value for the property.

### 5.15.35 TCustomBufDataset.OnUpdateError

**Synopsis:** Event handler signalled when an error occurs while updating records

**Declaration:** `Property OnUpdateError : TResolverErrorEvent`

Visibility: published

Access: Read,Write

**Description:** `OnUpdateError` is a `TResolverErrorEvent` property that represents the event handler signalled when an error occurs while applying updates to records in the buffered dataset. `OnUpdateError` allows an application to perform actions needed when a database exception occurs in the `ApplyUpdates` method.

Applications can assign a procedure to the event handler that implements the `TResolverErrorEvent` signature to respond to the event notification. The procedure must set the value in its `Response` argument to indicate whether the condition is handled, ignored, or can be re-raised in the calling method.

See also: `TResolverErrorEvent` ([132](#)), `TCustomBufDataset.ApplyUpdates` ([156](#))

### 5.15.36 TCustomBufDataset.IndexDefs

**Synopsis:** Index definitions for the dataset

**Declaration:** `Property IndexDefs : TIndexDefs`

**Visibility:** published

**Access:** Read

**Description:** `IndexDefs` is a read-only `TIndexDefs` property that contains the index definitions for the buffered dataset. Read access to the property is redirected to an internal `TBufDatasetIndexDefs` member used for the `BufIndexDefs` and `BufIndexes` properties.

See also: `TIndexDefs` ([469](#)), `TBufDatasetIndexDefs` ([130](#)), `TCustomBufDataset.BufIndexDefs` ([153](#)), `TCustomBufDataset.BufIndexes` ([153](#))

### 5.15.37 TCustomBufDataset.IndexName

**Synopsis:** Name of the selected index for the dataset

**Declaration:** `Property IndexName : string`

**Visibility:** published

**Access:** Read, Write

**Description:** `IndexName` is a `String` property that contains the name for the selected index for the buffered dataset. The value in `IndexName` is read from the corresponding property in `CurrentIndexBuf` (when assigned).

If an empty string (") is assigned to the property, the the default index is selected for the dataset and the property is updated to 'DEFAULT\_ORDER'. Any other value assigned to the property is compared to the index definitions for the dataset. If an index cannot be located with the specified name, a Database exception is raised. The index with the specified name (and its storage mechanism) are stored in the `CurrentIndexDef` and `CurrentIndexBuf` properties. If the dataset is Active, the `Resync` method is called to enable the new record order for the dataset.

See also: `TCustomBufDataset.CurrentIndexDef` ([153](#)), `TCustomBufDataset.CurrentIndexBuf` ([153](#)), `TDataset.Active` ([400](#)), `TDataset.Resync` ([390](#))

### 5.15.38 TCustomBufDataset.IndexFieldNames

Synopsis: Field names included in the custom index

Declaration: Property IndexFieldNames : string

Visibility: published

Access: Read,Write

Description: IndexFieldNames is a String property which contains a delimited list of field names used to construct the custom index for the dataset. Field names are separated by a ';' (Semicolon) delimiter. An optional directive can be included after the field name to indicate that the field should be in descending sort order in the index. For example:

```
ABufDataset.IndexFieldNames := 'LASTNAME; FIRSTNAME; UPDATETS DESC';
```

Please note that the leading Space character before the **DESC** directive is **required**.

When reading the value in IndexFieldNames, the current index is examined to determine if any of the field names in the property also appear in the descending fields for the index. The ' DESC' directive is added to the field name to indicate the sort order used in the index.

Setting the value for the property to an empty string (") causes the default index (' DEFAULT\_ORDER') to be used as the selected index in the CurrentIndexDef property. When the new property value is not an empty string, and the dataset is Active, the BuildCustomIndex method is called to populate index storage with values for the specified field names.

See also: TCustomBufDataset.CurrentIndexDef ([153](#)), TDataset.Active ([130](#)), TCustomBufDataset.BuildCustomIndex ([153](#))

### 5.15.39 TCustomBufDataset.UniDirectional

Synopsis: Indicates if the dataset is for uni-directional navigation only

Declaration: Property UniDirectional : Boolean

Visibility: published

Access: Read,Write

Description: UniDirectional is a Boolean property which indicates if the dataset is limited to forward navigation through its records. The default value for the property is False.

UniDirectional is used, along with ReadOnly, to determine if the dataset can be modified. When either property contains True, the dataset cannot be changed.

UniDirectional also affects the index storage mechanisms created when indexes are initialized. When UniDirectional is True, the TUniDirectionalBufIndex type (which omits bookmarks and record numbers) is used for index storage mechanisms. In addition, the custom index (' CUSTOM\_ORDER') is skipped for the uni-directional dataset.

Setting the value in UniDirectional requires the dataset to be inactive (Active contains False) to allow existing indexes to be cleared and rebuilt. Use the Close method to close the dataset prior to setting the value in the UniDirectional property.

See also: TCustomBufDataset.BufIndexDefs ([153](#)), TCustomBufDataset.BufIndexes ([153](#)), TDataset.ReadOnly ([371](#)), TDataset.Active ([400](#))

## 5.16 TDataPacketReader

### 5.16.1 Description

TDataPacketReader is a TObject descendant which implements a data packet reader for TBufDataset. TDataPacketReader is a concept borrowed from TClientDataset in Delphi. Data packets are used to access the values in its dataset. Data packets can contain field definitions or record data. Data packets can store their values in binary format or as an XML document.

TDataPacketReader provides properties and methods that allow access to the field definitions, field values, and row states for record buffers in its dataset. Methods are provided to load and save field definitions or record data using a stream. Methods are provided to recognize the storage format for the data packets in the stream.

Many of the methods in TDataPacketReader are declared as virtual or abstract. They are implemented in descendent classes that use a specific data packet format, such as TFpcBinaryDatapacketReader or TXMLDatapacketReader (in xmldatapacketreader.pp).

See also: TFpcBinaryDatapacketReader (178), TXMLDatapacketReader (??)

### 5.16.2 Method overview

Page	Method	Description
<a href="#">171</a>	Create	Constructor for the class instance
<a href="#">173</a>	FinalizeStoreRecords	Called after all records are stored
<a href="#">172</a>	GetCurrentRecord	Returns True if there is at least one more record available in the stream
<a href="#">172</a>	GetRecordRowState	Returns the RowState of the current record, and the order of the update
<a href="#">172</a>	GotoNextRecord	Moves to the next record in the stream
<a href="#">172</a>	InitLoadRecords	Called before the records are loaded
<a href="#">171</a>	LoadFieldDefs	Loads the field definitions for the dataset from a stream
<a href="#">173</a>	RecognizeStream	Checks if the provided stream is of the right format for this class
<a href="#">172</a>	RestoreRecord	Restores a record from the stream into the current record buffer
<a href="#">172</a>	StoreFieldDefs	Saves field definitions to a stream
<a href="#">172</a>	StoreRecord	Saves the current record buffer to the stream

### 5.16.3 TDataPacketReader.Create

Synopsis: Constructor for the class instance

Declaration: `constructor Create(ADataset: TCustomBufDataset; AStream: TStream); Virtual`

Visibility: public

### 5.16.4 TDataPacketReader.LoadFieldDefs

Synopsis: Loads the field definitions for the dataset from a stream

Declaration: `procedure LoadFieldDefs(var AnAutoIncValue: Integer); Virtual; Abstract`

Visibility: public

### 5.16.5 TDataPacketReader.InitLoadRecords

Synopsis: Called before the records are loaded

Declaration: `procedure InitLoadRecords; Virtual; Abstract`

Visibility: public

### 5.16.6 TDataPacketReader.GetCurrentRecord

Synopsis: Returns True if there is at least one more record available in the stream

Declaration: `function GetCurrentRecord : Boolean; Virtual; Abstract`

Visibility: public

### 5.16.7 TDataPacketReader.GetRecordRowState

Synopsis: Returns the RowState of the current record, and the order of the update

Declaration: `function GetRecordRowState(out AUpdOrder: Integer) : TRowState; Virtual  
; Abstract`

Visibility: public

### 5.16.8 TDataPacketReader.RestoreRecord

Synopsis: Restores a record from the stream into the current record buffer

Declaration: `procedure RestoreRecord; Virtual; Abstract`

Visibility: public

### 5.16.9 TDataPacketReader.GotoNextRecord

Synopsis: Moves to the next record in the stream

Declaration: `procedure GotoNextRecord; Virtual; Abstract`

Visibility: public

### 5.16.10 TDataPacketReader.StoreFieldDefs

Synopsis: Saves field definitions to a stream

Declaration: `procedure StoreFieldDefs(AnAutoIncValue: Integer); Virtual; Abstract`

Visibility: public

### 5.16.11 TDataPacketReader.StoreRecord

Synopsis: Saves the current record buffer to the stream

Declaration: `procedure StoreRecord(ARowState: TRowState; AUpdOrder: Integer)  
; Virtual; Abstract`

Visibility: public

### 5.16.12 TDataPacketReader.FinalizeStoreRecords

Synopsis: Called after all records are stored

Declaration: `procedure FinalizeStoreRecords; Virtual; Abstract`

Visibility: `public`

### 5.16.13 TDataPacketReader.RecognizeStream

Synopsis: Checks if the provided stream is of the right format for this class

Declaration: `class function RecognizeStream(AStream: TStream) : Boolean; Virtual  
; Abstract`

Visibility: `public`

## 5.17 TDoubleLinkedBufIndex

### 5.17.1 Description

TDoubleLinkedBufIndex is a TBufIndex descendant that implements an index using a doubly-linked list. Nodes in the doubly-linked list are implemented using the TBufRecLinkItem record type and the PBufRecLinkItem pointer type. TDoubleLinkedBufIndex provides two sentinel nodes that represent the first and last items in the linked list.

TDoubleLinkedBufIndex is the type used to implement the default index ('DEFAULT\_ORDER') in TBufDataset.

See also: TBufIndex ([145](#)), TBufRecLinkItem ([133](#)), PBufRecLinkItem ([131](#)), TBufDataset ([139](#))

**5.17.2 Method overview**

Page	Method	Description
178	AddRecord	Adds a record to the index
177	BeginUpdate	
176	CanScrollForward	Indicates if the index can be scrolled towards the end of the index
177	CompareBookmarks	
176	DoScrollForward	Implements actions needed to scroll forward using the index
178	EndUpdate	
174	GetCurrent	
175	GetRecord	
176	GotoBookmark	Navigates the index to the specified Bookmark
177	InitialiseIndex	Initializes the index
177	InitialiseSpareRecord	Initializes the spare record in the index
178	InsertRecordBeforeCurrentRecord	Inserts a record prior to the current record in the index
178	OrderCurrentRecord	
177	ReleaseSpareRecord	Releases the spare record in the index
178	RemoveRecordFromIndex	Removes the record at the specified Bookmark from the index
176	RestoreCurrentRecord	
177	SameBookmarks	Compares Bookmark values for ordering in the index
174	ScrollBackward	Scrolls the index toward the beginning of the index
175	ScrollFirst	Scrolls to the first entry in the index
174	ScrollForward	Scrolls the index towards the end of the index
175	ScrollLast	Scrolls to the last entry in the index
175	SetToFirstRecord	
175	SetToLastRecord	
176	StoreCurrentRecIntoBookmark	
175	StoreCurrentRecord	
176	StoreSpareRecIntoBookmark	

**5.17.3 TDoubleLinkedBufIndex.ScrollBackward**

Synopsis: Scrolls the index toward the beginning of the index

Declaration: `function ScrollBackward : TGetResult; Override`

Visibility: public

**5.17.4 TDoubleLinkedBufIndex.ScrollForward**

Synopsis: Scrolls the index towards the end of the index

Declaration: `function ScrollForward : TGetResult; Override`

Visibility: public

**5.17.5 TDoubleLinkedBufIndex.GetCurrent**

Synopsis:

Declaration: function GetCurrent : TGetResult; Override

Visibility: public

#### **5.17.6 TDoubleLinkedBufIndex.ScrollFirst**

Synopsis: Scrolls to the first entry in the index

Declaration: function ScrollFirst : TGetResult; Override

Visibility: public

#### **5.17.7 TDoubleLinkedBufIndex.ScrollLast**

Synopsis: Scrolls to the last entry in the index

Declaration: procedure ScrollLast; Override

Visibility: public

#### **5.17.8 TDoubleLinkedBufIndex.GetRecord**

Synopsis:

Declaration: function GetRecord(ABookmark: PBufBookmark; GetMode: TGetMode)  
: TGetResult; Override

Visibility: public

#### **5.17.9 TDoubleLinkedBufIndex.SetToFirstRecord**

Synopsis:

Declaration: procedure SetToFirstRecord; Override

Visibility: public

#### **5.17.10 TDoubleLinkedBufIndex.SetToLastRecord**

Synopsis:

Declaration: procedure SetToLastRecord; Override

Visibility: public

#### **5.17.11 TDoubleLinkedBufIndex.StoreCurrentRecord**

Synopsis:

Declaration: procedure StoreCurrentRecord; Override

Visibility: public



### 5.17.12 TDoubleLinkedBufIndex.RestoreCurrentRecord

Synopsis:

Declaration: `procedure RestoreCurrentRecord; Override`

Visibility: `public`

### 5.17.13 TDoubleLinkedBufIndex.CanScrollForward

Synopsis: Indicates if the index can be scrolled towards the end of the index

Declaration: `function CanScrollForward : Boolean; Override`

Visibility: `public`

### 5.17.14 TDoubleLinkedBufIndex.DoScrollForward

Synopsis: Implements actions needed to scroll forward using the index

Declaration: `procedure DoScrollForward; Override`

Visibility: `public`

### 5.17.15 TDoubleLinkedBufIndex.StoreCurrentRecIntoBookmark

Synopsis:

Declaration: `procedure StoreCurrentRecIntoBookmark(const ABookmark: PBufBookmark)  
; Override`

Visibility: `public`

### 5.17.16 TDoubleLinkedBufIndex.StoreSpareRecIntoBookmark

Synopsis:

Declaration: `procedure StoreSpareRecIntoBookmark(const ABookmark: PBufBookmark)  
; Override`

Visibility: `public`

### 5.17.17 TDoubleLinkedBufIndex.GotoBookmark

Synopsis: Navigates the index to the specified Bookmark

Declaration: `procedure GotoBookmark(const ABookmark: PBufBookmark); Override`

Visibility: `public`

### 5.17.18 TDoubleLinkedBufIndex.CompareBookmarks

Synopsis:

Declaration: `function CompareBookmarks(const ABookmark1: PBufBookmark;  
const ABookmark2: PBufBookmark) : Integer  
; Override`

Visibility: public

### 5.17.19 TDoubleLinkedBufIndex.SameBookmarks

Synopsis: Compares Bookmark values for ordering in the index

Declaration: `function SameBookmarks(const ABookmark1: PBufBookmark;  
const ABookmark2: PBufBookmark) : Boolean  
; Override`

Visibility: public

### 5.17.20 TDoubleLinkedBufIndex.InitialiseIndex

Synopsis: Initializes the index

Declaration: `procedure InitialiseIndex; Override`

Visibility: public

### 5.17.21 TDoubleLinkedBufIndex.InitialiseSpareRecord

Synopsis: Initializes the spare record in the index

Declaration: `procedure InitialiseSpareRecord(const ASpareRecord: TRecordBuffer)  
; Override`

Visibility: public

### 5.17.22 TDoubleLinkedBufIndex.ReleaseSpareRecord

Synopsis: Releases the spare record in the index

Declaration: `procedure ReleaseSpareRecord; Override`

Visibility: public

### 5.17.23 TDoubleLinkedBufIndex.BeginUpdate

Synopsis:

Declaration: `procedure BeginUpdate; Override`

Visibility: public

#### 5.17.24 TDoubleLinkedBufIndex.AddRecord

Synopsis: Adds a record to the index

Declaration: `procedure AddRecord; Override`

Visibility: `public`

#### 5.17.25 TDoubleLinkedBufIndex.InsertRecordBeforeCurrentRecord

Synopsis: Inserts a record prior to the current record in the index

Declaration: `procedure InsertRecordBeforeCurrentRecord(const ARecord: TRecordBuffer)  
; Override`

Visibility: `public`

#### 5.17.26 TDoubleLinkedBufIndex.RemoveRecordFromIndex

Synopsis: Removes the record at the specified Bookmark from the index

Declaration: `procedure RemoveRecordFromIndex(const ABookmark: TBufBookmark)  
; Override`

Visibility: `public`

#### 5.17.27 TDoubleLinkedBufIndex.OrderCurrentRecord

Synopsis:

Declaration: `procedure OrderCurrentRecord; Override`

Visibility: `public`

#### 5.17.28 TDoubleLinkedBufIndex.EndUpdate

Synopsis:

Declaration: `procedure EndUpdate; Override`

Visibility: `public`

### 5.18 TFpcBinaryDatapacketReader

#### 5.18.1 Description

TFpcBinaryDatapacketReader is a TDatapacketReader descendant that implements a data packet reader using binary data packets. Binary data packets use the following layout:

Table 5.4:

Section	Name	Length	Description
Header	Identification	13 bytes	Contains the value 'B'
	Version	1 byte	Version number
	Field Definition Count	2 bytes	Column Layout for field
Field Definitions	Field Name Length	2 bytes	Section occurs the number of times
	Field Name	See above	Name of the field in the dataset
	Display Name Length	2 bytes	
	Display Name	See above	Caption for the field
	Data Type	2 bytes	
Parameters	Read-Only Attribute	1 byte	0=read/write, 1=read-only
	AutoInc Value	4 bytes	Integer value for the auto-increment
Row Header	Row Marker	1 byte	Contains the value \$F
	Row State	1 byte	0=original, 1=deleted
	Update Order	4 bytes	
	Null Bitmap	Variable length.	1 byte required for each field
Row Data	Optional Field Length	4 bytes	Variable-length fields
	Field Data	Bytes values for the field. See Optional Field Length.	Occurs the number of times

See also: TDatapacketReader ([171](#))

### 5.18.2 Method overview

Page	Method	Description
<a href="#">179</a>	Create	Constructor for the class instance
<a href="#">181</a>	FinalizeStoreRecords	
<a href="#">180</a>	GetCurrentRecord	
<a href="#">180</a>	GetRecordRowState	
<a href="#">180</a>	GotoNextRecord	
<a href="#">180</a>	InitLoadRecords	
<a href="#">179</a>	LoadFieldDefs	Loads field definitions using the data packet reader
<a href="#">181</a>	RecognizeStream	
<a href="#">180</a>	RestoreRecord	
<a href="#">180</a>	StoreFieldDefs	Stores field definitions using the data packet reader
<a href="#">181</a>	StoreRecord	

### 5.18.3 TFpcBinaryDatapacketReader.Create

Synopsis: Constructor for the class instance

Declaration: `constructor Create(ADataSet: TCustomBufDataset; AStream: TStream)  
; Override`

Visibility: public

### 5.18.4 TFpcBinaryDatapacketReader.LoadFieldDefs

Synopsis: Loads field definitions using the data packet reader

Declaration: `procedure LoadFieldDefs(var AnAutoIncValue: Integer); Override`

Visibility: public

### **5.18.5 TFpcBinaryDatapacketReader.StoreFieldDefs**

Synopsis: Stores field definitions using the data packet reader

Declaration: `procedure StoreFieldDefs (AnAutoIncValue: Integer); Override`

Visibility: public

### **5.18.6 TFpcBinaryDatapacketReader.InitLoadRecords**

Synopsis:

Declaration: `procedure InitLoadRecords; Override`

Visibility: public

### **5.18.7 TFpcBinaryDatapacketReader.GetCurrentRecord**

Synopsis:

Declaration: `function GetCurrentRecord : Boolean; Override`

Visibility: public

### **5.18.8 TFpcBinaryDatapacketReader.GetRecordRowState**

Synopsis:

Declaration: `function GetRecordRowState(out AUpdOrder: Integer) : TRowState  
; Override`

Visibility: public

### **5.18.9 TFpcBinaryDatapacketReader.RestoreRecord**

Synopsis:

Declaration: `procedure RestoreRecord; Override`

Visibility: public

### **5.18.10 TFpcBinaryDatapacketReader.GotoNextRecord**

Synopsis:

Declaration: `procedure GotoNextRecord; Override`

Visibility: public

### 5.18.11 TFpcBinaryDatapacketReader.StoreRecord

Synopsis:

Declaration: `procedure StoreRecord(ARowState: TRowState; AUpdOrder: Integer)  
; Override`

Visibility: public

### 5.18.12 TFpcBinaryDatapacketReader.FinalizeStoreRecords

Synopsis:

Declaration: `procedure FinalizeStoreRecords; Override`

Visibility: public

### 5.18.13 TFpcBinaryDatapacketReader.RecognizeStream

Synopsis:

Declaration: `class function RecognizeStream(AStream: TStream) : Boolean; Override`

Visibility: public

## 5.19 TUniDirectionalBufIndex

### 5.19.1 Description

TUniDirectionalBufIndex is a TBufIndex descendant that implements a uni-directional index. TUniDirectionalBufIndex does not require bookmarks used for navigation; it is uni-directional and the next record is always available in the buffers allocated for the index.

See also: TBufIndex ([145](#))

**5.19.2 Method overview**

Page	Method	Description
<a href="#">185</a>	AddRecord	
<a href="#">185</a>	BeginUpdate	
<a href="#">183</a>	CanScrollForward	
<a href="#">184</a>	DoScrollForward	
<a href="#">185</a>	EndUpdate	
<a href="#">182</a>	GetCurrent	
<a href="#">184</a>	GotoBookmark	
<a href="#">184</a>	InitialiseIndex	
<a href="#">184</a>	InitialiseSpareRecord	
<a href="#">185</a>	InsertRecordBeforeCurrentRecord	
<a href="#">185</a>	OrderCurrentRecord	
<a href="#">184</a>	ReleaseSpareRecord	
<a href="#">185</a>	RemoveRecordFromIndex	
<a href="#">183</a>	RestoreCurrentRecord	
<a href="#">182</a>	ScrollBackward	
<a href="#">182</a>	ScrollFirst	
<a href="#">182</a>	ScrollForward	
<a href="#">183</a>	ScrollLast	
<a href="#">183</a>	SetToFirstRecord	
<a href="#">183</a>	SetToLastRecord	
<a href="#">184</a>	StoreCurrentRecIntoBookmark	
<a href="#">183</a>	StoreCurrentRecord	
<a href="#">184</a>	StoreSpareRecIntoBookmark	

**5.19.3 TUniDirectionalBuflIndex.ScrollBackward**

Synopsis:

Declaration: `function ScrollBackward : TGetResult; Override`Visibility: `public`**5.19.4 TUniDirectionalBuflIndex.ScrollForward**

Synopsis:

Declaration: `function ScrollForward : TGetResult; Override`Visibility: `public`**5.19.5 TUniDirectionalBuflIndex.GetCurrent**

Synopsis:

Declaration: `function GetCurrent : TGetResult; Override`Visibility: `public`**5.19.6 TUniDirectionalBuflIndex.ScrollFirst**

Synopsis:

Declaration: function ScrollFirst : TGetResult; Override

Visibility: public

### **5.19.7 TUniDirectionalBuflIndex.ScrollLast**

Synopsis:

Declaration: procedure ScrollLast; Override

Visibility: public

### **5.19.8 TUniDirectionalBuflIndex.SetToFirstRecord**

Synopsis:

Declaration: procedure SetToFirstRecord; Override

Visibility: public

### **5.19.9 TUniDirectionalBuflIndex.SetToLastRecord**

Synopsis:

Declaration: procedure SetToLastRecord; Override

Visibility: public

### **5.19.10 TUniDirectionalBuflIndex.StoreCurrentRecord**

Synopsis:

Declaration: procedure StoreCurrentRecord; Override

Visibility: public

### **5.19.11 TUniDirectionalBuflIndex.RestoreCurrentRecord**

Synopsis:

Declaration: procedure RestoreCurrentRecord; Override

Visibility: public

### **5.19.12 TUniDirectionalBuflIndex.CanScrollForward**

Synopsis:

Declaration: function CanScrollForward : Boolean; Override

Visibility: public



### 5.19.13 TUniDirectionalBuflIndex.DoScrollForward

Synopsis:

Declaration: `procedure DoScrollForward; Override`

Visibility: `public`

### 5.19.14 TUniDirectionalBuflIndex.StoreCurrentRecIntoBookmark

Synopsis:

Declaration: `procedure StoreCurrentRecIntoBookmark(const ABookmark: PBufBookmark)  
; Override`

Visibility: `public`

### 5.19.15 TUniDirectionalBuflIndex.StoreSpareRecIntoBookmark

Synopsis:

Declaration: `procedure StoreSpareRecIntoBookmark(const ABookmark: PBufBookmark)  
; Override`

Visibility: `public`

### 5.19.16 TUniDirectionalBuflIndex.GotoBookmark

Synopsis:

Declaration: `procedure GotoBookmark(const ABookmark: PBufBookmark); Override`

Visibility: `public`

### 5.19.17 TUniDirectionalBuflIndex.InitialiseIndex

Synopsis:

Declaration: `procedure InitialiseIndex; Override`

Visibility: `public`

### 5.19.18 TUniDirectionalBuflIndex.InitialiseSpareRecord

Synopsis:

Declaration: `procedure InitialiseSpareRecord(const ASpareRecord: TRecordBuffer)  
; Override`

Visibility: `public`

### 5.19.19 TUniDirectionalBuflIndex.ReleaseSpareRecord

Synopsis:

Declaration: `procedure ReleaseSpareRecord; Override`

Visibility: `public`

### **5.19.20 TUniDirectionalBuflIndex.BeginUpdate**

Synopsis:

Declaration: `procedure BeginUpdate; Override`

Visibility: `public`

### **5.19.21 TUniDirectionalBuflIndex.AddRecord**

Synopsis:

Declaration: `procedure AddRecord; Override`

Visibility: `public`

### **5.19.22 TUniDirectionalBuflIndex.InsertRecordBeforeCurrentRecord**

Synopsis:

Declaration: `procedure InsertRecordBeforeCurrentRecord(const ARecord: TRecordBuffer)  
; Override`

Visibility: `public`

### **5.19.23 TUniDirectionalBuflIndex.RemoveRecordFromIndex**

Synopsis:

Declaration: `procedure RemoveRecordFromIndex(const ABookmark: TBufBookmark)  
; Override`

Visibility: `public`

### **5.19.24 TUniDirectionalBuflIndex.OrderCurrentRecord**

Synopsis:

Declaration: `procedure OrderCurrentRecord; Override`

Visibility: `public`

### **5.19.25 TUniDirectionalBuflIndex.EndUpdate**

Synopsis:

Declaration: `procedure EndUpdate; Override`

Visibility: `public`

## Chapter 6

# Reference for unit 'bufstream'

### 6.1 Used units

Table 6.1: Used units by unit 'bufstream'

Name	Page
Classes	??
sysutils	??

### 6.2 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 ([190](#)) which is for reading only, and TWriteBufStream ([191](#)) 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.

### 6.3 Constants, types and variables

#### 6.3.1 Constants

`DefaultBufferCapacity : Integer = 16`

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

## 6.4 TBufferedFileStream

### 6.4.1 Method overview

Page	Method	Description
<a href="#">187</a>	Create	
<a href="#">187</a>	Destroy	
<a href="#">187</a>	Flush	
<a href="#">188</a>	InitializeCache	
<a href="#">187</a>	Read	
<a href="#">187</a>	Seek	
<a href="#">187</a>	Write	

### 6.4.2 TBufferedFileStream.Create

```
Declaration: constructor Create(const AFileName: string; Mode: Word)
           constructor Create(const AFileName: string; Mode: Word;
                               Rights: Cardinal)
```

Visibility: public

### 6.4.3 TBufferedFileStream.Destroy

**Declaration:** destructor Destroy; Override

Visibility: public

#### 6.4.4 TBufferedFileStream.Seek

```
Declaration: function Seek(Offset: LongInt; Origin: Word) : LongInt;  Override
              ;  Overload
              function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64
              ;  Override;  Overload
```

Visibility: public

#### 6.4.5 TBufferedFileStream.Read

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

Visibility: public

#### 6.4.6 TBufferedFileStream.Write

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

Visibility: public

#### 6.4.7 TBufferedFileStream.Flush

```
Declaration: procedure Flush
```

Visibility: public

### 6.4.8 TBufferedFileStream.InitializeCache

**Declaration:** `procedure InitializeCache(const aCacheBlockSize: Integer;  
const aCacheBlockCount: Integer)`

**Visibility:** public

## 6.5 TBufStream

### 6.5.1 Description

`TBufStream` is the common ancestor for the `TReadBufStream` (190) and `TWriteBufStream` (191) streams. It completely handles the buffer memory management and position management. An instance of `TBufStream` should never be created directly. It also keeps the instance of the source stream.

See also: `TReadBufStream` (190), `TWriteBufStream` (191)

### 6.5.2 Method overview

Page	Method	Description
188	Create	Create a new <code>TBufStream</code> instance.
189	Destroy	Destroys the <code>TBufStream</code> instance

### 6.5.3 Property overview

Page	Properties	Access	Description
189	Buffer	r	The current buffer
189	BufferPos	r	Current buffer position.
190	BufferSize	r	Amount of data in the buffer
189	Capacity	rw	Current buffer capacity

### 6.5.4 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` (186) is used as the capacity.

An instance of `TBufStream` should never be instantiated directly. Instead, an instance of `TReadBufStream` (190) or `TWriteBufStream` (191) should be created.

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

See also: `TBufStream.Destroy` (189), `TReadBufStream` (190), `TWriteBufStream` (191)

### 6.5.5 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 (188), TReadBufStream (190), TWriteBufStream (191)

### 6.5.6 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 (188), TBufStream.Capacity (189), TBufStream.BufferSize (190)

### 6.5.7 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 (188), TBufStream.Buffer (189), TBufStream.BufferSize (190), TBufStream.BufferPos (189)

### 6.5.8 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 (188), TBufStream.Buffer (189), TBufStream.BufferSize (190), TBufStream.Capacity (189)

### 6.5.9 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` (189).

See also: `TBufStream.Create` (188), `TBufStream.Buffer` (189), `TBufStream.BufferPos` (189), `TBufStream.Capacity` (189)

## 6.6 TReadBufStream

### 6.6.1 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.

See also: `TBufStream` (188), `TWriteBufStream` (191)

### 6.6.2 Method overview

Page	Method	Description
<a href="#">191</a>	<code>Read</code>	Reads data from the stream
<a href="#">190</a>	<code>Seek</code>	Set location in the buffer

### 6.6.3 TReadBufStream.Seek

Synopsis: Set location in the buffer

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; 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` (191), `TReadBufStream.Read` (191), `TReadBufStream.Write` (190)

### 6.6.4 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` ([190](#)), `TReadBufStream.Read` ([191](#))

## 6.7 TWriteBufStream

### 6.7.1 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.

See also: `TBufStream` ([188](#)), `TReadBufStream` ([190](#))

### 6.7.2 Method overview

Page	Method	Description
<a href="#">191</a>	<code>Destroy</code>	Remove the <code>TWriteBufStream</code> instance from memory
<a href="#">191</a>	<code>Seek</code>	Set stream position.
<a href="#">192</a>	<code>Write</code>	Write data to the stream

### 6.7.3 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` ([189](#)).

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

See also: `Create` ([188](#)), `TBufStream.Destroy` ([189](#))

### 6.7.4 TWriteBufStream.Seek

Synopsis: Set stream position.

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; 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.

See also: `TWriteBufStream.Write` ([192](#)), `TWriteBufStream.Read` ([191](#)), `TReadBufStream.Seek` ([190](#))

### 6.7.5 `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` ([191](#)), `TWriteBufStream.Read` ([191](#)), `TReadBufStream.Write` ([190](#))

## Chapter 7

# Reference for unit 'CacheCls'

### 7.1 Used units

Table 7.1: Used units by unit 'CacheCls'

Name	Page
sysutils	??

### 7.2 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`

### 7.3 Constants, types and variables

#### 7.3.1 Resource strings

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

Message shown when an invalid index is passed.

#### 7.3.2 Types

```
PCacheSlot = ^TCacheSlot
```

Pointer to `TCacheSlot` ([194](#)) record.

```
PCacheSlotArray = ^TCacheSlotArray
```

Pointer to `TCacheSlotArray` ([194](#)) array

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

`TCacheSlotArray` is an array of `TCacheSlot` items. Do not use `TCacheSlotArray` directly, instead, use `PCacheSlotArray` (193) 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` (195) 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 anisstrings, then the contents should be compared.

## 7.4 TCacheSlot

```
TCacheSlot = record
    Prev : PCacheSlot;
    Next : PCacheSlot;
    Data
        : Pointer;
    Index : Integer;
end
```

`TCacheSlot` is internally used by the `TCache` (194) class. It represents 1 element in the linked list.

## 7.5 ECacheError

### 7.5.1 Description

Exception class used in the `cachecls` unit.

## 7.6 TCache

### 7.6.1 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` (197) property offers indexed access to the array of items. When accessing the array through this property, the `MRUSlot` (197) property is updated.

### 7.6.2 Method overview

Page	Method	Description
<a href="#">195</a>	Add	Add a data element to the list.
<a href="#">196</a>	AddNew	Add a new item to the list.
<a href="#">195</a>	Create	Create a new cache class.
<a href="#">195</a>	Destroy	Free the TCache class from memory
<a href="#">196</a>	FindSlot	Find data pointer in the list
<a href="#">196</a>	IndexOf	Return index of a data pointer in the list.
<a href="#">197</a>	Remove	Remove a data item from the list.

### 7.6.3 Property overview

Page	Properties	Access	Description
<a href="#">197</a>	Data	rw	Indexed access to data items
<a href="#">198</a>	LRUSlot	r	Last used item
<a href="#">197</a>	MRUSlot	rw	Most recent item slot.
<a href="#">199</a>	OnFreeSlot	rw	Event called when a slot is freed
<a href="#">198</a>	OnIsDataEqual	rw	Event to compare 2 items.
<a href="#">198</a>	SlotCount	rw	Number of slots in the list
<a href="#">198</a>	Slots	r	Indexed array to the slots

### 7.6.4 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` ([198](#))

### 7.6.5 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` ([195](#))

### 7.6.6 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` (196) 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` (196), `TCache.FindSlot` (196), `TCache.IndexOf` (196), `TCache.Data` (197), `TCache.MRUSlot` (197)

### 7.6.7 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` (195) 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` (195), `TCache.FindSlot` (196), `TCache.IndexOf` (196), `TCache.Data` (197), `TCache.MRUSlot` (197)

### 7.6.8 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` (198) event must be set.

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

See also: `TCache.IndexOf` (196), `TCache.Add` (195), `TCache.OnIsDataEqual` (198)

### 7.6.9 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` (198) event must be set.

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

See also: `TCache.FindSlot` (196), `TCache.Add` (195), `TCache.OnIsDataEqual` (198)

### 7.6.10 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` (196)

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

The index is zero-based and can maximally be `SlotCount-1` (198). Providing an invalid index will result in an exception.

See also: `TCache.MRUSlot` (197)

### 7.6.12 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` (197) property, or when an item is added to the list with `Add` (195) or `AddNew` (196)

See also: `TCache.Add` (195), `TCache.AddNew` (196), `TCache.Data` (197), `TCache.LRUSlot` (198)

### 7.6.13 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 ([195](#)), TCache.AddNew ([196](#)), TCache.Data ([197](#)), TCache.MRUSlot ([197](#))

### 7.6.14 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 ([199](#)) is called.

See also: TCache.Create ([195](#)), TCache.Data ([197](#)), TCache.Slots ([198](#))

### 7.6.15 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 ([198](#)). Providing an invalid index will result in an exception.

See also: TCache.Data ([197](#)), TCache.SlotCount ([198](#))

### 7.6.16 TCache.OnIsDataEqual

Synopsis: Event to compare 2 items.

Declaration: Property OnIsDataEqual : TOnIsDataEqual

Visibility: public

Access: Read,Write

**Description:** `OnIsDataEqual` is used by `FindSlot` (196) and `IndexOf` (196) to compare items when looking for a particular item. These functions are called by the `Add` (195) 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` (196), `TCache.IndexOf` (196), `TCache.Add` (195)

### 7.6.17 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` (195), `TCache.AddNew` (196), `TCache.SlotCount` (198)



## Chapter 8

# Reference for unit 'Contrns'

### 8.1 Used units

Table 8.1: Used units by unit 'Contrns'

Name	Page
Classes	??
sysutils	??

### 8.2 Overview

The `contrns` unit implements various general-purpose classes:

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

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

**Class lists** lists that manage class pointers instead of pointers.

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

**Queues** Classes to manage a FIFO list of pointers or objects

**Hash lists** General-purpose Hash lists.

### 8.3 Constants, types and variables

#### 8.3.1 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.

### 8.3.2 Types

`PBucket = ^TBucket`

Pointer to `TBucket` (204)" type.

`PHashItem = ^THashItem`

`PHashItem` is a pointer type, pointing to the `THashItem` (205) record.

`PHashItemList = ^THashItemList`

`PHashItemList` is a pointer to the `THashItemList` (202). It's used in the `TFPHashList` (221) as a pointer to the memory area containing the hash item records.

`PHashTable = ^THashTable`

`PHashTable` is a pointer to the `THashTable` (202). It's used in the `TFPHashList` (221) as a pointer to the memory area containing the hash values.

`Array = Array of TBucket`

Array of `TBucket` (204) records.

`Array = Array of TBucketItem`

Array of `TBucketItem` records

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

Table 8.2: Enumeration values for type `TBucketListSizes`

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

TBucketListSizes is used to set the bucket list size: It specified the number of buckets created by TBucketList (205).

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

TBucketProc is the prototype for the TCustomBucketList.Foreach (214) call. It is the plain procedural form. The Continue parameter can be set to False to indicate that the Foreach call should stop the iteration.

For a procedure of object (a method) callback, see the TBucketProcObject (202) prototype.

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

TBucketProcObject is the prototype for the TCustomBucketList.Foreach (214) call. It is the method (procedure of object) form. The Continue parameter can be set to False to indicate that the Foreach call should stop the iteration.

For a plain procedural callback, see the TBucketProc (202) prototype.

```
TDataIteratorCallBack = procedure(Item: Pointer; const Key: string
;
                                var Continue: Boolean)
```

TDataIteratorCallBack is a callback prototype for the TFPDataHashTable.Iterate (220) static CallBack. 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.

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

TDataIteratorMethod is a callback prototype for the TFPDataHashTable.Iterate (220) 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.

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

THashItemList is an array type, primarily used to be able to define the PHashItemList (201) type. It's used in the TFPHashList (221) class.

```
Array = 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 (201) class.

```
THTCustomNodeClass = Class of THTCustomNode
```

THTCustomNodeClass was used by TFPCustomHashTable (214) 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 TFPDataHashTable (220) method.

```
TObjectIteratorCallback = procedure(Item: TObject; const Key: string
;
                                var Continue: Boolean)
```

TObjectIteratorCallback 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.

```
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 (246) 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 (246) 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.

```
TStringIteratorCallback = procedure(Item: string; const Key: string
;
                                var Continue: Boolean)
```

TStringIteratorCallback is the callback prototype for the Iterate (214) method. It is called for each element in the hash table, with the string. If Continue is set to false, the iteration stops.

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

TStringIteratorMethod is the callback prototype for the Iterate (214) method. It is called for each element in the hash table, with the string. If Continue is set to false, the iteration stops.

## 8.4 Procedures and functions

### 8.4.1 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 (214) hash class. It's Robert Sedgwick's "Algorithms in C" hash function.

Errors: None.

See also: TFPCustomHashTable (214)

## 8.5 TBucket

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

TBucket describes 1 bucket in the TCustomBucketList (211) class. It is a container for TBucketItem (204) records. It should never be used directly.

## 8.6 TBucketItem

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

TBucketItem is a record used for internal use in TCustomBucketList (211). It should not be necessary to use it directly.

## 8.7 THashItem

```
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.

## 8.8 EDuplicate

### 8.8.1 Description

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

See also: TFPCustomHashTable.Add ([214](#))

## 8.9 EKeyNotFound

### 8.9.1 Description

Exception raised when a key is not found.

See also: TFPCustomHashTable.Delete ([217](#))

## 8.10 TBucketList

### 8.10.1 Description

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

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

See also: TCustomBucketList ([211](#))

### 8.10.2 Method overview

Page	Method	Description
<a href="#">205</a>	Create	Create a new TBucketList instance.

### 8.10.3 TBucketList.Create

Synopsis: Create a new TBucketList instance.

Declaration: constructor Create (ABuckets: TBucketListSizes)

Visibility: public

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

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

See also: `TBucketListSizes` (201)

## 8.11 TClassList

### 8.11.1 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.

See also: `#rtl.classes.tlist` (??), `TComponentList` (208), `TObjectList` (253)

### 8.11.2 Method overview

Page	Method	Description
<a href="#">206</a>	<code>Add</code>	Add a new class pointer to the list.
<a href="#">207</a>	<code>Extract</code>	Extract a class pointer from the list.
<a href="#">207</a>	<code>First</code>	Return first non-nil class pointer
<a href="#">207</a>	<code>IndexOf</code>	Search for a class pointer in the list.
<a href="#">208</a>	<code>Insert</code>	Insert a new class pointer in the list.
<a href="#">208</a>	<code>Last</code>	Return last non- <code>Nil</code> class pointer
<a href="#">207</a>	<code>Remove</code>	Remove a class pointer from the list.

### 8.11.3 Property overview

Page	Properties	Access	Description
<a href="#">208</a>	<code>Items</code>	rw	Index based access to class pointers.

### 8.11.4 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` (207), `#rtl.classes.tlist.add` (??)

### 8.11.5 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` (207), `#rtl.classes.Tlist.Extract` (??)

### 8.11.6 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` (207), `#rtl.classes.Tlist.Remove` (??)

### 8.11.7 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` (??)

### 8.11.8 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` (208), `TClassList.Pack` (206)



### 8.11.9 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` (207), `TClassList.Pack` (206)

### 8.11.10 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` (206), `TClassList.Remove` (207)

### 8.11.11 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` (??)

## 8.12 TComponentList

### 8.12.1 Description

`TComponentList` is a `TObjectList` (253) 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` (253), `TComponentList` removes any `TComponent` from the list if the `TComponent` instance was freed externally. It uses the `FreeNotification` mechanism for this.

See also: `#rtl.classes.TList` (??), `TFPObjectList` (240), `TObjectList` (253), `TClassList` (206)

### 8.12.2 Method overview

Page	Method	Description
<a href="#">209</a>	Add	Add a component to the list.
<a href="#">209</a>	Destroy	Destroys the instance
<a href="#">209</a>	Extract	Remove a component from the list without destroying it.
<a href="#">210</a>	First	First non-nil instance in the list.
<a href="#">210</a>	IndexOf	Search for an instance in the list
<a href="#">211</a>	Insert	Insert a new component in the list
<a href="#">211</a>	Last	Last non-nil instance in the list.
<a href="#">210</a>	Remove	Remove a component from the list, possibly destroying it.

### 8.12.3 Property overview

Page	Properties	Access	Description
<a href="#">211</a>	Items	rw	Index-based access to the elements in the list.

### 8.12.4 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` ([253](#)), `#rtl.classes.TComponent` (??)

### 8.12.5 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` ([254](#))

### 8.12.6 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` (253) 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` (210), `TObjectList.Extract` (254)

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

**Errors:** None.

See also: `TComponentList.Extract` (209), `TObjectList.Remove` (255)

### 8.12.8 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` (255)

### 8.12.9 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` (211), `TObjectList.First` (256)

### 8.12.10 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` (210), `TObjectList.Last` (256)

### 8.12.11 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` (256), `TComponentList.Add` (209), `TComponentList.Remove` (210)

### 8.12.12 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` (257)

## 8.13 TCustomBucketList

### 8.13.1 Description

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

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

The `TCustomBucketList` class does not determine the number of buckets or the bucket hash mechanism, this must be done by descendent classes such as `TBucketList` (205). `TCustomBucketList` only takes care of storage and retrieval of items in the various buckets.

Because `TCustomBucketList` is an abstract class - it does not determine the number of buckets - one should never instantiate an instance of `TCustomBucketList`, but always use a descendent class such as `TCustomBucketList` (211).

See also: `TBucketList` (205)

### 8.13.2 Method overview

Page	Method	Description
<a href="#">213</a>	Add	Add an item to the list
<a href="#">213</a>	Assign	Assign one bucket list to another
<a href="#">212</a>	Clear	Clear the list
<a href="#">212</a>	Destroy	Frees the bucketlist from memory
<a href="#">213</a>	Exists	Check if an item exists in the list.
<a href="#">213</a>	Find	Find an item in the list
<a href="#">214</a>	ForEach	Loop over all items.
<a href="#">214</a>	Remove	Remove an item from the list.

### 8.13.3 Property overview

Page	Properties	Access	Description
<a href="#">214</a>	Data	rw	Associative array for data pointers

### 8.13.4 TCustomBucketList.Destroy

Synopsis: Frees the bucketlist from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

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

### 8.13.5 TCustomBucketList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear`

Visibility: `public`

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

Errors: None.

See also: `TCustomBucketList.Add` (213)

### 8.13.6 TCustomBucketList.Add

Synopsis: Add an item to the list

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

Visibility: `public`

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

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

See also: `TCustomBucketList.Exists` (213), `TCustomBucketList.Clear` (212)

### 8.13.7 TCustomBucketList.Assign

Synopsis: Assign one bucket list to another

Declaration: `procedure Assign(AList: TCustomBucketList)`

Visibility: `public`

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

See also: `TCustomBucketList.Add` (213), `TCustomBucketList.Clear` (212)

### 8.13.8 TCustomBucketList.Exists

Synopsis: Check if an item exists in the list.

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

Visibility: `public`

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

If the data pointer associated with `AItem` is also needed, then it is better to use `Find` (213).

See also: `TCustomBucketList.Find` (213)

### 8.13.9 TCustomBucketList.Find

Synopsis: Find an item in the list

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

Visibility: `public`

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

See also: `TCustomBucketList.Exists` (213)

### 8.13.10 TCustomBucketList.ForEach

Synopsis: Loop over all items.

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

Visibility: public

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

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

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

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

Errors: None.

See also: TCustomBucketList.Data ([214](#))

### 8.13.11 TCustomBucketList.Remove

Synopsis: Remove an item from the list.

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

Visibility: public

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

See also: Find ([213](#))

### 8.13.12 TCustomBucketList.Data

Synopsis: Associative array for data pointers

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

Visibility: public

Access: Read,Write

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

See also: TCustomBucketList.Find ([213](#)), TCustomBucketList.Exists ([213](#))

## 8.14 TFPCustomHashTable

### 8.14.1 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` (204).

The `TFPHasList` (200) can also be used when fast lookup of data based on some key is required. It is slightly faster than the `TFPCustomHashTable` implementation, but the keys are limited to a length of 256 characters, and it is not suitable for re-use: it is a one-time fill, many times search object. `TFPCustomHashTable` is slower, but handles re-use better.

See also: `THTCustomNode` (249), `TFPObjectList` (240), `RSHash` (204)

### 8.14.2 Method overview

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

### 8.14.3 Property overview

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

### 8.14.4 `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` (204)

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

See also: `CreateWith` (216)



### 8.14.5 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: `Create` ([215](#))

### 8.14.6 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: `Destroy` ([216](#)), `Create` ([215](#)), `Create` ([216](#)), `THTCustomNode.Data` ([249](#))

### 8.14.7 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: `HashTableSize` ([218](#))

### 8.14.8 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: `Destroy` ([216](#))

### 8.14.9 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` (217), `TFPCustomHashTable.Add` (214)

### 8.14.10 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` (249) 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` (217) method should be used instead.

Errors: None.

See also: `Add` (214), `Delete` (217)

### 8.14.11 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.

See also: `TFPCustomHashTable.Count` (218), `TFPCustomHashTable.HashTableSize` (218), `TFPCustomHashTable.AVGChainLen` (219), `TFPCustomHashTable.MaxChainLength` (219)

### 8.14.12 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` (214). 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: `SetHashFunction` (214), `HashTableSize` (218)

### 8.14.13 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` (217), `TFPCustomHashTable.HashTableSize` (218), `TFPCustomHashTable.AVGChainLen` (219), `TFPCustomHashTable.MaxChainLength` (219)

### 8.14.14 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` (217), `TFPCustomHashTable.Count` (218), `TFPCustomHashTable.AVGChainLen` (219), `TFPCustomHashTable.MaxChainLength` (219), `TFPCustomHashTable.VoidSlots` (218), `TFPCustomHashTable.Density` (220)

### 8.14.15 TFPCustomHashTable.HashTable

Synopsis: Hash table instance

Declaration: `Property HashTable : TFPObjectList`

Visibility: public

Access: Read

Description: `TFPCustomHashTable` is the internal list object (`TFPObjectList` (240)) used for the hash table. Each element in this table is again a `TFPObjectList` (240) instance or `Nil`.

### 8.14.16 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` (217), `TFPCustomHashTable.Count` (218), `TFPCustomHashTable.AVGChainLen` (219), `TFPCustomHashTable.MaxChainLength` (219), `TFPCustomHashTable.LoadFactor` (219), `TFPCustomHashTable.Density` (220), `TFPCustomHashTable.NumberOfCollisions` (219)

### 8.14.17 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` (217), `TFPCustomHashTable.Count` (218), `TFPCustomHashTable.AVGChainLen` (219), `TFPCustomHashTable.MaxChainLength` (219), `TFPCustomHashTable.VoidSlots` (218), `TFPCustomHashTable.Density` (220), `TFPCustomHashTable.NumberOfCollisions` (219)

### 8.14.18 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` (217), `TFPCustomHashTable.Count` (218), `TFPCustomHashTable.LoadFactor` (219), `TFPCustomHashTable.MaxChainLength` (219), `TFPCustomHashTable.VoidSlots` (218), `TFPCustomHashTable.Density` (220), `TFPCustomHashTable.NumberOfCollisions` (219)

### 8.14.19 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` (217), `TFPCustomHashTable.Count` (218), `TFPCustomHashTable.LoadFactor` (219), `TFPCustomHashTable.AvgChainLength` (214), `TFPCustomHashTable.VoidSlots` (218), `TFPCustomHashTable.Density` (220), `TFPCustomHashTable.NumberOfCollisions` (219)

### 8.14.20 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` (217), `TFPCustomHashTable.Count` (218), `TFPCustomHashTable.LoadFactor` (219), `TFPCustomHashTable.AvgChainLength` (214), `TFPCustomHashTable.VoidSlots` (218), `TFPCustomHashTable.Density` (220)

### 8.14.21 `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` (217), `TFPCustomHashTable.Count` (218), `TFPCustomHashTable.LoadFactor` (219), `TFPCustomHashTable.AvgChainLength` (214), `TFPCustomHashTable.VoidSlots` (218), `TFPCustomHashTable.Density` (220)

## 8.15 `TFPDataHashTable`

### 8.15.1 Description

`TFPDataHashTable` is a `TFPCustomHashTable` (214) 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` (238), or for string data, `TFPStringHashTable` (247) is more suitable. The data pointers are exposed with their keys through the `Items` (221) property.

See also: `TFPObjectHashTable` (238), `TFPStringHashTable` (247), `Items` (221)

### 8.15.2 Method overview

Page	Method	Description
<a href="#">221</a>	<code>Add</code>	Add a data pointer to the list.
<a href="#">220</a>	<code>Iterate</code>	Iterate over the pointers in the hash table

### 8.15.3 Property overview

Page	Properties	Access	Description
<a href="#">221</a>	<code>Items</code>	rw	Key-based access to the items in the table

### 8.15.4 `TFPDataHashTable.Iterate`

Synopsis: Iterate over the pointers in the hash table

**Declaration:** `function Iterate(aMethod: TDataIteratorMethod) : Pointer; Virtual`  
`function Iterate(aMethod: TDataIteratorCallback) : Pointer; Virtual`

Visibility: public

**Description:** `Iterate` iterates over all elements in the array, calling `aMethod` for each pointer, or until the method returns `False` in its `continue` parameter. It returns `Nil` if all elements were processed, or the pointer that was being processed when `aMethod` returned `False` in the `Continue` parameter. The `aMethod` callback can be a method of an object, or a normal, static procedure.

See also: `ForeachCall` ([200](#))

### 8.15.5 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` ([221](#))

### 8.15.6 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` ([248](#))

## 8.16 TFPHashList

### 8.16.1 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 non-nil pointers.

if a base class for an own hash class is wanted, the `TFPCustomHashTable` ([214](#)) class can be used. If a hash class for objects is needed instead of pointers, the `TFPHashObjectList` ([231](#)) class can be used.

See also: `TFPCustomHashTable` ([214](#)), `TFPHashObjectList` ([231](#)), `TFPDataHashTable` ([220](#)), `TFPStringHashTable` ([247](#))

### 8.16.2 Method overview

Page	Method	Description
<a href="#">223</a>	Add	Add a new key/data pair to the list
<a href="#">223</a>	Clear	Clear the list
<a href="#">222</a>	Create	Create a new instance of the hashlist
<a href="#">224</a>	Delete	Delete an item from the list.
<a href="#">222</a>	Destroy	Removes an instance of the hashlist from the heap
<a href="#">224</a>	Error	Raise an error
<a href="#">224</a>	Expand	Expand the list
<a href="#">224</a>	Extract	Extract a pointer from the list
<a href="#">225</a>	Find	Find data associated with key
<a href="#">225</a>	FindIndexOf	Return index of named item.
<a href="#">225</a>	FindWithHash	Find first element with given name and hash value
<a href="#">227</a>	ForEachCall	Call a procedure for each element in the list
<a href="#">224</a>	GetNextCollision	Get next collision number
<a href="#">223</a>	HashOfIndex	Return the hash value of an item by index
<a href="#">225</a>	IndexOf	Return the index of the data pointer
<a href="#">223</a>	NameOfIndex	Returns the key name of an item by index
<a href="#">226</a>	Pack	Remove nil pointers from the list
<a href="#">226</a>	Remove	Remove first instance of a pointer
<a href="#">226</a>	Rename	Rename a key
<a href="#">226</a>	ShowStatistics	Return some statistics for the list.

### 8.16.3 Property overview

Page	Properties	Access	Description
<a href="#">227</a>	Capacity	rw	Capacity of the list.
<a href="#">227</a>	Count	rw	Current number of elements in the list.
<a href="#">227</a>	Items	rw	Indexed array with pointers
<a href="#">228</a>	List	r	Low-level hash list
<a href="#">228</a>	Strs	r	Low-level memory area with strings.

### 8.16.4 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` ([222](#))

### 8.16.5 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 \(222\)](#), [TFPHashList.Clear \(223\)](#)

### 8.16.6 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.

See also: [TFPHashList.Extract \(224\)](#), [TFPHashList.Remove \(226\)](#), [TFPHashList.Delete \(224\)](#)

### 8.16.7 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 \(224\)](#), [TFPHashList.Remove \(226\)](#), [TFPHashList.Delete \(224\)](#), [TFPHashList.Add \(223\)](#)

### 8.16.8 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 \(223\)](#), [TFPHashList.Find \(225\)](#), [TFPHashList.FindIndexOf \(225\)](#), [TFPHashList.FindWithHash \(225\)](#)

### 8.16.9 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 \(221\)](#), [TFPHashList.Find \(225\)](#), [TFPHashList.FindIndexOf \(225\)](#), [TFPHashList.FindWithHash \(225\)](#)



### 8.16.10 TFPHashList.GetNextCollision

Synopsis: Get next collision number

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

Visibility: public

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

### 8.16.11 TFPHashList.Delete

Synopsis: Delete an item from the list.

Declaration: `procedure Delete(Index: Integer)`

Visibility: public

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

Errors: `TFPHashList.Extract` (224)`TFPHashList.Remove` (226)`TFPHashList.Add` (223)

### 8.16.12 TFPHashList.Error

Synopsis: Raise an error

Declaration: `class 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.

### 8.16.13 TFPHashList.Expand

Synopsis: Expand the list

Declaration: `function Expand : TFPHashList`

Visibility: public

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

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

See also: `TFPHashList.Clear` (223)

### 8.16.14 TFPHashList.Extract

Synopsis: Extract a pointer from the list

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

Visibility: public

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

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

See also: `TFPHashList.Delete` (224), `TFPHashList.Remove` (226), `TFPHashList.Clear` (223)

**8.16.15 TFPHashList.IndexOf**

Synopsis: Return the index of the data pointer

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

Visibility: public

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

The performed search is linear, and not very efficient.

See also: `TFPHashList.HashOfIndex` (223), `TFPHashList.NameOfIndex` (223), `TFPHashList.Find` (225), `TFPHashList.FindIndexOf` (225), `TFPHashList.FindWithHash` (225)

**8.16.16 TFPHashList.Find**

Synopsis: Find data associated with key

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

Visibility: public

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

See also: `TFPHashList.HashOfIndex` (223), `TFPHashList.NameOfIndex` (223), `TFPHashList.IndexOf` (225), `TFPHashList.FindIndexOf` (225), `TFPHashList.FindWithHash` (225)

**8.16.17 TFPHashList.FindIndexOf**

Synopsis: Return index of named item.

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

Visibility: public

Description: `FindIndexOf` returns the index of the key `AName`, or -1 if the key does not exist in the list. It uses the hash value to search for the key. Note that `Nil` data pointers will result in -1 as well.

See also: `TFPHashList.HashOfIndex` (223), `TFPHashList.NameOfIndex` (223), `TFPHashList.IndexOf` (225), `TFPHashList.Find` (225), `TFPHashList.FindWithHash` (225)

**8.16.18 TFPHashList.FindWithHash**

Synopsis: Find first element with given name and hash value

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

Visibility: public

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

See also: `TFPHashList.HashOfIndex` (223), `TFPHashList.NameOfIndex` (223), `TFPHashList.IndexOf` (225), `TFPHashList.Find` (225), `TFPHashList.FindIndexOf` (225)

### 8.16.19 TFPHashList.Rename

Synopsis: Rename a key

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

Visibility: public

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

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

### 8.16.20 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` (224), `TFPHashList.Clear` (223), `TFPHashList.Extract` (224)

### 8.16.21 TFPHashList.Pack

Synopsis: Remove nil pointers from the list

Declaration: `procedure Pack`

Visibility: public

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

See also: `TFPHashList.Clear` (223)

### 8.16.22 TFPHashList.ShowStatistics

Synopsis: Return some statistics for the list.

Declaration: `procedure ShowStatistics`

Visibility: public

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

**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

**8.16.23 TFPHashList.ForEachCall**

Synopsis: Call a procedure for each element in the list

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

Visibility: public

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

**8.16.24 TFPHashList.Capacity**

Synopsis: Capacity of the list.

Declaration: `Property Capacity : Integer`

Visibility: public

Access: Read,Write

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

See also: [Count \(227\)](#), [Items \(227\)](#)

**8.16.25 TFPHashList.Count**

Synopsis: Current number of elements in the list.

Declaration: `Property Count : Integer`

Visibility: public

Access: Read,Write

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

See also: [Capacity \(227\)](#), [Items \(227\)](#)

**8.16.26 TFPHashList.Items**

Synopsis: Indexed array with pointers

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

Visibility: public

Access: Read,Write

Description: `Items` provides indexed access to the pointers, the index runs from 0 to `Count-1` ([227](#)).

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

See also: [Capacity \(227\)](#), [Count \(227\)](#)

### 8.16.27 TFPHashList.List

Synopsis: Low-level hash list

Declaration: `Property List : PHashItemList`

Visibility: public

Access: Read

Description: `List` exposes the low-level item list (202). It should not be used directly.

See also: `Strs` (228), `THashItemList` (202)

### 8.16.28 TFPHashList.Strs

Synopsis: Low-level memory area with strings.

Declaration: `Property Strs : PChar`

Visibility: public

Access: Read

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

See also: `List` (228)

## 8.17 TFPHashObject

### 8.17.1 Description

`TFPHashObject` is a `TObject` descendent which is aware of the `TFPHashObjectList` (231) 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.

See also: `TFPHashObject.Name` (230), `TFPHashObject.ChangeOwner` (229), `TFPHashObject.ChangeOwnerAndName` (229)

### 8.17.2 Method overview

Page	Method	Description
229	<code>ChangeOwner</code>	Change the list owning the object.
229	<code>ChangeOwnerAndName</code>	Simultaneously change the list owning the object and the name of the object.
229	<code>Create</code>	Create a named instance, and insert in a hash list.
229	<code>CreateNotOwned</code>	Create an instance not owned by any list.
230	<code>Rename</code>	Rename the object

### 8.17.3 Property overview

Page	Properties	Access	Description
230	<code>Hash</code>	r	Hash value
230	<code>Name</code>	r	Current name of the object

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

See also: `TFPHashObject.Name` (230), `TFPHashObject.ChangeOwner` (229), `TFPHashObject.ChangeOwnerAndName` (229)

### 8.17.5 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` (231).

See also: `CreateNotOwned` (229), `TFPHashObject.ChangeOwner` (229), `TFPHashObject.Name` (230)

### 8.17.6 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: `ChangeOwnerAndName` (229), `Name` (230)

### 8.17.7 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: `ChangeOwner` (229), `Name` (230)

### 8.17.8 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: `ChangeOwner` ([229](#)), `ChangeOwnerAndName` ([229](#)), `Name` ([230](#))

### 8.17.9 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: `Rename` ([230](#)), `ChangeOwnerAndName` ([229](#))

### 8.17.10 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: `Name` ([230](#))

## 8.18 TFPHashObjectList

### 8.18.1 Method overview

Page	Method	Description
<a href="#">232</a>	Add	Add a new key/data pair to the list
<a href="#">232</a>	Clear	Clear the list
<a href="#">231</a>	Create	Create a new instance of the hashlist
<a href="#">233</a>	Delete	Delete an object from the list.
<a href="#">231</a>	Destroy	Removes an instance of the hashlist from the heap
<a href="#">233</a>	Expand	Expand the list
<a href="#">234</a>	Extract	Extract a object instance from the list
<a href="#">234</a>	Find	Find data associated with key
<a href="#">235</a>	FindIndexOf	Return index of named object.
<a href="#">235</a>	FindInstanceOf	Search an instance of a certain class
<a href="#">235</a>	FindWithHash	Find first element with given name and hash value
<a href="#">236</a>	ForEachCall	Call a procedure for each object in the list
<a href="#">233</a>	GetNextCollision	Get next collision number
<a href="#">233</a>	HashOfIndex	Return the hash value of an object by index
<a href="#">234</a>	IndexOf	Return the index of the object instance
<a href="#">232</a>	NameOfIndex	Returns the key name of an object by index
<a href="#">236</a>	Pack	Remove nil object instances from the list
<a href="#">234</a>	Remove	Remove first occurrence of a object instance
<a href="#">235</a>	Rename	Rename a key
<a href="#">236</a>	ShowStatistics	Return some statistics for the list.

### 8.18.2 Property overview

Page	Properties	Access	Description
<a href="#">236</a>	Capacity	rw	Capacity of the list.
<a href="#">237</a>	Count	rw	Current number of elements in the list.
<a href="#">237</a>	Items	rw	Indexed array with object instances
<a href="#">237</a>	List	r	Low-level hash list
<a href="#">237</a>	OwnsObjects	rw	Does the list own the objects it contains

### 8.18.3 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` ([231](#)), `TFPHashObjectList.OwnsObjects` ([237](#))

### 8.18.4 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` (231), `TFPHashObjectList.Clear` (232)

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

**Errors:** None.

See also: `TFPHashObjectList.Extract` (234), `TFPHashObjectList.Remove` (234), `TFPHashObjectList.Delete` (233), `TFPHashObjectList.Add` (232)

### 8.18.6 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` (234), `TFPHashObjectList.Remove` (234), `TFPHashObjectList.Delete` (233)

### 8.18.7 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` (233), `TFPHashObjectList.Find` (234), `TFPHashObjectList.FindIndexOf` (235), `TFPHashObjectList.FindWithHash` (235)

### 8.18.8 TFPHashObjectList.HashOfIndex

Synopsis: Return the hash value 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` (231), `TFPHashObjectList.Find` (234), `TFPHashObjectList.FindIndexOf` (235), `TFPHashObjectList.FindWithHash` (235)

### 8.18.9 TFPHashObjectList.GetNextCollision

Synopsis: Get next collision number

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

Visibility: public

Description: Get next collision number

### 8.18.10 TFPHashObjectList.Delete

Synopsis: Delete an object from the list.

Declaration: `procedure Delete(Index: Integer)`

Visibility: public

Description: `Delete` deletes the object at position `Index`. If `OwnsObjects` (237) is `True`, then the object itself is also freed from memory.

See also: `TFPHashObjectList.Extract` (234), `TFPHashObjectList.Remove` (234), `TFPHashObjectList.Add` (232), `OwnsObjects` (237)

### 8.18.11 TFPHashObjectList.Expand

Synopsis: Expand the list

Declaration: `function Expand : TFPHashObjectList`

Visibility: public

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

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

See also: `TFPHashObjectList.Clear` (232)

### 8.18.12 TFPHashObjectList.Extract

Synopsis: Extract a object instance from the list

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

Visibility: public

Description: `Extract` removes the data object from the list, if it is in the list. It returns the object instance if it was removed from the list, `Nil` otherwise. The object is *not* freed from memory, regardless of the value of `OwnsObjects` (237).

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

See also: `TFPHashObjectList.Delete` (233), `TFPHashObjectList.Remove` (234), `TFPHashObjectList.Clear` (232)

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

See also: `TFPHashObjectList.Delete` (233), `TFPHashObjectList.Clear` (232), `TFPHashObjectList.Extract` (234)

### 8.18.14 TFPHashObjectList.IndexOf

Synopsis: Return the index of the object instance

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

Visibility: public

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

The performed search is linear, and not very efficient.

See also: `TFPHashObjectList.HashOfIndex` (233), `TFPHashObjectList.NameOfIndex` (232), `TFPHashObjectList.Find` (234), `TFPHashObjectList.FindIndexOf` (235), `TFPHashObjectList.FindWithHash` (235)

### 8.18.15 TFPHashObjectList.Find

Synopsis: Find data associated with key

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

Visibility: public

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

See also: `TFPHashObjectList.HashOfIndex` (233), `TFPHashObjectList.NameOfIndex` (232), `TFPHashObjectList.IndexOf` (234), `TFPHashObjectList.FindIndexOf` (235), `TFPHashObjectList.FindWithHash` (235)

**8.18.16 TFPHashObjectList.FindIndexOf**

Synopsis: Return index of named object.

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

Visibility: public

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

See also: `TFPHashObjectList.HashOfIndex` (233), `TFPHashObjectList.NameOfIndex` (232), `TFPHashObjectList.IndexOf` (234), `TFPHashObjectList.Find` (234), `TFPHashObjectList.FindWithHash` (235)

**8.18.17 TFPHashObjectList.FindWithHash**

Synopsis: Find first element with given name and hash value

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

Visibility: public

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

See also: `TFPHashObjectList.HashOfIndex` (233), `TFPHashObjectList.NameOfIndex` (232), `TFPHashObjectList.IndexOf` (234), `TFPHashObjectList.Find` (234), `TFPHashObjectList.FindIndexOf` (235)

**8.18.18 TFPHashObjectList.Rename**

Synopsis: Rename a key

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

Visibility: public

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

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

**8.18.19 TFPHashObjectList.FindInstanceOf**

Synopsis: Search an instance of a certain class

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

Visibility: public

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

### 8.18.20 TFPHashObjectList.Pack

Synopsis: Remove nil object instances from the list

Declaration: `procedure Pack`

Visibility: public

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

See also: `TFPHashObjectList.Clear` ([232](#))

### 8.18.21 TFPHashObjectList.ShowStatistics

Synopsis: Return some statistics for the list.

Declaration: `procedure ShowStatistics`

Visibility: public

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

**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

### 8.18.22 TFPHashObjectList.ForEachCall

Synopsis: Call a procedure for each object in the list

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

Visibility: public

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

### 8.18.23 TFPHashObjectList.Capacity

Synopsis: Capacity of the list.

Declaration: `Property Capacity : Integer`

Visibility: public

Access: Read,Write

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

See also: `Count` ([237](#)), `Items` ([237](#))

### 8.18.24 TFPHashObjectList.Count

Synopsis: Current number of elements in the list.

Declaration: `Property Count : Integer`

Visibility: `public`

Access: `Read,Write`

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

See also: [Capacity \(236\)](#), [Items \(237\)](#)

### 8.18.25 TFPHashObjectList.OwnsObjects

Synopsis: Does the list own the objects it contains

Declaration: `Property OwnsObjects : Boolean`

Visibility: `public`

Access: `Read,Write`

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

The value of `OwnsObjects` is set when the hash list is created, and may not be changed during the lifetime of the hash list. (The property is made read-only in versions later than 3.0 of Free Pascal).

See also: [TFPHashObjectList.Create \(231\)](#)

### 8.18.26 TFPHashObjectList.Items

Synopsis: Indexed array with object instances

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

Visibility: `public`

Access: `Read,Write`

Description: `Items` provides indexed access to the object instances, the index runs from 0 to `Count-1` ([237](#)).

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

See also: [Capacity \(236\)](#), [Count \(237\)](#)

### 8.18.27 TFPHashObjectList.List

Synopsis: Low-level hash list

Declaration: `Property List : TFPHashList`

Visibility: `public`

Access: `Read`

Description: `List` exposes the low-level hash list ([221](#)). It should not be used directly.

See also: [TFPHashList \(221\)](#)

## 8.19 TFPObjectHashTable

### 8.19.1 Description

TFPStringHashTable is a TFPCustomHashTable (214) 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 (247), or for arbitrary pointer data, TFPDataHashTable (220) is more suitable. The objects are exposed with their keys through the Items (239) property.

See also: TFPStringHashTable (247), TFPDataHashTable (220), TFPObjectHashTable.Items (239)

### 8.19.2 Method overview

Page	Method	Description
<a href="#">239</a>	Add	Add a new object to the hash table
<a href="#">238</a>	Create	Create a new instance of TFPObjectHashTable
<a href="#">238</a>	CreateWith	Create a new hash table with given size and hash function
<a href="#">239</a>	Iterate	Iterate over the objects in the hash table

### 8.19.3 Property overview

Page	Properties	Access	Description
<a href="#">239</a>	Items	rw	Key-based access to the objects
<a href="#">240</a>	OwnsObjects	r	Does the hash table own the objects ?

### 8.19.4 TFPObjectHashTable.Create

Synopsis: Create a new instance of TFPObjectHashTable

Declaration: constructor Create(AOwnsObjects: Boolean)

Visibility: public

Description: Create creates a new instance of TFPObjectHashTable on the heap. It sets the OwnsObjects (240) property to AOwnsObjects, and then calls the inherited Create. If AOwnsObjects is set to True, then the hash table owns the objects: whenever an object is removed from the list, it is automatically freed.

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

See also: TFPObjectHashTable.OwnsObjects (240), TFPObjectHashTable.CreateWith (238), TFPObjectHashTable.Items (239)

### 8.19.5 TFPObjectHashTable.CreateWith

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

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

Visibility: public

Description: CreateWith sets the OwnsObjects (240) property to AOwnsObjects, and then calls the inherited CreateWith. If AOwnsObjects is set to True, then the hash table owns the objects: whenever an object is removed from the list, it is automatically freed.

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

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

See also: `TFPObjectHashTable.OwnsObjects` ([240](#)), `TFPObjectHashTable.Create` ([238](#)), `TFPObjectHashTable.Items` ([239](#))

### 8.19.6 TFPObjectHashTable.Iterate

Synopsis: Iterate over the objects in the hash table

Declaration: `function Iterate(aMethod: TObjectIteratorMethod) : TObject; Virtual`  
`function Iterate(aMethod: TObjectIteratorCallback) : TObject; Virtual`

Visibility: public

Description: `Iterate` iterates over all elements in the array, calling `aMethod` for each object, or until the method returns `False` in its `continue` parameter. It returns `Nil` if all elements were processed, or the object that was being processed when `aMethod` returned `False` in the `Continue` parameter.

See also: `ForeachCall` ([200](#))

### 8.19.7 TFPObjectHashTable.Add

Synopsis: Add a new object to the hash table

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

Visibility: public

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

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

See also: `TFPObjectHashTable.Items` ([239](#))

### 8.19.8 TFPObjectHashTable.Items

Synopsis: Key-based access to the objects

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

Visibility: public

Access: Read,Write

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

See also: `TFPObjectHashTable.Add` ([239](#))



### 8.19.9 TFObjectHashTable.OwnsObjects

Synopsis: Does the hash table own the objects ?

Declaration: `Property OwnsObjects : Boolean`

Visibility: `public`

Access: `Read`

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` (238), `TFObjectHashTable.Items` (239)

## 8.20 TFObjectList

### 8.20.1 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` (253), `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).

See also: `#rtl.classes.TFPList` (??), `TObjectList` (253)

### 8.20.2 Method overview

Page	Method	Description
<a href="#">241</a>	<code>Add</code>	Add an object to the list.
<a href="#">245</a>	<code>Assign</code>	Copy the contents of a list.
<a href="#">241</a>	<code>Clear</code>	Clear all elements in the list.
<a href="#">241</a>	<code>Create</code>	Create a new object list
<a href="#">242</a>	<code>Delete</code>	Delete an element from the list.
<a href="#">241</a>	<code>Destroy</code>	Clears the list and destroys the list instance
<a href="#">242</a>	<code>Exchange</code>	Exchange the location of two objects
<a href="#">242</a>	<code>Expand</code>	Expand the capacity of the list.
<a href="#">243</a>	<code>Extract</code>	Extract an object from the list
<a href="#">243</a>	<code>FindInstanceOf</code>	Search for an instance of a certain class
<a href="#">244</a>	<code>First</code>	Return the first non-nil object in the list
<a href="#">246</a>	<code>ForEachCall</code>	For each object in the list, call a method or procedure, passing it the object.
<a href="#">243</a>	<code>IndexOf</code>	Search for an object in the list
<a href="#">244</a>	<code>Insert</code>	Insert a new object in the list
<a href="#">244</a>	<code>Last</code>	Return the last non-nil object in the list.
<a href="#">245</a>	<code>Move</code>	Move an object to another location in the list.
<a href="#">245</a>	<code>Pack</code>	Remove all <code>Nil</code> references from the list
<a href="#">243</a>	<code>Remove</code>	Remove an item from the list.
<a href="#">245</a>	<code>Sort</code>	Sort the list of objects

### 8.20.3 Property overview

Page	Properties	Access	Description
<a href="#">246</a>	Capacity	rw	Capacity of the list
<a href="#">246</a>	Count	rw	Number of elements in the list.
<a href="#">247</a>	Items	rw	Indexed access to the elements of the list.
<a href="#">247</a>	List	r	Internal list used to keep the objects.
<a href="#">247</a>	OwnsObjects	rw	Should the list free elements when they are removed.

### 8.20.4 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` ([241](#)), `TFObjectList.OwnsObjects` ([247](#)), `TObjectList` ([253](#))

### 8.20.5 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` ([247](#)) is `True`.

See also: `TFObjectList.OwnsObjects` ([247](#)), `TObjectList.Create` ([254](#))

### 8.20.6 TFObjectList.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` ([247](#)) is `True`.

See also: `TObjectList.Destroy` ([253](#))

### 8.20.7 TFObjectList.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` (247) 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: `TFObjectList.OwnsObjects` (247), `TFObjectList.Delete` (242)

### 8.20.8 TFObjectList.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` (247) is `True`, the object is also freed.

**Errors:** An access violation may occur when `OwnsObjects` (247) is `True` and either the object was freed externally, or when the same object is in the same list twice.

See also: `TFObjectList.Remove` (243), `TFObjectList.Extract` (243), `TFObjectList.OwnsObjects` (247), `TFObjectList.Add` (241), `TFObjectList.Clear` (241)

### 8.20.9 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: `TFObjectList.Add` (241), `TFObjectList.Delete` (242)

### 8.20.10 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` (245), `TFObjectList.Clear` (241), `#rtl.classes.tfplist.expand` (??)

### 8.20.11 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` (245), `TFObjectList.Clear` (241), `TFObjectList.Remove` (243), `TFObjectList.Delete` (242)

### 8.20.12 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` (247) 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: `TFObjectList.Pack` (245), `TFObjectList.Clear` (241), `TFObjectList.Delete` (242), `TFObjectList.Extract` (243)

### 8.20.13 TFObjectList.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: `TFObjectList.Items` (247), `TFObjectList.Remove` (243), `TFObjectList.Extract` (243)

### 8.20.14 TFObjectList.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: `TFObjectList.IndexOf` ([243](#))

### 8.20.15 TFObjectList.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: `TFObjectList.Add` ([241](#)), `TFObjectList.Delete` ([242](#))

### 8.20.16 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` ([244](#)), `TFObjectList.Pack` ([245](#))

### 8.20.17 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` ([244](#)), `TFObjectList.Pack` ([245](#))

### 8.20.18 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 (242), 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` (242), `TFObjectList.Delete` (242), `TFObjectList.Insert` (244)

### 8.20.19 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.

### 8.20.20 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` (244), `TFObjectList.Last` (244)

### 8.20.21 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` (??)

### 8.20.22 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` (203), `TObjectListCallback` (203)

### 8.20.23 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` (246).

See also: `TFObjectList.Count` (246)

### 8.20.24 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` (246)

### 8.20.25 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 (241), TFObjectList.Delete (242), TFObjectList.Remove (243), TFObjectList.Clear (241)

### 8.20.26 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 (246)

### 8.20.27 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 (??)

## 8.21 TFPStringHashTable

### 8.21.1 Description

TFPStringHashTable is a TFPCustomHashTable (214) descendent which stores simple strings together with the keys. In case the data associated with the keys are objects, it's better to use TFObjectHashTable (238), or for arbitrary pointer data, TFpDataHashTable (220) is more suitable. The strings are exposed with their keys through the Items (248) property.

See also: TFObjectHashTable (238), TFPDataHashTable (220), Items (248)



### 8.21.2 Method overview

Page	Method	Description
<a href="#">248</a>	Add	Add a new string to the hash list
<a href="#">248</a>	Iterate	Iterate over the strings in the hash table

### 8.21.3 Property overview

Page	Properties	Access	Description
<a href="#">248</a>	Items	rw	Key based access to the strings in the hash table

### 8.21.4 TFPStringHashTable.Iterate

Synopsis: Iterate over the strings in the hash table

Declaration: `function Iterate(aMethod: TStringIteratorMethod) : string; Virtual`  
`function Iterate(aMethod: TStringIteratorCallback) : string; Virtual`

Visibility: public

Description: `Iterate` iterates over all elements in the array, calling `aMethod` for each string, or until the method returns `False` in its `continue` parameter. It returns an empty string if all elements were processed, or the string that was being processed when `aMethod` returned `False` in the `Continue` parameter.

See also: `ForeachCall` ([200](#))

### 8.21.5 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` ([248](#))

### 8.21.6 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` ([248](#))

## 8.22 THTCustomNode

### 8.22.1 Description

THTCustomNode is used by the TFPCustomHashTable (214) class to store the keys and associated values.

See also: TFPCustomHashTable (214)

### 8.22.2 Method overview

Page	Method	Description
<a href="#">249</a>	CreateWith	Create a new instance of THTCustomNode
<a href="#">249</a>	HasKey	Check whether this node matches the given key.

### 8.22.3 Property overview

Page	Properties	Access	Description
<a href="#">250</a>	Key	r	Key value associated with this hash item.

### 8.22.4 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 TFPCustomHashTable (214) class when needed.

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

See also: TFPCustomHashTable (214)

### 8.22.5 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 (250)

### 8.22.6 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` ([249](#))

## 8.23 THTDataNode

### 8.23.1 Description

`THTDataNode` is used by `TFPDataHashTable` ([220](#)) 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`

See also: `TFPDataHashTable` ([220](#)), `THTObjectNode` ([250](#)), `THTStringNode` ([251](#))

### 8.23.2 Property overview

Page	Properties	Access	Description
<a href="#">250</a>	Data	rw	Data pointer

### 8.23.3 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.

## 8.24 THTObjectNode

### 8.24.1 Description

`THTObjectNode` is a `THTCustomNode` ([249](#)) descendent which holds the data in the `TFPObjectHashTable` ([238](#)) hash table. It exposes a data string.

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

See also: `TFPObjectHashTable` ([238](#))

### 8.24.2 Property overview

Page	Properties	Access	Description
<a href="#">251</a>	Data	rw	Object instance

### 8.24.3 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` ([239](#))

See also: `TFPObjectHashTable` ([238](#)), `TFPObjectHashTable.Items` ([239](#)), `THTOwnedObjectNode` ([251](#))

## 8.25 THTOwnedObjectNode

### 8.25.1 Description

`THTOwnedObjectNode` is used instead of `THTObjectNode` ([250](#)) in case `TFPObjectHashTable` ([238](#)) owns it's objects. When this object is destroyed, the associated data object is also destroyed.

See also: `TFPObjectHashTable` ([238](#)), `THTObjectNode` ([250](#)), `TFPObjectHashTable.OwnsObjects` ([240](#))

### 8.25.2 Method overview

Page	Method	Description
<a href="#">251</a>	Destroy	Destroys the node and the object.

### 8.25.3 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` ([251](#)), `TFPObjectHashTable.OwnsObjects` ([240](#))

## 8.26 THTStringNode

### 8.26.1 Description

`THTStringNode` is a `THTCustomNode` ([249](#)) descendent which holds the data in the `TFPStringHashTable` ([247](#)) hash table. It exposes a data string.

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

See also: `TFPStringHashTable` ([247](#))

### 8.26.2 Property overview

Page	Properties	Access	Description
<a href="#">252</a>	Data	rw	String data

### 8.26.3 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` ([248](#))

See also: `TFPStringHashTable` ([247](#))

## 8.27 TObjectBucketList

### 8.27.1 Description

`TObjectBucketList` is a class that redefines the associative `Data` array using `TObject` instead of `Pointer`. It also adds some overloaded versions of the `Add` and `Remove` calls using `TObject` instead of `Pointer` for the argument and result types.

See also: `TObjectBucketList` ([252](#))

### 8.27.2 Method overview

Page	Method	Description
<a href="#">252</a>	Add	Add an object to the list
<a href="#">253</a>	Remove	Remove an object from the list

### 8.27.3 Property overview

Page	Properties	Access	Description
<a href="#">253</a>	Data	rw	Associative array of data items

### 8.27.4 TObjectBucketList.Add

Synopsis: Add an object to the list

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

Visibility: `public`

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

See also: `TObjectBucketList.Data` ([253](#)), `TObjectBucketList.Remove` ([253](#))

### 8.27.5 TObjectBucketList.Remove

Synopsis: Remove an object from the list

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

Visibility: public

Description: Remove removes the object `AItem` from the list. It returns the `Data` object which was associated with the item. If `AItem` was not in the list, then `Nil` is returned.

See also: `TObjectBucketList.Add` (252), `TObjectBucketList.Data` (253)

### 8.27.6 TObjectBucketList.Data

Synopsis: Associative array of data items

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

Visibility: public

Access: Read,Write

Description: `Data` provides associative access to the data in the list: it returns the data object associated with the `AItem` object. If the `AItem` object is not in the list, an `EListError` exception is raised.

See also: `TObjectBucketList.Add` (252)

## 8.28 TObjectList

### 8.28.1 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` (240), `TObjectList` offers a notification mechanism of list change operations: insert, delete. This slows down bulk operations, so if the notifications are not needed, `TFPObjectList` may be more appropriate.

See also: `#rtl.classes.TList` (??), `TFPObjectList` (240), `TComponentList` (208), `TClassList` (206)

### 8.28.2 Method overview

Page	Method	Description
254	Add	Add an object to the list.
254	Create	Create a new object list.
254	Extract	Extract an object from the list.
255	FindInstanceOf	Search for an instance of a certain class
256	First	Return the first non-nil object in the list
255	IndexOf	Search for an object in the list
256	Insert	Insert an object in the list.
256	Last	Return the last non-nil object in the list.
255	Remove	Remove (and possibly free) an element from the list.

### 8.28.3 Property overview

Page	Properties	Access	Description
<a href="#">257</a>	Items	rw	Indexed access to the elements of the list.
<a href="#">256</a>	OwnsObjects	rw	Should the list free elements when they are removed.

### 8.28.4 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` ([253](#)), `TObjectList.OwnsObjects` ([256](#)), `TFPObjectList` ([240](#))

### 8.28.5 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` ([256](#)), `#rtl.classes.TList.Delete` (??), `TObjectList.Extract` ([254](#)), `TObjectList.Remove` ([255](#))

### 8.28.6 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` ([255](#)), `Extract` does not free the extracted element if `OwnsObjects` ([256](#)) 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` ([255](#))

### 8.28.7 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` (256) 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` (254)

### 8.28.8 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` (255)

### 8.28.9 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` (255)



### 8.28.10 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` (254), `TObjectList.Remove` (255)

### 8.28.11 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` (256), `TObjectList.Pack` (253)

### 8.28.12 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` (256), `TObjectList.Pack` (253)

### 8.28.13 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` (254), `TObjectList.Delete` (253), `TObjectList.Remove` (255), `TObjectList.Clear` (253)

### 8.28.14 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` (??)

## 8.29 TObjectQueue

### 8.29.1 Method overview

Page	Method	Description
<a href="#">258</a>	<code>Peek</code>	Look at the first object in the queue.
<a href="#">257</a>	<code>Pop</code>	Pop the first element off the queue
<a href="#">257</a>	<code>Push</code>	Push an object on the queue

### 8.29.2 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` (257), `TObjectQueue.Peek` (258)

### 8.29.3 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` (257), `TObjectQueue.Peek` (258)

### 8.29.4 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` (257), `TObjectQueue.Pop` (257)

## 8.30 TObjectStack

### 8.30.1 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.

See also: `TOrderedList` (259), `TStack` (261), `TQueue` (261), `TObjectQueue` (257)

### 8.30.2 Method overview

Page	Method	Description
<a href="#">259</a>	<code>Peek</code>	Look at the top object in the stack.
<a href="#">258</a>	<code>Pop</code>	Pop the top object of the stack.
<a href="#">258</a>	<code>Push</code>	Push an object on the stack.

### 8.30.3 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` (258), `TObjectStack.Peek` (259)

### 8.30.4 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` (258), `TObjectStack.Peek` (259)

### 8.30.5 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` (258), `TObjectStack.Pop` (258)

## 8.31 TOrderedList

### 8.31.1 Description

`TOrderedList` provides the base class for `TQueue` (261) and `TStack` (261). 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.

**See also:** `TQueue` (261), `TStack` (261)

### 8.31.2 Method overview

Page	Method	Description
260	<code>AtLeast</code>	Check whether the list contains a certain number of elements.
260	<code>Count</code>	Number of elements on the list.
259	<code>Create</code>	Create a new ordered list
260	<code>Destroy</code>	Free an ordered list
261	<code>Peek</code>	Return the next element to be popped from the list.
261	<code>Pop</code>	Remove an element from the list.
260	<code>Push</code>	Push another element on the list.

### 8.31.3 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` (260)

#### 8.31.4 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` ([259](#))

#### 8.31.5 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` ([260](#))

#### 8.31.6 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` ([260](#))

#### 8.31.7 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` ([261](#)), `TOrderedList.Peek` ([261](#))

### 8.31.8 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` (261), `TOrderedList.Push` (260)

### 8.31.9 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` (261), without actually popping it from the list.

Errors: None.

See also: `TOrderedList.Pop` (261), `TOrderedList.Push` (260)

## 8.32 TQueue

### 8.32.1 Description

`TQueue` is a descendent of `TOrderedList` (259) which implements `Push` (260) and `Pop` (261) 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` (259)

See also: `TOrderedList` (259), `TObjectQueue` (257), `TStack` (261)

## 8.33 TStack

### 8.33.1 Description

`TStack` is a descendent of `TOrderedList` (259) which implements `Push` (260) and `Pop` (261) 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` (259)

See also: `TOrderedList` (259), `TObjectStack` (258), `TQueue` (261)

## Chapter 9

# Reference for unit 'CustApp'

### 9.1 Used units

Table 9.1: Used units by unit 'CustApp'

Name	Page
Classes	??
singleinstance	??
sysutils	??

### 9.2 Overview

The `CustApp` unit implements the `TCustomApplication` (263) 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.

### 9.3 Constants, types and variables

#### 9.3.1 Types

```
TEventLogTypes = Set of TEventType
```

`TEventLogTypes` is a set of `TEventType` (??), used in `TCustomApplication.EventLogFilter` (274) to filter events that are sent to the system log.

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

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

```
Array = Array of string
```

`TStringArray` is an array of strings, used in the `TCustomApplication.GetOptionValues` (268) call.

### 9.3.2 Variables

```
CustomApplication : TCustomApplication = Nil
```

`CustomApplication` contains the global application instance. All descendents of `TCustomApplication` (263) should, in addition to storing an instance pointer in some variable (most likely called "Application") store the instance pointer in this variable. This ensures that, whatever kind of application is being created, user code can access the application object.

## 9.4 TCustomApplication

### 9.4.1 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` (265), `ShowException` (266), `OnException` (271) and `StopOnException` (274).
- Command-line parameter parsing in `FindOptionIndex` (267), `GetOptionValue` (267), `CheckOptions` (268) and `HasOption` (268)
- Environment variable handling in `GetEnvironmentList` (270) and `EnvironmentVariable` (273).

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



### 9.4.2 Method overview

Page	Method	Description
<a href="#">268</a>	CheckOptions	Check whether all given options on the command-line are valid.
<a href="#">264</a>	Create	Create a new instance of the <code>TCustomApplication</code> class
<a href="#">265</a>	Destroy	Destroys the <code>TCustomApplication</code> instance.
<a href="#">267</a>	FindOptionIndex	Return the index of an option.
<a href="#">270</a>	GetEnvironmentList	Return a list of environment variables.
<a href="#">269</a>	GetNonOptions	Get all non-switch options
<a href="#">267</a>	GetOptionValue	Return the value of a command-line option.
<a href="#">268</a>	GetOptionValues	Get the values for an option that may be specified multiple times
<a href="#">265</a>	HandleException	Handle an exception.
<a href="#">268</a>	HasOption	Check whether an option was specified.
<a href="#">265</a>	Initialize	Initialize the application
<a href="#">270</a>	Log	Write a message to the event log
<a href="#">266</a>	Run	Runs the application.
<a href="#">266</a>	ShowException	Show an exception to the user
<a href="#">266</a>	Terminate	Terminate the application.

### 9.4.3 Property overview

Page	Properties	Access	Description
<a href="#">273</a>	CaseSensitiveOptions	rw	Are options interpreted case sensitive or not
<a href="#">272</a>	ConsoleApplication	r	Is the application a console application or not
<a href="#">273</a>	EnvironmentVariable	r	Environment variable access
<a href="#">274</a>	EventLogFilter	rw	Event to filter events, before they are sent to the system log
<a href="#">274</a>	ExceptionExitCode	rw	ExitCode to use then terminating the program due to an exception
<a href="#">270</a>	ExeName	r	Name of the executable.
<a href="#">271</a>	HelpFile	rw	Location of the application help file.
<a href="#">272</a>	Location	r	Application location
<a href="#">271</a>	OnException	rw	Exception handling event
<a href="#">273</a>	OptionChar	rw	Command-line switch character
<a href="#">273</a>	ParamCount	r	Number of command-line parameters
<a href="#">272</a>	Params	r	Command-line parameters
<a href="#">275</a>	SingleInstance	r	Single instance used to control single application instance behaviour
<a href="#">275</a>	SingleInstanceClass	rw	Class to use when creating single instance
<a href="#">275</a>	SingleInstanceEnabled	rw	Enable single application instance control.
<a href="#">274</a>	StopOnException	rw	Should the program loop stop on an exception
<a href="#">271</a>	Terminated	r	Was <code>Terminate</code> called or not
<a href="#">271</a>	Title	rw	Application title

### 9.4.4 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 \(265\)](#)

### 9.4.5 TCustomApplication.Destroy

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

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

**Visibility:** `public`

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

See also: [TCustomApplication.Create \(264\)](#)

### 9.4.6 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 (271)` 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 (266)` 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 (274)` property is set to `True`, the `Terminated (271)` property is set to `True`, which will cause the `Run (266)` loop to stop, and the application will exit.

See also: [ShowException \(266\)](#), [StopOnException \(274\)](#), [Terminated \(271\)](#), [Run \(266\)](#)

### 9.4.7 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 \(266\)](#), [TCustomApplication.Terminated \(271\)](#)

### 9.4.8 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` (265). If `TCustomApplication.StopOnException` (274) 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` (265), `TCustomApplication.StopException` (263)

### 9.4.9 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: `ShowException` (??), `TCustomApplication.HandleException` (265), `TCustomApplication.StopException` (263)

### 9.4.10 TCustomApplication.Terminate

Synopsis: Terminate the application.

Declaration: `procedure Terminate; Virtual`  
`procedure Terminate(AExitCode: Integer); 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` (271) property and properly shut down the application if it is set to `True`.

When `AExitCode` is specified, it will be passed to `System.ExitCode` (262), and when the program is halted, that is the exit code of the program as returned to the OS. If the application is terminated due to an exception, `ExceptionExitCode` (274) will be used as the value for this argument.

See also: `TCustomApplication.Terminated` (271), `TCustomApplication.Run` (266), `ExceptionExitCode` (274), `System.ExitCode` (262)

### 9.4.11 TCustomApplication.FindOptionIndex

Synopsis: Return the index of an option.

Declaration: `function FindOptionIndex(const S: string; var Longopt: Boolean;  
StartAt: Integer) : 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` (273) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with `OptionChar` (273) (by default the dash ('-') character).

See also: `HasOption` (268), `GetOptionValue` (267), `CheckOptions` (268), `CaseSensitiveOptions` (273), `OptionChar` (273)

### 9.4.12 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` (273) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with `OptionChar` (273) (by default the dash ('-') character).

If an option can appear multiple times, use `TCustomApplication.GetOptionValues` (268) to retrieve all values. This function only returns the value of the first occurrence of an option.

See also: `FindOptionIndex` (267), `HasOption` (268), `CheckOptions` (268), `CaseSensitiveOptions` (273), `OptionChar` (273), `TCustomApplication.GetOptionValues` (268)

### 9.4.13 TCustomApplication.GetOptionValues

Synopsis: Get the values for an option that may be specified multiple times

Declaration: `function GetOptionValues(const C: Char; const S: string) : TStringArray`

Visibility: public

Description: `GetOptionValues` returns all values specified by command-line option switches C or S. For each occurrence of the command-line option C or S, the associated value is added to the array.

`TCustomApplication.GetOptionValue` (267) will only return the first occurrence of a value.

Errors: None.

See also: `TCustomApplication.GetOptionValue` (267)

### 9.4.14 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` (273) property, the search is performed case sensitive or case insensitive.

Options are identified as command-line parameters which start with `OptionChar` (273) (by default the dash ('-') character).

See also: `FindOptionIndex` (267), `GetOptionValue` (267), `CheckOptions` (268), `CaseSensitiveOptions` (273), `OptionChar` (273)

### 9.4.15 TCustomApplication.CheckOptions

Synopsis: Check whether all given options on the command-line are valid.

Declaration: `function CheckOptions(const ShortOptions: string;`  
`const Longopts: TStringList; Opts: TStringList;`  
`NonOpts: TStringList; AllErrors: Boolean) : string`  
`function CheckOptions(const ShortOptions: string;`  
`const Longopts: Array of string; Opts: TStringList;`  
`NonOpts: TStringList; AllErrors: Boolean) : string`  
`function CheckOptions(const ShortOptions: string;`  
`const Longopts: TStringList; AllErrors: Boolean)`  
`: string`  
`function CheckOptions(const ShortOptions: string;`  
`const Longopts: Array of string;`  
`AllErrors: Boolean) : string`  
`function CheckOptions(const ShortOptions: string;`  
`const Longopts: string; AllErrors: Boolean)`  
`: 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` (273) (by default the dash ('-') character).

if `AllErrors` is `True` then all errors are returned, separated by a `sLineBreak` (??) character.

**Errors:** If an error was found during the check, the return value is a string describing the error(s).

**See also:** `FindOptionIndex` (267), `GetOptionValue` (267), `HasOption` (268), `CaseSensitiveOptions` (273), `OptionChar` (273)

#### 9.4.16 TCustomApplication.GetNonOptions

**Synopsis:** Get all non-switch options

**Declaration:**

```
function GetNonOptions(const ShortOptions: string;
                      const Longopts: Array of string) : TStringArray
procedure GetNonOptions(const ShortOptions: string;
                      const Longopts: Array of string;
                      NonOptions: TStrings)
```

Visibility: public

**Description:** `GetNonOptions` returns the items on the command-line that are not associated with a switch. It checks the command-line for allowed switches as they are indicated by `ShortOptions` and `Longopts`. The format is identical to `TCustomApplication.Checkoptions` (268). This is useful for an application which accepts a command form such as `svn`:

```
svn commit [options] files
```

In the above example, "commit" and "files" would be returned by `GetNonOptions`

The non-options are returned in the form of a string array, or a stringlist instance can be passed in `NonOptions`. Either will be filled with the non-options on return.

**Errors:** None.

**See also:** `TCustomApplication.HasOption` (268), `TCustomApplication.Checkoptions` (268), `TCustomApplication.GetOptionValue` (267), `TCustomApplication.GetOptionValues` (268)

### 9.4.17 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: `EnvironmentVariable` ([273](#))

### 9.4.18 TCustomApplication.Log

Synopsis: Write a message to the event log

Declaration: `procedure Log(EventType: TEventType; const Msg: string)`  
`procedure Log(EventType: TEventType; const Fmt: string;`  
`const Args: Array of const)`

Visibility: public

Description: `Log` is meant for all applications to have a default logging mechanism. By default it does not do anything, descendent classes should override this method to provide appropriate logging: they should write the message `Msg` with type `EventType` to some log mechanism such as `#fcl.eventlog.TEventLog` ([524](#))

The second form using `Fmt` and `Args` will format the message using the provided arguments prior to logging it.

Errors: None.

See also: `#rtl.sysutils.TEventType` ([??](#))

### 9.4.19 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: `ParamStr` ([??](#))

#### 9.4.20 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` ([271](#))

#### 9.4.21 TCustomApplication.Terminated

Synopsis: Was `Terminate` called or not

Declaration: `Property Terminated : Boolean`

Visibility: `public`

Access: `Read`

Description: `Terminated` indicates whether `Terminate` ([266](#)) 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: `Terminate` ([266](#))

#### 9.4.22 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: `HelpFile` ([271](#))

#### 9.4.23 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` (263).

If set, `OnException` is called by the `HandleEvent` (263) routine. Do not use the `OnException` event directly, instead call `HandleEvent`

See also: `ShowEvent` (263)

#### 9.4.24 `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: `IsConsole` (??)

#### 9.4.25 `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 for example has no reliable way to get this information). See the discussion of `Paramstr` (??) in the RTL documentation.

See also: `Paramstr` (??), `Params` (272)

#### 9.4.26 `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` (273). It is equivalent to calling `ParamStr` (??).

See also: `ParamCount` (273), `Paramstr` (??)

### 9.4.27 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` (272) property.

See also: `Params` (272), `Paramstr` (??), `ParamCount` (??)

### 9.4.28 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` (270) can be used.

See also: `GetEnvironmentList` (270), `TCustomApplication.Params` (272)

### 9.4.29 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: `FindOptionIndex` (267), `GetOptionValue` (267), `HasOption` (268), `CaseSensitiveOptions` (273), `CheckOptions` (268)

### 9.4.30 TCustomApplication.CaseSensitiveOptions

Synopsis: Are options interpreted case sensitive or not

Declaration: `Property CaseSensitiveOptions : Boolean`

Visibility: public

Access: Read,Write

**Description:** `CaseSensitiveOptions` determines whether `FindOptionIndex` (267) and `CheckOptions` (268) 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: `FindOptionIndex` (267), `GetOptionValue` (267), `HasOption` (268), `OptionChar` (273), `CheckOptions` (268)

### 9.4.31 `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` (266) and `HandleException` (265) procedures in case of an unhandled exception in the `DoRun` code. If `StopOnException` is `True` then `Terminate` (266) will be called after the exception was handled.

See also: `Run` (266), `HandleException` (265), `Terminate` (266)

### 9.4.32 `TCustomApplication.ExceptionExitCode`

**Synopsis:** `ExitCode` to use then terminating the program due to an exception

**Declaration:** `Property ExceptionExitCode : LongInt`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `ExceptionExitCode` is the exit code that will be passed to `TCustomApplication.Terminate` (266)

### 9.4.33 `TCustomApplication.EventLogFilter`

**Synopsis:** Event to filter events, before they are sent to the system log

**Declaration:** `Property EventLogFilter : TEventLogTypes`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `EventLogFilter` can be set to a set of event types that should be logged to the system log. If the set is empty, all event types are sent to the system log. If the set is non-empty, the `TCustomApplication.Log` (270) routine will check if the log event type is in the set, and if not, will not send the message to the system log.

See also: `TCustomApplication.Log` (270)

#### 9.4.34 TCustomApplication.SingleInstance

Synopsis: Single instance used to control single application instance behaviour

Declaration: `Property SingleInstance : TBaseSingleInstance`

Visibility: public

Access: Read

Description: `SingleInstance` is used when `SingleInstanceEnabled` (262) is set to `True`. It can be used to send a message to an already running instance, or to check for messages if the current instance is the sole ("server") instance running.

See also: `TCustomApplication.SingleInstanceClass` (275), `TCustomApplication.SingleInstanceEnabled` (275)

#### 9.4.35 TCustomApplication.SingleInstanceClass

Synopsis: Class to use when creating single instance

Declaration: `Property SingleInstanceClass : TBaseSingleInstanceClass`

Visibility: public

Access: Read,Write

Description: `SingleInstanceClass` can be used to set the class used to instantiate `SingleInstance` (275). The default class is determined by the global `singleinstance` default class as specified in `singleinstance.DefaultSingleInstanceClass` (262).

See also: `TCustomApplication.SingleInstance` (275), `singleinstance.DefaultSingleInstanceClass` (262)

#### 9.4.36 TCustomApplication.SingleInstanceEnabled

Synopsis: Enable single application instance control.

Declaration: `Property SingleInstanceEnabled : Boolean`

Visibility: public

Access: Read,Write

Description: `SingleInstanceEnabled` can be set to `true` to start single-instance application control. This will instantiate `TCustomApplication.SingleInstance` (275) using `TCustomApplication.SingleInstanceClass` (275) and starts the check to see whether this application is a client or server instance.

See also: `TCustomApplication.SingleInstance` (275), `TCustomApplication.SingleInstanceClass` (275)

## Chapter 10

# Reference for unit 'daemonapp'

### 10.1 Used units

Table 10.1: Used units by unit 'daemonapp'

Name	Page
Classes	??
CustApp	<a href="#">262</a>
eventlog	<a href="#">522</a>
rtlconsts	??
sysutils	??

### 10.2 Overview

The `daemonapp` unit implements a `TApplication` class which encapsulates a daemon or service application. It handles installation where this is necessary, and does instantiation of the various daemons where necessary.

The unit consists of 3 separate classes which cooperate tightly:

**TDaemon** This is a class that implements the daemon's functionality. One or more descendents of this class can be implemented and instantiated in a single daemon application. For more information, see `TDaemon` ([293](#)).

**TDaemonApplication** This is the actual daemon application class. A global instance of this class is instantiated. It handles the command-line arguments, and instantiates the various daemons. For more information, see `TDaemonApplication` ([297](#)).

**TDaemonDef** This class defines the daemon in the operation system. The `TDaemonApplication` class has a collection of `TDaemonDef` instances, which it uses to start the various daemons. For more information, see `TDaemonDef` ([300](#)).

As can be seen, a single application can implement one ore more daemons (services). Each daemon will be run in a separate thread which is controlled by the application class.

The classes take care of logging through the `TEventLog` ([524](#)) class.

Many options are needed only to make the application behave as a windows service application on windows. These options are ignored in UNIX-like environment. The documentation will mention this.

## 10.3 Daemon application architecture

[Still needs to be completed]

## 10.4 Constants, types and variables

### 10.4.1 Resource strings

```
SControlFailed = 'Control code %s handling failed: %s'
```

The control code was not handled correctly

```
SCustomCode = '[Custom code %d]'
```

A custom code was received

```
SDaemonStatus = 'Daemon %s current status: %s'
```

Daemon status report log message

```
SErrApplicationAlreadyCreated =  
'An application instance of class %s was already created.'
```

A second application instance is created

```
SErrDaemonStartFailed = 'Failed to start daemon %s : %s'
```

The application failed to start the daemon

```
SErrDuplicateName = 'Duplicate daemon name: %s'
```

Duplicate service name

```
SErrNoDaemonDefForStatus =  
'%s: No daemon definition for status report'
```

Internal error: no daemon definition to report status for

```
SErrNoDaemonForStatus = '%s: No daemon for status report'
```

Internal error: no daemon to report status for

```
SErrNoServiceMapper = 'No daemon mapper class registered.'
```

No service mapper was found.

```
SErrNothingToDo = 'No command given, use ''%s -h'' for usage.'
```

No operation can be performed

```
SErrOnlyOneMapperAllowed =
    'Not changing daemon mapper class %s with %s: Only 1 mapper allowed.'
```

An attempt was made to install a second service mapper

```
SErrServiceManagerStartFailed =
    'Failed to start service manager: %s'
```

Unable to start or contact the service manager

```
SErrUnknownDaemonClass = 'Unknown daemon class name: %s'
```

Unknown daemon class requested

```
SErrWindowClass = 'Could not register window class'
```

Could not register window class

```
SHelpCommand = 'Where command is one of the following:'
```

Options message displayed when writing help to the console

```
SHelpInstall = 'To install the program as a service'
```

Install option message displayed when writing help to the console

```
SHelpRun = 'To run the service'
```

Run option message displayed when writing help to the console

```
SHelpUnInstall = 'To uninstall the service'
```

Uninstall option message displayed when writing help to the console

```
SHelpUsage = 'Usage: %s [command]'
```

Usage message displayed when writing help to the console

## 10.4.2 Types

```
TCurrentStatus = (csStopped, csStartPending, csStopPending, csRunning
    ,
    csContinuePending, csPausePending, csPaused)
```

Table 10.2: Enumeration values for type TCurrentStatus

Value	Explanation
csContinuePending	The daemon is continuing, but not yet running
csPaused	The daemon is paused: running but not active.
csPausePending	The daemon is about to be paused.
csRunning	The daemon is running (it is operational).
csStartPending	The daemon is starting, but not yet fully running.
csStopped	The daemon is stopped, i.e. inactive.
csStopPending	The daemon is stopping, but not yet fully stopped.

TCurrentStatus indicates the current state of the daemon. It changes from one state to the next during the time the instance is active. The daemon application changes the state of the daemon, depending on signals it gets from the operating system, by calling the appropriate methods.

```
TCustomControlCodeEvent = procedure(Sender: TCustomDaemon;
  ACode: DWord; var Handled: Boolean
)
                                of object
```

In case the system sends a non-standard control code to the daemon, an event handler is executed with this prototype.

```
TCustomDaemonApplicationClass = Class of TCustomDaemonApplication
```

Class pointer for TCustomDaemonApplication

```
TCustomDaemonClass = Class of TCustomDaemon
```

The class type is needed in the TDaemonDef (300) definition.

```
TCustomDaemonMapperClass = Class of TCustomDaemonMapper
```

TCustomDaemonMapperClass is the class of TCustomDaemonMapper. It is used in the RegisterDaemonMapper (282) call.

```
TDaemonClass = Class of TDaemon
```

Class type of TDaemon

```
TDaemonEvent = procedure(Sender: TCustomDaemon) of object
```

TDaemonEvent is used in event handling. The Sender is the TCustomDaemon (283) instance that has initiated the event.

```
TDaemonOKEvent = procedure(Sender: TCustomDaemon; var OK: Boolean
)
                                of object
```

TDaemonOKEvent is used in event handling, when a boolean result must be obtained, for instance, to see if an operation was performed successfully.

```
TDaemonOption = (doAllowStop, doAllowPause, doInteractive)
```

Table 10.3: Enumeration values for type TDaemonOption

Value	Explanation
doAllowPause	The daemon can be paused.
doAllowStop	The daemon can be stopped.
doInteractive	The daemon interacts with the desktop.

Enumerated that enumerates the various daemon operation options.



TDaemonOptions = Set of TDaemonOption

TDaemonOption enumerates the various options a daemon can have.

TDaemonRunMode = (drmUnknown, drmInstall, drmUninstall, drmRun)

Table 10.4: Enumeration values for type TDaemonRunMode

Value	Explanation
drmInstall	Daemon install mode (windows only)
drmRun	Daemon is running normally
drmUninstall	Daemon uninstall mode (windows only)
drmUnknown	Unknown mode

TDaemonRunMode indicates in what mode the daemon application (as a whole) is currently running.

TErrorSeverity = (esIgnore, esNormal, esSevere, esCritical)

Table 10.5: Enumeration values for type TErrorSeverity

Value	Explanation
esCritical	Error is logged, and startup is stopped if last known good configuration is active, or system is restarted using last known good configuration
esIgnore	Ignore startup errors
esNormal	Error is logged, but startup continues
esSevere	Error is logged, and startup is continued if last known good configuration is active, or system is restarted using last known good configuration

TErrorSeverity determines what action windows takes when the daemon fails to start. It is used on windows only, and is ignored on other platforms.

TGuiLoopEvent = procedure of object

TGuiLoopEvent is the main GUI loop event procedure prototype. It is called by the application instance in case the daemon has a visual part, which needs to handle visual events. It is run in the main application thread.

TServiceType = (stWin32, stDevice, stFileSystem)

Table 10.6: Enumeration values for type TServiceType

Value	Explanation
stDevice	Device driver
stFileSystem	File system driver
stWin32	Regular win32 service

The type of service. This type is used on windows only, to signal the operating system what kind of service is being installed or run.

`TStartType = (stBoot, stSystem, stAuto, stManual, stDisabled)`

Table 10.7: Enumeration values for type `TStartType`

Value	Explanation
<code>stAuto</code>	Started automatically by service manager during system startup
<code>stBoot</code>	During system boot
<code>stDisabled</code>	Service is not started, it is disabled
<code>stManual</code>	Started manually by the user or other processes.
<code>stSystem</code>	During load of device drivers

`TStartType` can be used to define when the service must be started on windows. This type is not used on other platforms.

### 10.4.3 Variables

`AppClass : TCustomDaemonApplicationClass`

`AppClass` can be set to the class of a `TCustomDaemonApplication` (286) descendant. When the `Application` (281) function needs to create an application instance, this class will be used. If `Application` was already called, the value of `AppClass` will be ignored.

`CurrentStatusNames : Array[TCurrentStatus] of string = ('Stopped', 'Start Pending', 'Stop Pending', 'Running', 'Continue Pending', 'Pause Pending', 'Paused')`

Names for various service statuses

`DefaultDaemonOptions : TDaemonOptions = [doAllowStop, doAllowPause]`

`DefaultDaemonOptions` are the default options with which a daemon definition (`TDaemonDef` (300)) is created.

`SStatus : Array[1..5] of string = ('Stop', 'Pause', 'Continue', 'Interrogate', 'Shutdown')`

Status message

## 10.5 Procedures and functions

### 10.5.1 Application

Synopsis: Application instance

Declaration: `function Application : TCustomDaemonApplication`

Visibility: default

**Description:** `Application` is the `TCustomDaemonApplication` (286) instance used by this application. The instance is created at the first invocation of this function, so it is possible to use `RegisterDaemonApplicationClass` (282) to register an alternative `TCustomDaemonApplication` class to run the application.

See also: `TCustomDaemonApplication` (286), `RegisterDaemonApplicationClass` (282)

### 10.5.2 DaemonError

**Synopsis:** Raise an `EDaemon` exception

**Declaration:** `procedure DaemonError(Msg: string)`  
`procedure DaemonError(Fmt: string; Args: Array of const)`

**Visibility:** default

**Description:** `DaemonError` raises an `EDaemon` (283) exception with message `Msg` or it formats the message using `Fmt` and `Args`.

See also: `EDaemon` (283)

### 10.5.3 RegisterDaemonApplicationClass

**Synopsis:** Register alternative `TCustomDaemonApplication` class.

**Declaration:** `procedure RegisterDaemonApplicationClass`  
`(AClass: TCustomDaemonApplicationClass)`

**Visibility:** default

**Description:** `RegisterDaemonApplicationClass` can be used to register an alternative `TCustomDaemonApplication` (286) descendent which will be used when creating the global `Application` (281) instance. Only the last registered class pointer will be used.

See also: `TCustomDaemonApplication` (286), `Application` (281)

### 10.5.4 RegisterDaemonClass

**Synopsis:** Register daemon

**Declaration:** `procedure RegisterDaemonClass(AClass: TCustomDaemonClass)`

**Visibility:** default

**Description:** `RegisterDaemonClass` must be called for each `TCustomDaemon` (283) descendent that is used in the class: the class pointer and class name are used by the `TCustomDaemonMapperClass` (279) class to create a `TCustomDaemon` instance when a daemon is required.

See also: `TCustomDaemonMapperClass` (279), `TCustomDaemon` (283)

### 10.5.5 RegisterDaemonMapper

**Synopsis:** Register a daemon mapper class

**Declaration:** `procedure RegisterDaemonMapper(AMapperClass: TCustomDaemonMapperClass)`

**Visibility:** default

**Description:** `RegisterDaemonMapper` can be used to register an alternative class for the global daemon-mapper. The daemonmapper will be used only when the application is being run, by the `TCustomDaemonApplication` (286) code, so registering an alternative mapping class should happen in the initialization section of the application units.

See also: `TCustomDaemonApplication` (286), `TCustomDaemonMapperClass` (279)

## 10.6 EDaemon

### 10.6.1 Description

`EDaemon` is the exception class used by all code in the `DaemonApp` unit.

See also: `DaemonError` (282)

## 10.7 TCustomDaemon

### 10.7.1 Description

`TCustomDaemon` implements all the basic calls that are needed for a daemon to function. Descendents of `TCustomDaemon` can override these calls to implement the daemon-specific behaviour.

`TCustomDaemon` is an abstract class, it should never be instantiated. Either a descendent of it must be created and instantiated, or a descendent of `TDaemon` (293) can be designed to implement the behaviour of the daemon.

See also: `TDaemon` (293), `TDaemonDef` (300), `TDaemonController` (297), `TDaemonApplication` (297)

### 10.7.2 Method overview

Page	Method	Description
283	<code>CheckControlMessages</code>	
284	<code>LogMessage</code>	Log a message to the system log
284	<code>ReportStatus</code>	Report the current status to the operating system

### 10.7.3 Property overview

Page	Properties	Access	Description
285	<code>Controller</code>	r	<code>TDaemonController</code> instance controlling this daemon instance
285	<code>DaemonThread</code>	r	Thread in which daemon is running
284	<code>Definition</code>	r	The definition used to instantiate this daemon instance
285	<code>Logger</code>	r	<code>TEventLog</code> instance used to send messages to the system log
285	<code>Status</code>	rw	Current status of the daemon

### 10.7.4 TCustomDaemon.CheckControlMessages

**Declaration:** `procedure CheckControlMessages (Wait: Boolean)`

**Visibility:** `public`

### 10.7.5 TCustomDaemon.LogMessage

Synopsis: Log a message to the system log

Declaration: `procedure LogMessage(const Msg: string)`

Visibility: public

Description: `LogMessage` can be used to send a message `Msg` to the system log. A `TEventLog` (524) instance is used to actually send messages to the system log.

The message is sent with an 'error' flag (using `TEventLog.Error` (527)).

Errors: None.

See also: `ReportStatus` (284)

### 10.7.6 TCustomDaemon.ReportStatus

Synopsis: Report the current status to the operating system

Declaration: `procedure ReportStatus`

Visibility: public

Description: `ReportStatus` can be used to report the current status to the operating system. The start and stop or pause and continue operations can be slow to start up. This call can (and should) be used to report the current status to the operating system during such lengthy operations, or else it may conclude that the daemon has died.

This call is mostly important on windows operating systems, to notify the service manager that the operation is still in progress.

The implementation of `ReportStatus` simply calls `ReportStatus` in the controller.

Errors: None.

See also: `LogMessage` (284)

### 10.7.7 TCustomDaemon.Definition

Synopsis: The definition used to instantiate this daemon instance

Declaration: `Property Definition : TDaemonDef`

Visibility: public

Access: Read

Description: `Definition` is the `TDaemonDef` (300) definition that was used to start the daemon instance. It can be used to retrieve additional information about the intended behaviour of the daemon.

See also: `TDaemonDef` (300)

### 10.7.8 TCustomDaemon.DaemonThread

Synopsis: Thread in which daemon is running

Declaration: `Property DaemonThread : TThread`

Visibility: public

Access: Read

Description: `DaemonThread` is the thread in which the daemon instance is running. Each daemon instance in the application runs in it's own thread, none of which are the main thread of the application. The application main thread is used to handle control messages coming from the operating system.

See also: [Controller \(285\)](#)

### 10.7.9 TCustomDaemon.Controller

Synopsis: `TDaemonController` instance controlling this daemon instance

Declaration: `Property Controller : TDaemonController`

Visibility: public

Access: Read

Description: `Controller` points to the `TDaemonController` instance that was created by the application instance to control this daemon.

See also: [DaemonThread \(285\)](#)

### 10.7.10 TCustomDaemon.Status

Synopsis: Current status of the daemon

Declaration: `Property Status : TCurrentStatus`

Visibility: public

Access: Read,Write

Description: `Status` indicates the current status of the daemon. It is set by the various operations that the controller operates on the daemon, and should not be set manually.

`Status` is the value which `ReportStatus` will send to the operating system.

See also: [ReportStatus \(284\)](#)

### 10.7.11 TCustomDaemon.Logger

Synopsis: `TEventLog` instance used to send messages to the system log

Declaration: `Property Logger : TEventLog`

Visibility: public

Access: Read

Description: `Logger` is the `TEventLog (524)` instance used to send messages to the system log. It is used by the `LogMessage (284)` call, but is accessible through the `Logger` property in case more configurable logging is needed than offered by `LogMessage`.

See also: [LogMessage \(284\)](#), [TEventLog \(524\)](#)

## 10.8 TCustomDaemonApplication

### 10.8.1 Description

TCustomDaemonApplication is a TCustomApplication (263) descendent which is the main application instance for a daemon. It handles the command-line and decides what to do when the application is started, depending on the command-line options given to the application, by calling the various methods.

It creates the necessary TDaemon (293) instances by checking the TCustomDaemonMapperClass (279) instance that contains the daemon maps.

See also: TCustomApplication (263), TCustomDaemonMapperClass (279)

### 10.8.2 Method overview

Page	Method	Description
<a href="#">286</a>	Create	
<a href="#">287</a>	CreateDaemon	Create daemon instance
<a href="#">288</a>	CreateForm	Create a component
<a href="#">286</a>	Destroy	Clean up the TCustomDaemonApplication instance
<a href="#">287</a>	InstallDaemons	Install all daemons.
<a href="#">288</a>	RunDaemons	Run all daemons.
<a href="#">287</a>	ShowException	Show an exception
<a href="#">288</a>	ShowHelp	Display a help message
<a href="#">287</a>	StopDaemons	Stop all daemons
<a href="#">288</a>	UnInstallDaemons	Uninstall all daemons

### 10.8.3 Property overview

Page	Properties	Access	Description
<a href="#">290</a>	AutoRegisterMessageFile	rw	
<a href="#">289</a>	EventLog	r	Event logger instance
<a href="#">289</a>	GuiHandle	rw	Handle of GUI loop main application window handle
<a href="#">289</a>	GUIMainLoop	rw	GUI main loop callback
<a href="#">289</a>	OnRun	rw	Event executed when the daemon is run.
<a href="#">290</a>	RunMode	r	Application mode

### 10.8.4 TCustomDaemonApplication.Create

Declaration: constructor Create(AOwner: TComponent); Override

Visibility: public

### 10.8.5 TCustomDaemonApplication.Destroy

Synopsis: Clean up the TCustomDaemonApplication instance

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy cleans up the event log instance and then calls the inherited destroy.

See also: `TCustomDaemonApplication.EventLog` ([289](#))

### 10.8.6 `TCustomDaemonApplication.ShowException`

Synopsis: Show an exception

Declaration: `procedure ShowException(E: Exception); Override`

Visibility: public

Description: `ShowException` is overridden by `TCustomDaemonApplication`, it sends the exception message to the system log.

### 10.8.7 `TCustomDaemonApplication.CreateDaemon`

Synopsis: Create daemon instance

Declaration: `function CreateDaemon(DaemonDef: TDaemonDef) : TCustomDaemon`

Visibility: public

Description: `CreateDaemon` is called whenever a `TCustomDaemon` ([283](#)) instance must be created from a `TDaemonDef` ([300](#)) daemon definition, passed in `DaemonDef`. It initializes the `TCustomDaemon` instance, and creates a controller instance of type `TDaemonController` ([297](#)) to control the daemon. Finally, it assigns the created daemon to the `TDaemonDef.Instance` ([301](#)) property.

Errors: In case of an error, an exception may be raised.

See also: `TDaemonController` ([297](#)), `TCustomDaemon` ([283](#)), `TDaemonDef` ([300](#)), `TDaemonDef.Instance` ([301](#))

### 10.8.8 `TCustomDaemonApplication.StopDaemons`

Synopsis: Stop all daemons

Declaration: `procedure StopDaemons(Force: Boolean)`

Visibility: public

Description: `StopDaemons` sends the `STOP` control code to all daemons, or the `SHUTDOWN` control code in case `Force` is `True`.

See also: `TDaemonController.Controller` ([298](#)), `TCustomDaemonApplication.UnInstallDaemons` ([288](#)), `TCustomDaemonApplication.RunDaemons` ([288](#))

### 10.8.9 `TCustomDaemonApplication.InstallDaemons`

Synopsis: Install all daemons.

Declaration: `procedure InstallDaemons`

Visibility: public

Description: `InstallDaemons` installs all known daemons, i.e. registers them with the service manager on Windows. This method is called if the application is run with the `-i` or `-install` or `/install` command-line option.

See also: `TCustomDaemonApplication.UnInstallDaemons` ([288](#)), `TCustomDaemonApplication.RunDaemons` ([288](#)), `TCustomDaemonApplication.StopDaemons` ([287](#))



### 10.8.10 TCustomDaemonApplication.RunDaemons

Synopsis: Run all daemons.

Declaration: `procedure RunDaemons`

Visibility: `public`

Description: `RunDaemons` runs (starts) all known daemons. This method is called if the application is run with the `-r` or `-run` methods.

See also: `TCustomDaemonApplication.UnInstallDaemons` (288), `TCustomDaemonApplication.InstallDaemons` (287), `TCustomDaemonApplication.StopDaemons` (287)

### 10.8.11 TCustomDaemonApplication.UnInstallDaemons

Synopsis: Uninstall all daemons

Declaration: `procedure UnInstallDaemons`

Visibility: `public`

Description: `UnInstallDaemons` uninstalls all known daemons, i.e. deregisters them with the service manager on Windows. This method is called if the application is run with the `-u` or `-uninstall` or `/uninstall` command-line option.

See also: `TCustomDaemonApplication.RunDaemons` (288), `TCustomDaemonApplication.InstallDaemons` (287), `TCustomDaemonApplication.StopDaemons` (287)

### 10.8.12 TCustomDaemonApplication.ShowHelp

Synopsis: Display a help message

Declaration: `procedure ShowHelp`

Visibility: `public`

Description: `ShowHelp` displays a help message explaining the command-line options on standard output.

### 10.8.13 TCustomDaemonApplication.CreateForm

Synopsis: Create a component

Declaration: `procedure CreateForm(InstanceClass: TComponentClass; var Reference)  
; Virtual`

Visibility: `public`

Description: `CreateForm` creates an instance of `InstanceClass` and fills `Reference` with the class instance pointer. It's main purpose is to give an IDE a means of assuring that forms or datamodules are created on application startup: the IDE will generate calls for all modules that are auto-created.

Errors: An exception may arise if the instance wants to stream itself from resources, but no resources are found.

See also: `TCustomDaemonApplication.CreateDaemon` (287)

### 10.8.14 TCustomDaemonApplication.OnRun

Synopsis: Event executed when the daemon is run.

Declaration: `Property OnRun : TNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `OnRun` is triggered when the daemon application is run and no appropriate options (one of install, uninstall or run) was given.

See also: `TCustomDaemonApplication.RunDaemons` ([288](#)), `TCustomDaemonApplication.InstallDaemons` ([287](#)), `TCustomDaemonApplication.UnInstallDaemons` ([288](#))

### 10.8.15 TCustomDaemonApplication.EventLog

Synopsis: Event logger instance

Declaration: `Property EventLog : TEventLog`

Visibility: `public`

Access: `Read`

Description: `EventLog` is the `TEventLog` ([524](#)) instance which is used to log events to the system log with the `Log` ([286](#)) method. It is created when the application instance is created, and destroyed when the application is destroyed.

See also: `TEventLog` ([524](#)), `Log` ([286](#))

### 10.8.16 TCustomDaemonApplication.GUIMainLoop

Synopsis: GUI main loop callback

Declaration: `Property GUIMainLoop : TGuiLoopEvent`

Visibility: `public`

Access: `Read,Write`

Description: `GUIMainLoop` contains a reference to a method that can be called to process a main GUI loop. The procedure should return only when the main GUI has finished and the application should exit. It is called when the daemons are running.

See also: `TCustomDaemonApplication.GuiHandle` ([289](#))

### 10.8.17 TCustomDaemonApplication.GuiHandle

Synopsis: Handle of GUI loop main application window handle

Declaration: `Property GuiHandle : THandle`

Visibility: `public`

Access: `Read,Write`

Description: `GuiHandle` is the handle of a GUI window which can be used to run a message handling loop on. It is created when no `GUIMainLoop` ([289](#)) procedure exists, and the application creates and runs a message loop by itself.

See also: `GUIMainLoop` ([289](#))

### 10.8.18 TCustomDaemonApplication.RunMode

Synopsis: Application mode

Declaration: Property RunMode : TDaemonRunMode

Visibility: public

Access: Read

Description: RunMode indicates in which mode the application is running currently. It is set automatically by examining the command-line, and when set, one of InstallDaemons (287), RunDaemons (288) or UnInstallDaemons (288) is called.

See also: InstallDaemons (287), RunDaemons (288), UnInstallDaemons (288)

### 10.8.19 TCustomDaemonApplication.AutoRegisterMessageFile

Declaration: Property AutoRegisterMessageFile : Boolean

Visibility: public

Access: Read, Write

## 10.9 TCustomDaemonMapper

### 10.9.1 Description

The TCustomDaemonMapper class is responsible for mapping a daemon definition to an actual TDaemon instance. It maintains a TDaemonDefs (304) collection with daemon definitions, which can be used to map the definition of a daemon to a TDaemon descendent class.

An IDE such as Lazarus can design a TCustomDaemonMapper instance visually, to help establish the relationship between various TDaemonDef (300) definitions and the actual TDaemon (293) instances that will be used to run the daemons.

The TCustomDaemonMapper class has no support for streaming. The TDaemonMapper (306) class has support for streaming (and hence visual designing).

See also: TDaemon (293), TDaemonDef (300), TDaemonDefs (304), TDaemonMapper (306)

### 10.9.2 Method overview

Page	Method	Description
<a href="#">291</a>	Create	Create a new instance of TCustomDaemonMapper
<a href="#">291</a>	Destroy	Clean up and destroy a TCustomDaemonMapper instance.

### 10.9.3 Property overview

Page	Properties	Access	Description
<a href="#">291</a>	DaemonDefs	rw	Collection of daemons
<a href="#">291</a>	OnCreate	rw	Event called when the daemon mapper is created
<a href="#">292</a>	OnDestroy	rw	Event called when the daemon mapper is freed.
<a href="#">292</a>	OnInstall	rw	Event called when the daemons are installed
<a href="#">292</a>	OnRun	rw	Event called when the daemons are executed.
<a href="#">292</a>	OnUnInstall	rw	Event called when the daemons are uninstalled

### 10.9.4 TCustomDaemonMapper.Create

Synopsis: Create a new instance of TCustomDaemonMapper

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` creates a new instance of a TCustomDaemonMapper. It creates the TDaemonDefs (304) collection and then calls the inherited constructor. It should never be necessary to create a daemon mapper manually, the application will create a global TCustomDaemonMapper instance.

See also: TDaemonDefs (304), TCustomDaemonApplication (286), TCustomDaemonMapper.Destroy (291)

### 10.9.5 TCustomDaemonMapper.Destroy

Synopsis: Clean up and destroy a TCustomDaemonMapper instance.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` frees the DaemonDefs (291) collection and calls the inherited destructor.

See also: TDaemonDefs (304), TCustomDaemonMapper.Create (291)

### 10.9.6 TCustomDaemonMapper.DaemonDefs

Synopsis: Collection of daemons

Declaration: `Property DaemonDefs : TDaemonDefs`

Visibility: `published`

Access: `Read,Write`

Description: `DaemonDefs` is the application's global collection of daemon definitions. This collection will be used to decide at runtime which TDaemon class must be created to run or install a daemon.

See also: TCustomDaemonApplication (286)

### 10.9.7 TCustomDaemonMapper.OnCreate

Synopsis: Event called when the daemon mapper is created

Declaration: `Property OnCreate : TNotifyEvent`

Visibility: `published`

Access: `Read,Write`

Description: `OnCreate` is an event that is called when the TCustomDaemonMapper instance is created. It can for instance be used to dynamically create daemon definitions at runtime.

See also: OnDestroy (292), OnUnInstall (292), OnCreate (291), OnDestroy (292)

### 10.9.8 TCustomDaemonMapper.OnDestroy

Synopsis: Event called when the daemon mapper is freed.

Declaration: `Property OnDestroy : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnDestroy` is called when the global daemon mapper instance is destroyed. it can be used to release up any resources that were allocated when the instance was created, in the `OnCreate` (291) event.

See also: `OnCreate` (291), `OnInstall` (292), `OnUnInstall` (292), `OnCreate` (291)

### 10.9.9 TCustomDaemonMapper.OnRun

Synopsis: Event called when the daemons are executed.

Declaration: `Property OnRun : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnRun` is the event called when the daemon application is executed to run the daemons (with command-line parameter '-r'). it is called exactly once.

See also: `OnInstall` (292), `OnUnInstall` (292), `OnCreate` (291), `OnDestroy` (292)

### 10.9.10 TCustomDaemonMapper.OnInstall

Synopsis: Event called when the daemons are installed

Declaration: `Property OnInstall : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnInstall` is the event called when the daemon application is executed to install the daemons (with command-line parameter '-i' or '/install'). it is called exactly once.

See also: `OnRun` (292), `OnUnInstall` (292), `OnCreate` (291), `OnDestroy` (292)

### 10.9.11 TCustomDaemonMapper.OnUnInstall

Synopsis: Event called when the daemons are uninstalled

Declaration: `Property OnUnInstall : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnUnInstall` is the event called when the daemon application is executed to uninstall the daemons (with command-line parameter '-u' or '/uninstall'). it is called exactly once.

See also: `OnRun` (292), `OnInstall` (292), `OnCreate` (291), `OnDestroy` (292)

## 10.10 TDaemon

### 10.10.1 Description

TDaemon is a TCustomDaemon (283) descendent which is meant for development in a visual environment: it contains event handlers for all major operations. Whenever a TCustomDaemon method is executed, it's execution is shunted to the event handler, which can be filled with code in the IDE.

All the events of the daemon are executed in the thread in which the daemon's controller is running (as given by DaemonThread (285)), which is not the main program thread.

See also: TCustomDaemon (283), TDaemonController (297)

### 10.10.2 Property overview

Page	Properties	Access	Description
296	AfterInstall	rw	Called after the daemon was installed
296	AfterUnInstall	rw	Called after the daemon is uninstalled
295	BeforeInstall	rw	Called before the daemon will be installed
296	BeforeUnInstall	rw	Called before the daemon is uninstalled
293	Definition		
294	OnContinue	rw	Daemon continue
296	OnControlCode	rw	Called when a control code is received for the daemon
295	OnExecute	rw	Daemon execute event
294	OnPause	rw	Daemon pause event
295	OnShutDown	rw	Daemon shutdown
293	OnStart	rw	Daemon start event
294	OnStop	rw	Daemon stop event
293	Status		

### 10.10.3 TDaemon.Definition

Declaration: Property Definition :

Visibility: public

Access:

### 10.10.4 TDaemon.Status

Declaration: Property Status :

Visibility: public

Access:

### 10.10.5 TDaemon.OnStart

Synopsis: Daemon start event

Declaration: Property OnStart : TDaemonOKEvent

Visibility: published

Access: Read,Write

**Description:** `OnStart` is the event called when the daemon must be started. This event handler should return as quickly as possible. If it must perform lengthy operations, it is best to report the status to the operating system at regular intervals using the `ReportStatus` (284) method.

If the start of the daemon should do some continuous action, then this action should be performed in a new thread: this thread should then be created and started in the `OnExecute` (295) event handler, so the event handler can return at once.

See also: `TDaemon.OnStop` (294), `TDaemon.OnExecute` (295), `TDaemon.OnContinue` (294), `ReportStatus` (284)

### 10.10.6 `TDaemon.OnStop`

**Synopsis:** Daemon stop event

**Declaration:** `Property OnStop : TDaemonOKEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `OnStart` is the event called when the daemon must be stopped. This event handler should return as quickly as possible. If it must perform lengthy operations, it is best to report the status to the operating system at regular intervals using the `ReportStatus` (284) method.

If a thread was started in the `OnExecute` (295) event, this is the place where the thread should be stopped.

See also: `TDaemon.OnStart` (293), `TDaemon.OnPause` (294), `ReportStatus` (284)

### 10.10.7 `TDaemon.OnPause`

**Synopsis:** Daemon pause event

**Declaration:** `Property OnPause : TDaemonOKEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `OnPause` is the event called when the daemon must be stopped. This event handler should return as quickly as possible. If it must perform lengthy operations, it is best to report the status to the operating system at regular intervals using the `ReportStatus` (284) method.

If a thread was started in the `OnExecute` (295) event, this is the place where the thread's execution should be suspended.

See also: `TDaemon.OnStop` (294), `TDaemon.OnContinue` (294), `ReportStatus` (284)

### 10.10.8 `TDaemon.OnContinue`

**Synopsis:** Daemon continue

**Declaration:** `Property OnContinue : TDaemonOKEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `OnPause` is the event called when the daemon must be stopped. This event handler should return as quickly as possible. If it must perform lengthy operations, it is best to report the status to the operating system at regular intervals using the `ReportStatus` (284) method.

If a thread was started in the `OnExecute` (295) event and it was suspended in a `OnPause` (293) event, this is the place where the thread's executed should be resumed.

See also: `TDaemon.OnStart` (293), `TDaemon.OnPause` (294), `ReportStatus` (284)

### 10.10.9 `TDaemon.OnShutDown`

**Synopsis:** Daemon shutdown

**Declaration:** `Property OnShutDown : TDaemonEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `OnShutDown` is the event called when the daemon must be shut down. When the system is being shut down and the daemon does not respond to stop signals, then a shutdown message is sent to the daemon. This event can be used to respond to such a message. The daemon process will simply be stopped after this event.

If a thread was started in the `OnExecute` (295), this is the place where the thread's executed should be stopped or the thread freed from memory.

See also: `TDaemon.OnStart` (293), `TDaemon.OnPause` (294), `ReportStatus` (284)

### 10.10.10 `TDaemon.OnExecute`

**Synopsis:** Daemon execute event

**Declaration:** `Property OnExecute : TDaemonEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `OnExecute` is executed once after the daemon was started. If assigned, it should perform whatever operation the daemon is designed.

If the daemon's action is event based, then no `OnExecute` handler is needed, and the events will control the daemon's execution: the daemon thread will then go in a loop, passing control messages to the daemon.

If an `OnExecute` event handler is present, the checking for control messages must be done by the implementation of the `OnExecute` handler.

See also: `TDaemon.OnStart` (293), `TDaemon.OnStop` (294)

### 10.10.11 `TDaemon.BeforeInstall`

**Synopsis:** Called before the daemon will be installed

**Declaration:** `Property BeforeInstall : TDaemonEvent`

**Visibility:** published

**Access:** Read,Write



**Description:** `BeforeInstall` is called before the daemon is installed. It can be done to specify extra dependencies, or change the daemon description etc.

See also: [AfterInstall \(296\)](#), [BeforeUnInstall \(296\)](#), [AfterUnInstall \(296\)](#)

### 10.10.12 **TDaemon.AfterInstall**

**Synopsis:** Called after the daemon was installed

**Declaration:** `Property AfterInstall : TDaemonEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `AfterInstall` is called after the daemon was successfully installed.

See also: [BeforeInstall \(295\)](#), [BeforeUnInstall \(296\)](#), [AfterUnInstall \(296\)](#)

### 10.10.13 **TDaemon.BeforeUnInstall**

**Synopsis:** Called before the daemon is uninstalled

**Declaration:** `Property BeforeUnInstall : TDaemonEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `BeforeUnInstall` is called before the daemon is uninstalled.

See also: [BeforeInstall \(295\)](#), [AfterInstall \(296\)](#), [AfterUnInstall \(296\)](#)

### 10.10.14 **TDaemon.AfterUnInstall**

**Synopsis:** Called after the daemon is uninstalled

**Declaration:** `Property AfterUnInstall : TDaemonEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `AfterUnInstall` is called after the daemon is successfully uninstalled.

See also: [BeforeInstall \(295\)](#), [AfterInstall \(296\)](#), [BeforeUnInstall \(296\)](#)

### 10.10.15 **TDaemon.OnControlCode**

**Synopsis:** Called when a control code is received for the daemon

**Declaration:** `Property OnControlCode : TCustomControlCodeEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `OnControlCode` is called when the daemon receives a control code. If the daemon has not handled the control code, it should set the `Handled` parameter to `False`. By default it is set to `True`.

See also: [Architecture \(277\)](#)

## 10.11 TDaemonApplication

### 10.11.1 Description

`TDaemonApplication` is the default `TCustomDaemonApplication` (286) descendent that is used to run the daemon application. It is possible to register an alternative `TCustomDaemonApplication` class (using `RegisterDaemonApplicationClass` (282)) to run the application in a different manner.

See also: `TCustomDaemonApplication` (286), `RegisterDaemonApplicationClass` (282)

## 10.12 TDaemonController

### 10.12.1 Description

`TDaemonController` is a class that is used by the `TDaemonApplication` (297) class to control the daemon during runtime. The `TDaemonApplication` class instantiates an instance of `TDaemonController` for each daemon in the application and communicates with the daemon through the `TDaemonController` instance. It should rarely be necessary to access or use this class.

See also: `TCustomDaemon` (283), `TDaemonApplication` (297)

### 10.12.2 Method overview

Page	Method	Description
298	Controller	Controller
297	Create	Create a new instance of the <code>TDaemonController</code> class
298	Destroy	Free a <code>TDaemonController</code> instance.
298	Main	Daemon main entry point
299	ReportStatus	Report the status to the operating system.
298	StartService	Start the service

### 10.12.3 Property overview

Page	Properties	Access	Description
300	CheckPoint	r	Send checkpoint signal to the operating system
299	Daemon	r	Daemon instance this controller controls.
299	LastStatus	r	Last reported status
299	Params	r	Parameters passed to the daemon

### 10.12.4 TDaemonController.Create

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

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` creates a new instance of the `TDaemonController` class. It should never be necessary to create a new instance manually, because the controllers are created by the global `TDaemonApplication` (297) instance, and `AOwner` will be set to the global `TDaemonApplication` (297) instance.

See also: `TDaemonApplication` (297), `Destroy` (298)

### 10.12.5 TDaemonController.Destroy

Synopsis: Free a TDaemonController instance.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: Destroy deallocates some resources allocated when the instance was created.

See also: Create ([297](#))

### 10.12.6 TDaemonController.StartService

Synopsis: Start the service

Declaration: `procedure StartService; Virtual`

Visibility: `public`

Description: StartService starts the service controlled by this instance.

Errors: None.

See also: TDaemonController.Main ([298](#))

### 10.12.7 TDaemonController.Main

Synopsis: Daemon main entry point

Declaration: `procedure Main(Argc: DWord; Args: PPChar); Virtual`

Visibility: `public`

Description: Main is the service's main entry point, called when the system wants to start the service. The global application will call this function whenever required, with the appropriate arguments.

The standard implementation starts the daemon thread, and waits for it to stop. All other daemon action - such as responding to control code events - is handled by the thread.

Errors: If the daemon thread cannot be created, an exception is raised.

See also: TDaemonThread ([307](#))

### 10.12.8 TDaemonController.Controller

Synopsis: Controller

Declaration: `procedure Controller(ControlCode: DWord; EventType: DWord;  
EventData: Pointer); Virtual`

Visibility: `public`

Description: Controller is responsible for sending the control code to the daemon thread so it can be processed.

This routine is currently only used on windows, as there is no service manager on Linux. Later on this may be changed to respond to signals on Linux as well.

See also: TDaemon.OnControlCode ([296](#))

### 10.12.9 TDaemonController.ReportStatus

Synopsis: Report the status to the operating system.

Declaration: `function ReportStatus : Boolean; Virtual`

Visibility: `public`

Description: `ReportStatus` reports the status of the daemon to the operating system. On windows, this sends the current service status to the service manager. On other operating systems, this sends a message to the system log.

Errors: If an error occurs, an error message is sent to the system log.

See also: `TDaemon.ReportStatus` ([293](#)), `TDaemonController.LastStatus` ([299](#))

### 10.12.10 TDaemonController.Daemon

Synopsis: Daemon instance this controller controls.

Declaration: `Property Daemon : TCustomDaemon`

Visibility: `public`

Access: `Read`

Description: `Daemon` is the daemon instance that is controller by this instance of the `TDaemonController` class.

### 10.12.11 TDaemonController.Params

Synopsis: Parameters passed to the daemon

Declaration: `Property Params : TStrings`

Visibility: `public`

Access: `Read`

Description: `Params` contains the parameters passed to the daemon application by the operating system, comparable to the application's command-line parameters. The property is set by the `Main` ([298](#)) method.

### 10.12.12 TDaemonController.LastStatus

Synopsis: Last reported status

Declaration: `Property LastStatus : TCurrentStatus`

Visibility: `public`

Access: `Read`

Description: `LastStatus` is the last status reported to the operating system.

See also: `ReportStatus` ([299](#))

### 10.12.13 TDaemonController.CheckPoint

Synopsis: Send checkpoint signal to the operating system

Declaration: `Property CheckPoint : DWord`

Visibility: public

Access: Read

Description: `CheckPoint` can be used to send a checkpoint signal during lengthy operations, to signal that a lengthy operation is in progress. This should be used mainly on windows, to signal the service manager that the service is alive.

See also: `ReportStatus` (299)

## 10.13 TDaemonDef

### 10.13.1 Description

`TDaemonDef` contains the definition of a daemon in the application: The name of the daemon, which `TCustomDaemon` (283) descendent should be started to run the daemon, a description, and various other options should be set in this class. The global `TDaemonApplication` instance maintains a collection of `TDaemonDef` instances and will use these definitions to install or start the various daemons.

See also: `TDaemonApplication` (297), `TDaemon` (293)

### 10.13.2 Method overview

Page	Method	Description
<a href="#">300</a>	Create	Create a new <code>TDaemonDef</code> instance
<a href="#">301</a>	Destroy	Free a <code>TDaemonDef</code> from memory

### 10.13.3 Property overview

Page	Properties	Access	Description
<a href="#">301</a>	<code>DaemonClass</code>	r	<code>TDaemon</code> class to use for this daemon
<a href="#">301</a>	<code>DaemonClassName</code>	rw	Name of the <code>TDaemon</code> class to use for this daemon
<a href="#">302</a>	<code>Description</code>	rw	Description of the daemon
<a href="#">302</a>	<code>DisplayName</code>	rw	Displayed name of the daemon (service)
<a href="#">303</a>	<code>Enabled</code>	rw	Is the daemon enabled or not
<a href="#">301</a>	<code>Instance</code>	rw	Instance of the daemon class
<a href="#">304</a>	<code>LogStatusReport</code>	rw	Log the status report to the system log
<a href="#">302</a>	<code>Name</code>	rw	Name of the daemon (service)
<a href="#">303</a>	<code>OnCreateInstance</code>	rw	Event called when a daemon is instantiated
<a href="#">303</a>	<code>Options</code>	rw	Service options
<a href="#">302</a>	<code>RunArguments</code>	rw	Additional command-line arguments when running daemon.
<a href="#">303</a>	<code>WinBindings</code>	rw	Windows-specific bindings (windows only)

### 10.13.4 TDaemonDef.Create

Synopsis: Create a new `TDaemonDef` instance

**Declaration:** `constructor Create(ACollection: TCollection); Override`

**Visibility:** `public`

**Description:** `Create` initializes a new `TDaemonDef` instance. It should not be necessary to instantiate a definition manually, it is handled by the collection.

**See also:** `TDaemonDefs` ([304](#))

### 10.13.5 `TDaemonDef.Destroy`

**Synopsis:** Free a `TDaemonDef` from memory

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

**Visibility:** `public`

**Description:** `Destroy` removes the `TDaemonDef` from memory.

### 10.13.6 `TDaemonDef.DaemonClass`

**Synopsis:** `TDaemon` class to use for this daemon

**Declaration:** `Property DaemonClass : TCustomDaemonClass`

**Visibility:** `public`

**Access:** `Read`

**Description:** `DaemonClass` is the `TDaemon` class that is used when this service is requested. It is looked up in the application's global daemon mapper by it's name in `DaemonClassName` ([301](#)).

**See also:** `DaemonClassName` ([301](#)), `TDaemonMapper` ([306](#))

### 10.13.7 `TDaemonDef.Instance`

**Synopsis:** Instance of the daemon class

**Declaration:** `Property Instance : TCustomDaemon`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `Instance` points to the `TDaemon` ([293](#)) instance that is used when the service is in operation at runtime.

**See also:** `TDaemonDef.DaemonClass` ([301](#))

### 10.13.8 `TDaemonDef.DaemonClassName`

**Synopsis:** Name of the `TDaemon` class to use for this daemon

**Declaration:** `Property DaemonClassName : string`

**Visibility:** `published`

**Access:** `Read,Write`

**Description:** `DaemonClassName` is the name of the `TDaemon` class that will be used whenever the service is needed. The name is used to look up the class pointer registered in the daemon mapper, when `TCustomDaemonApplication.CreateDaemonInstance` (286) creates an instance of the daemon.

See also: `TDaemonDef.Instance` (301), `TDaemonDef.DaemonClass` (301), `RegisterDaemonClass` (282)

### 10.13.9 TDaemonDef.Name

**Synopsis:** Name of the daemon (service)

**Declaration:** `Property Name : string`

**Visibility:** published

**Access:** Read,Write

**Description:** `Name` is the internal name of the daemon as it is known to the operating system.

See also: `TDaemonDef.DisplayName` (302)

### 10.13.10 TDaemonDef.Description

**Synopsis:** Description of the daemon

**Declaration:** `Property Description : string`

**Visibility:** published

**Access:** Read,Write

**Description:** `Description` is the description shown in the Windows service manager when managing this service. It is supplied to the windows service manager when the daemon is installed.

### 10.13.11 TDaemonDef.DisplayName

**Synopsis:** Displayed name of the daemon (service)

**Declaration:** `Property DisplayName : string`

**Visibility:** published

**Access:** Read,Write

**Description:** `DisplayName` is the displayed name of the daemon as it is known to the operating system.

See also: `TDaemonDef.Name` (302)

### 10.13.12 TDaemonDef.RunArguments

**Synopsis:** Additional command-line arguments when running daemon.

**Declaration:** `Property RunArguments : string`

**Visibility:** published

**Access:** Read,Write

**Description:** `RunArguments` specifies any additional command-line arguments that should be specified when running the daemon: these arguments will be passed to the service manager when registering the service on windows.

### 10.13.13 TDaemonDef.Options

Synopsis: Service options

Declaration: `Property Options : TDaemonOptions`

Visibility: published

Access: Read,Write

Description: `Options` tells the operating system which operations can be performed on the daemon while it is running.

This option is only used during the installation of the daemon.

### 10.13.14 TDaemonDef.Enabled

Synopsis: Is the daemon enabled or not

Declaration: `Property Enabled : Boolean`

Visibility: published

Access: Read,Write

Description: `Enabled` specifies whether a daemon should be installed, run or uninstalled. Disabled daemons are not installed, run or uninstalled.

### 10.13.15 TDaemonDef.WinBindings

Synopsis: Windows-specific bindings (windows only)

Declaration: `Property WinBindings : TWinBindings`

Visibility: published

Access: Read,Write

Description: `WinBindings` is used to group together the windows-specific properties of the daemon. This property is totally ignored on other platforms.

See also: `TWinBindings` ([311](#))

### 10.13.16 TDaemonDef.OnCreateInstance

Synopsis: Event called when a daemon is instantiated

Declaration: `Property OnCreateInstance : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnCreateInstance` is called whenever an instance of the daemon is created. This can be used for instance when a single `TDaemon` class is used to run several services, to correctly initialize the `TDaemon`.



### 10.13.17 TDaemonDef.LogStatusReport

Synopsis: Log the status report to the system log

Declaration: `Property LogStatusReport : Boolean`

Visibility: published

Access: Read,Write

Description: `LogStatusReport` can be set to `True` to send the status reports also to the system log. This can be used to track the progress of the daemon.

See also: `TDaemon.ReportStatus` (293)

## 10.14 TDaemonDefs

### 10.14.1 Description

`TDaemonDefs` is the class of the global list of daemon definitions. It contains an item for each daemon in the application.

Normally it is not necessary to create an instance of `TDaemonDefs` manually. The global `TCustomDaemonMapper` (290) instance will create a collection and maintain it.

See also: `TCustomDaemonMapper` (290), `TDaemonDef` (300)

### 10.14.2 Method overview

Page	Method	Description
<a href="#">304</a>	<code>Create</code>	Create a new instance of a <code>TDaemonDefs</code> collection.
<a href="#">305</a>	<code>DaemonDefByName</code>	Find and return instance of daemon definition with given name.
<a href="#">305</a>	<code>FindDaemonDef</code>	Find and return instance of daemon definition with given name.
<a href="#">305</a>	<code>IndexOfDaemonDef</code>	Return index of daemon definition

### 10.14.3 Property overview

Page	Properties	Access	Description
<a href="#">305</a>	<code>Daemons</code>	<code>rw</code>	Indexed access to <code>TDaemonDef</code> instances

### 10.14.4 TDaemonDefs.Create

Synopsis: Create a new instance of a `TDaemonDefs` collection.

Declaration: `constructor Create(AOwner: TPersistent; AClass: TCollectionItemClass)`

Visibility: public

Description: `Create` creates a new instance of the `TDaemonDefs` collection. It keeps the `AOwner` parameter for future reference and calls the inherited constructor.

Normally it is not necessary to create an instance of `TDaemonDefs` manually. The global `TCustomDaemonMapper` (290) instance will create a collection and maintain it.

See also: `TDaemonDef` (300)

### 10.14.5 TDaemonDefs.IndexOfDaemonDef

Synopsis: Return index of daemon definition

Declaration: `function IndexOfDaemonDef(const DaemonName: string) : Integer`

Visibility: public

Description: `IndexOfDaemonDef` searches the collection for a `TDaemonDef` (300) instance with a name equal to `DemonName`, and returns it's index. It returns -1 if no definition was found with this name. The search is case insensitive.

See also: `TDaemonDefs.FindDaemonDef` (305), `TDaemonDefs.DaemonDefByName` (305)

### 10.14.6 TDaemonDefs.FindDaemonDef

Synopsis: Find and return instance of daemon definition with given name.

Declaration: `function FindDaemonDef(const DaemonName: string) : TDaemonDef`

Visibility: public

Description: `FindDaemonDef` searches the list of daemon definitions and returns the `TDaemonDef` (300) instance whose name matches `DemonName`. If no definition is found, `Nil` is returned.

See also: `TDaemonDefs.IndexOfDaemonDef` (305), `TDaemonDefs.DaemonDefByName` (305)

### 10.14.7 TDaemonDefs.DaemonDefByName

Synopsis: Find and return instance of daemon definition with given name.

Declaration: `function DaemonDefByName(const DaemonName: string) : TDaemonDef`

Visibility: public

Description: `FindDaemonDef` searches the list of daemon definitions and returns the `TDaemonDef` (300) instance whose name matches `DemonName`. If no definition is found, an `EDaemon` (283) exception is raised.

The `FindDaemonDef` (305) call does not raise an error, but returns `Nil` instead.

Errors: If no definition is found, an `EDaemon` (283) exception is raised.

See also: `TDaemonDefs.IndexOfDaemonDef` (305), `TDaemonDefs.FindDaemonDef` (305)

### 10.14.8 TDaemonDefs.Daemons

Synopsis: Indexed access to `TDaemonDef` instances

Declaration: `Property Daemons[Index: Integer]: TDaemonDef; default`

Visibility: public

Access: Read,Write

Description: `Daemons` is the default property of `TDaemonDefs`, it gives access to the `TDaemonDef` instances in the collection.

See also: `TDaemonDef` (300)

## 10.15 TDaemonMapper

### 10.15.1 Description

`TDaemonMapper` is a direct descendent of `TCustomDaemonMapper` (290), but introduces no new functionality. It's sole purpose is to make it possible for an IDE to stream the `TDaemonMapper` instance.

For this purpose, it overrides the `Create` constructor and tries to find a resource with the same name as the class name, and tries to stream the instance from this resource.

If the instance should not be streamed, the `CreateNew` (306) constructor can be used instead.

See also: `CreateNew` (306), `Create` (306)

### 10.15.2 Method overview

Page	Method	Description
<a href="#">306</a>	<code>Create</code>	Create a new <code>TDaemonMapper</code> instance and initializes it from streamed resources.
<a href="#">306</a>	<code>CreateNew</code>	Create a new <code>TDaemonMapper</code> instance without initialization

### 10.15.3 TDaemonMapper.Create

**Synopsis:** Create a new `TDaemonMapper` instance and initializes it from streamed resources.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** default

**Description:** `Create` initializes a new instance of `TDaemonMapper` and attempts to read the component from resources compiled in the application.

If the instance should not be streamed, the `CreateNew` (306) constructor can be used instead.

**Errors:** If no streaming system is found, or no resource exists for the class, an exception is raised.

See also: `CreateNew` (306)

### 10.15.4 TDaemonMapper.CreateNew

**Synopsis:** Create a new `TDaemonMapper` instance without initialization

**Declaration:** `constructor CreateNew(AOwner: TComponent; Dummy: Integer)`

**Visibility:** default

**Description:** `CreateNew` initializes a new instance of `TDaemonMapper`. In difference with the `Create` constructor, it does not attempt to read the component from a stream.

See also: `Create` (306)

## 10.16 TDaemonThread

### 10.16.1 Description

TDaemonThread is the thread in which the daemons in the application are run. Each daemon is run in it's own thread.

It should not be necessary to create these threads manually, the TDaemonController (297) class will take care of this.

See also: TDaemonController (297), TDaemon (293)

### 10.16.2 Method overview

Page	Method	Description
<a href="#">308</a>	CheckControlMessage	Check if a control message has arrived
<a href="#">308</a>	ContinueDaemon	Continue the daemon
<a href="#">307</a>	Create	Create a new thread
<a href="#">307</a>	Execute	Run the daemon
<a href="#">309</a>	InterrogateDaemon	Report the daemon status
<a href="#">308</a>	PauseDaemon	Pause the daemon
<a href="#">309</a>	ShutDownDaemon	Shut down daemon
<a href="#">308</a>	StopDaemon	Stops the daemon

### 10.16.3 Property overview

Page	Properties	Access	Description
<a href="#">309</a>	Daemon	r	Daemon instance

### 10.16.4 TDaemonThread.Create

Synopsis: Create a new thread

Declaration: `constructor Create (ADaemon: TCustomDaemon)`

Visibility: public

Description: `Create` creates a new thread instance. It initializes the `Daemon` property with the passed `ADaemon`. The thread is created suspended.

See also: TDaemonThread.Daemon (309)

### 10.16.5 TDaemonThread.Execute

Synopsis: Run the daemon

Declaration: `procedure Execute; Override`

Visibility: public

Description: `Execute` starts executing the daemon and waits till the daemon stops. It also listens for control codes for the daemon.

See also: TDaemon.Execute (293)

### 10.16.6 TDaemonThread.CheckControlMessage

Synopsis: Check if a control message has arrived

Declaration: `procedure CheckControlMessage (WaitForMessage: Boolean)`

Visibility: `public`

Description: `CheckControlMessage` checks if a control message has arrived for the daemon and executes the appropriate daemon message. If the parameter `WaitForMessage` is `True`, then the routine waits for the message to arrive. If it is `False` and no message is present, it returns at once.

### 10.16.7 TDaemonThread.StopDaemon

Synopsis: Stops the daemon

Declaration: `function StopDaemon : Boolean; Virtual`

Visibility: `public`

Description: `StopDaemon` attempts to stop the daemon using its `TDaemon.Stop` (293) method, and terminates the thread.

See also: `TDaemon.Stop` (293), `TDaemonThread.PauseDaemon` (308), `TDaemonThread.ShutDownDaemon` (309)

### 10.16.8 TDaemonThread.PauseDaemon

Synopsis: Pause the daemon

Declaration: `function PauseDaemon : Boolean; Virtual`

Visibility: `public`

Description: `PauseDaemon` attempts to stop the daemon using its `TDaemon.Pause` (293) method, and suspends the thread. It returns `True` if the attempt was successful.

See also: `TDaemon.Pause` (293), `TDaemonThread.StopDaemon` (308), `TDaemonThread.ContinueDaemon` (308), `TDaemonThread.ShutDownDaemon` (309)

### 10.16.9 TDaemonThread.ContinueDaemon

Synopsis: Continue the daemon

Declaration: `function ContinueDaemon : Boolean; Virtual`

Visibility: `public`

Description: `ContinueDaemon` attempts to stop the daemon using its `TDaemon.Continue` (293) method. It returns `True` if the attempt was successful.

See also: `TDaemon.Continue` (293), `TDaemonThread.StopDaemon` (308), `TDaemonThread.PauseDaemon` (308), `TDaemonThread.ShutDownDaemon` (309)

### 10.16.10 TDaemonThread.ShutDownDaemon

Synopsis: Shut down daemon

Declaration: `function ShutDownDaemon : Boolean; Virtual`

Visibility: public

Description: `ShutDownDaemon` shuts down the daemon. This happens normally only when the system is shut down and the daemon didn't respond to the stop request. The return result is the result of the `TDaemon.Shutdown` (293) function. The thread is terminated by this method.

See also: `TDaemon.Shutdown` (293), `TDaemonThread.StopDaemon` (308), `TDaemonThread.PauseDaemon` (308), `TDaemonThread.ContinueDaemon` (308)

### 10.16.11 TDaemonThread.InterrogateDaemon

Synopsis: Report the daemon status

Declaration: `function InterrogateDaemon : Boolean; Virtual`

Visibility: public

Description: `InterrogateDaemon` simply calls `TDaemon.ReportStatus` (293) for the daemon that is running in this thread. It always returns `True`.

See also: `TDaemon.ReportStatus` (293)

### 10.16.12 TDaemonThread.Daemon

Synopsis: Daemon instance

Declaration: `Property Daemon : TCustomDaemon`

Visibility: public

Access: Read

Description: `Daemon` is the daemon instance which is running in this thread.

See also: `TDaemon` (293)

## 10.17 TDependencies

### 10.17.1 Description

`TDependencies` is just a descendent of `TCollection` which contains a series of dependencies on other services. It overrides the default property of `TCollection` to return `TDependency` (310) instances.

See also: `TDependency` (310)

### 10.17.2 Method overview

Page	Method	Description
310	Create	Create a new instance of a <code>TDependencies</code> collection.

### 10.17.3 Property overview

Page	Properties	Access	Description
<a href="#">310</a>	Items	rw	Default property override

### 10.17.4 TDependencies.Create

Synopsis: Create a new instance of a `TDependencies` collection.

Declaration: constructor `Create(AOwner: TPersistent)`

Visibility: public

Description: `Create` Create a new instance of a `TDependencies` collection.

### 10.17.5 TDependencies.Items

Synopsis: Default property override

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

Visibility: public

Access: Read,Write

Description: `Items` overrides the default property of `TCollection` so the items are of type `TDependency` ([310](#)).

See also: `TDependency` ([310](#))

## 10.18 TDependency

### 10.18.1 Description

`TDependency` is a collection item used to specify dependencies on other daemons (services) in windows. It is used only on windows and when installing the daemon: changing the dependencies of a running daemon has no effect.

See also: `TDependencies` ([309](#)), `TDaemonDef` ([300](#))

### 10.18.2 Method overview

Page	Method	Description
<a href="#">311</a>	Assign	Assign <code>TDependency</code> instance to another

### 10.18.3 Property overview

Page	Properties	Access	Description
<a href="#">311</a>	IsGroup	rw	Name refers to a service group
<a href="#">311</a>	Name	rw	Name of the service

### 10.18.4 TDependency.Assign

Synopsis: Assign TDependency instance to another

Declaration: `procedure Assign(Source: TPersistent); Override`

Visibility: public

Description: Assign is overridden by TDependency to copy all properties from one instance to another.

### 10.18.5 TDependency.Name

Synopsis: Name of the service

Declaration: `Property Name : string`

Visibility: published

Access: Read,Write

Description: Name is the name of a service or service group that the current daemon depends on.

See also: TDependency.IsGroup ([311](#))

### 10.18.6 TDependency.IsGroup

Synopsis: Name refers to a service group

Declaration: `Property IsGroup : Boolean`

Visibility: published

Access: Read,Write

Description: IsGroup can be set to True to indicate that Name refers to the name of a service group.

See also: TDependency.Name ([311](#))

## 10.19 TWinBindings

### 10.19.1 Description

TWinBindings contains windows-specific properties for the daemon definition (in TDaemonDef.WinBindings ([303](#))). If the daemon should not run on Windows, then the properties can be ignored.

See also: TDaemonDef ([300](#)), TDaemonDef.WinBindings ([303](#))

### 10.19.2 Method overview

Page	Method	Description
<a href="#">312</a>	Assign	Copies all properties
<a href="#">312</a>	Create	Create a new TWinBindings instance
<a href="#">312</a>	Destroy	Remove a TWinBindings instance from memory



### 10.19.3 Property overview

Page	Properties	Access	Description
<a href="#">313</a>	Dependencies	rw	Service dependencies
<a href="#">312</a>	ErrCode	rw	Service specific error code
<a href="#">315</a>	ErrorSeverity	rw	Error severity in case of startup failure
<a href="#">313</a>	GroupName	rw	Service group name
<a href="#">315</a>	IDTag	rw	Location in the service group
<a href="#">314</a>	Password	rw	Password for service startup
<a href="#">315</a>	ServiceType	rw	Type of service
<a href="#">314</a>	StartType	rw	Service startup type.
<a href="#">314</a>	UserName	rw	Username to run service as
<a href="#">314</a>	WaitHint	rw	Timeout wait hint
<a href="#">313</a>	Win32ErrCode	rw	General windows error code

### 10.19.4 TWinBindings.Create

Synopsis: Create a new TWinBindings instance

Declaration: `constructor Create`

Visibility: `public`

Description: `Create` initializes various properties such as the dependencies.

See also: `TDaemonDef` ([300](#)), `TDaemonDef.WinBindings` ([303](#)), `TWinBindings.Dependencies` ([313](#))

### 10.19.5 TWinBindings.Destroy

Synopsis: Remove a TWinBindings instance from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` cleans up the TWinBindings instance.

See also: `TWinBindings.Dependencies` ([313](#)), `TWinBindings.Create` ([312](#))

### 10.19.6 TWinBindings.Assign

Synopsis: Copies all properties

Declaration: `procedure Assign(Source: TPersistent); Override`

Visibility: `public`

Description: `Assign` is overridden by `TWinBindings` so all properties are copied from `Source` to the `TWinBindings` instance.

### 10.19.7 TWinBindings.ErrCode

Synopsis: Service specific error code

Declaration: `Property ErrCode : DWord`

Visibility: `public`

Access: Read,Write

Description: `ErrCode` contains a service specific error code that is reported with `TDaemon.ReportStatus` (293) to the windows service manager. If it is zero, then the contents of `Win32ErrCode` (313) are reported. If it is nonzero, then the windows-errorcode is set to `ERROR_SERVICE_SPECIFIC_ERROR`.

See also: `TWinBindings.Win32ErrCode` (313)

### 10.19.8 `TWinBindings.Win32ErrCode`

Synopsis: General windows error code

Declaration: `Property Win32ErrCode : DWord`

Visibility: public

Access: Read,Write

Description: `Win32ErrCode` is a general windows service error code that can be reported with `TDaemon.ReportStatus` (293) to the windows service manager. It is sent if `ErrCode` (312) is zero.

See also: `ErrCode` (312)

### 10.19.9 `TWinBindings.Dependencies`

Synopsis: Service dependencies

Declaration: `Property Dependencies : TDependencies`

Visibility: published

Access: Read,Write

Description: `Dependencies` contains the list of other services (or service groups) that this service depends on. Windows will first attempt to start these services prior to starting this service. If they cannot be started, then the service will not be started either.

This property is only used during installation of the service.

### 10.19.10 `TWinBindings.GroupName`

Synopsis: Service group name

Declaration: `Property GroupName : string`

Visibility: published

Access: Read,Write

Description: `GroupName` specifies the name of a service group that the service belongs to. If it is empty, then the service does not belong to any group.

This property is only used during installation of the service.

See also: `TDependency.IsGroup` (311)

### 10.19.11 TWinBindings.Password

Synopsis: Password for service startup

Declaration: `Property Password : string`

Visibility: published

Access: Read,Write

Description: `Password` contains the service password: if the service is started with credentials other than one of the system users, then the password for the user must be entered here.

This property is only used during installation of the service.

See also: `UserName` ([314](#))

### 10.19.12 TWinBindings.UserName

Synopsis: Username to run service as

Declaration: `Property UserName : string`

Visibility: published

Access: Read,Write

Description: `Username` specifies the name of a user whose credentials should be used to run the service. If it is left empty, the service is run as the system user. The password can be set in the `Password` ([314](#)) property.

This property is only used during installation of the service.

See also: `Password` ([314](#))

### 10.19.13 TWinBindings.StartType

Synopsis: Service startup type.

Declaration: `Property StartType : TStartType`

Visibility: published

Access: Read,Write

Description: `StartType` specifies when the service should be started during system startup.

This property is only used during installation of the service.

### 10.19.14 TWinBindings.WaitHint

Synopsis: Timeout wait hint

Declaration: `Property WaitHint : Integer`

Visibility: published

Access: Read,Write

Description: `WaitHint` specifies the estimated time for a start/stop/pause or continue operation (in milliseconds). `ReportStatus` should be called prior to this time to report the next status.

See also: `TDaemon.ReportStatus` ([293](#))

### **10.19.15 TWinBindings.IDTag**

Synopsis: Location in the service group

Declaration: `Property IDTag : DWord`

Visibility: published

Access: Read,Write

Description: `IDTag` contains the location of the service in the service group after installation of the service. It should not be set, it is reported by the service manager.

This property is only used during installation of the service.

### **10.19.16 TWinBindings.ServiceType**

Synopsis: Type of service

Declaration: `Property ServiceType : TServiceType`

Visibility: published

Access: Read,Write

Description: `ServiceType` specifies what kind of service is being installed.

This property is only used during installation of the service.

### **10.19.17 TWinBindings.ErrorSeverity**

Synopsis: Error severity in case of startup failure

Declaration: `Property ErrorSeverity : TErrorSeverity`

Visibility: published

Access: Read,Write

Description: `ErrorSeverity` can be used at installation time to tell the windows service manager how to behave when the service fails to start during system startup.

This property is only used during installation of the service.

# Chapter 11

## Reference for unit 'DB'

### 11.1 Used units

Table 11.1: Used units by unit 'DB'

Name	Page
Classes	??
FmtBCD	??
MaskUtils	??
sysutils	??
Variants	??

### 11.2 Overview

The `db` unit provides the basis for all database access mechanisms. It introduces abstract classes, on which all database access mechanisms are based: `TDataset` (371) representing a set of records from a database, `TField` (421) which represents the contents of a field in a record, `TDataSource` (409) which acts as an event distributor on behalf of a dataset and `TParams` (497) which can be used to parameterize queries. The databases connections themselves are abstracted in the `TDatabase` (362) class.

### 11.3 Constants, types and variables

#### 11.3.1 Constants

```
DefaultFieldClasses : Array[TFieldType] of TFieldClass = (Tfield,  
    TStringField, TSmallIntField, TLongIntField, TWordField, TBooleanField  
    , TFloatField, TCurrencyField, TBCDField, TDateField, TTimeField,  
    TDateTimeField, TBytesField, TVarBytesField, TAutoIncField, TBlobField  
    , TMemoField, TGraphicField, TBlobField, TBlobField, TBlobField, TBlobField  
    , Nil, TStringField, TWideStringField, TLargeIntField, Nil, Nil, Nil  
    , Nil, TBlobField, TMemoField, TVariantField, Nil, Nil, TGUIDField  
    , Nil, TFMTCBCDField, TWideStringField, TWideMemoField)
```

DefaultFieldClasses contains the TField (421) descendent class to use when a TDataset instance needs to create fields based on the TFieldDefs (451) field definitions when opening the dataset. The entries can be set to create customized TField descendents for certain field datatypes in all datasets.

```
dsEditModes = [dsEdit, dsInsert, dsSetKey]
```

dsEditModes contains the various values of TDataset.State (398) for which the dataset is in edit mode, i.e. states in which it is possible to set field values for that dataset.

```
dsMaxBufferCount = MAXINT div 8
```

Maximum data buffers count for dataset

```
dsMaxStringSize = 8192
```

Maximum size of string fields

```
dsWriteModes = [dsEdit, dsInsert, dsSetKey, dsCalcFields, dsFilter
, dsNewValue, dsInternalCalc, dsRefreshFields]
```

dsWriteModes contains the various values of TDataset.State (398) for which data can be written to the dataset buffer.

```
Fieldtypenames : Array[TFieldType] of string = ('Unknown', 'String'
, 'Smallint', 'Integer', 'Word', 'Boolean', 'Float', 'Currency', 'BCD'
, 'Date', 'Time', 'DateTime', 'Bytes', 'VarBytes', 'AutoInc', 'Blob'
, 'Memo', 'Graphic', 'FmtMemo', 'ParadoxOle', 'DBaseOle', 'TypedBinary'
, 'Cursor', 'FixedChar', 'WideString', 'Largeint', 'ADT', 'Array'
, 'Reference', 'DataSet', 'OraBlob', 'OraClob', 'Variant', 'Interface'
, 'IDispatch', 'Guid', 'TimeStamp', 'FMTBcd', 'FixedWideChar', 'WideMemo'
)
```

FieldTypeName contains the names (in English) for the various field data types.

```
FieldTypetoVariantMap : Array[TFieldType] of Integer = (varError,
varOleStr, varSmallint, varInteger, varSmallint, varBoolean, varDouble
, varCurrency, varCurrency, varDate, varDate, varDate, varOleStr,
varOleStr, varInteger, varOleStr, varOleStr, varOleStr, varOleStr
, varOleStr, varOleStr, varOleStr, varError, varOleStr, varOleStr
, varint64, varError, varError, varError, varError, varOleStr, varOleStr
, varVariant, varUnknown, varDispatch, varOleStr, varOleStr, varDouble
, varOleStr, varOleStr)
```

FieldTypetoVariantMap contains for each field datatype the variant value type that corresponds to it. If a field type cannot be expressed by a variant type, then varError is stored in the variant value.

```
ftBlobTypes = [ftBlob, ftMemo, ftGraphic, ftFmtMemo, ftParadoxOle
, ftDBaseOle, ftTypedBinary, ftOraBlob, ftOraClob, ftWideMemo]
```

ftBlobTypes is a constant containing all blob field data types. It is to be preferred over the TBlobType (319) range, which contains some non-blob types as well.

```
SQLDelimiterCharacters = [';', ', ', ' ', '(', ')', #13, #10, #9]
```

SQL statement delimiter token characters

```
YesNoChars : Array[Boolean] of Char = ('N', 'Y')
```

Array of characters mapping a boolean to Y/N

### 11.3.2 Types

```
LargeInt = Int64
```

Large (64-bit) integer

```
PBookmarkFlag = ^TBookmarkFlag
```

PBookmarkFlag is a convenience type, defined for internal use in TDataset (371) or one of its descendents.

```
PBufferList = ^TBufferList
```

PBufferList is a pointer to a structure of type TBufferList (319). It is an internal type, and should not be used in end-user code.

```
PDateTimeRec = ^TDateTimeRec
```

Pointer to TDateTimeRec record

```
PLargeInt = ^LargeInt
```

Pointer to Large (64-bit) integer

```
PLookupListRec = ^TLookupListRec
```

Pointer to TLookupListRec record

```
TBlobData = TBytes
```

TBlobData should never be used directly in application code.

```
TBlobStreamMode = (bmRead, bmWrite, bmReadWrite)
```

Table 11.2: Enumeration values for type TBlobStreamMode

Value	Explanation
bmRead	Read blob data
bmReadWrite	Read and write blob data
bmWrite	Write blob data

TBlobStramMode is used when creating a stream for reading BLOB data. It indicates what the data will be used for: reading, writing or both.

`TBlobType = ftBlob..ftWideMemo` deprecated

`TBlobType` is a subrange type, indicating the various datatypes of BLOB fields.

`TBookMark = TBytes`

`TBookMark` is the type used by the `TDataset.SetBookMark` (371) method. It is an opaque type, and should not be used any more, it is superseded by the `TBookmarkStr` (319) type.

`TBookmarkFlag = (bfCurrent, bfBOF, bfEOF, bfInserted)`

Table 11.3: Enumeration values for type `TBookmarkFlag`

Value	Explanation
<code>bfBOF</code>	First record in the dataset.
<code>bfCurrent</code>	Buffer used for the current record
<code>bfEOF</code>	Last record in the dataset
<code>bfInserted</code>	Buffer used for insert

`TBookmarkFlag` is used internally by `TDataset` (371) and it's descendent types to mark the internal memory buffers. It should not be used in end-user applications.

`TBookmarkStr = ansistring`

`TBookmarkStr` is the type used by the `TDataset.Bookmark` (392) property. It can be used as a string, but should in fact be considered an opaque type.

`TBufferArray = ^TRecordBuffer`

`TBufferArray` is an internally used type. It can change in future implementations, and should not be used in application code.

`Array = Array[0..dsMaxBufferCount-1] of TRecordBuffer`

`TBufferList` is used internally by the `TDataset` (371) class to manage the memory buffers for the data. It should not be necessary to use this type in end-user applications.

`TDataAction = (daFail, daAbort, daRetry)`

Table 11.4: Enumeration values for type `TDataAction`

Value	Explanation
<code>daAbort</code>	The operation should be aborted (edits are undone, and an <code>EAbort</code> exception is raised)
<code>daFail</code>	The operation should fail (an exception will be raised)
<code>daRetry</code>	Retry the operation.

`TDataAction` is used by the `TDataSetErrorEvent` (321) event handler prototype. The parameter `Action` of this event handler is of `TDataAction` type, and should indicate what action must be taken by the dataset.



`TDatabaseClass = Class of TDataBase`

`TDatabaseClass` is the class pointer for the `TDatabase` (362) class.

`TDataChangeEvent = procedure(Sender: TObject; Field: TField) of object`

`TDataChangeEvent` is the event handler prototype for the `TDataSource.OnDataChange` (413) event. The sender parameter is the `TDataSource` instance that triggered the event, and the `Field` parameter is the field whose data has changed. If the dataset has scrolled, then the `Field` parameter is `Nil`.

`TDataEvent = (deFieldChange, deRecordChange, deDataSetChange, deDataSetScroll, deLayoutChange, deUpdateRecord, deUpdateState, deCheckBrowseMode, dePropertyChange, deFieldListChange, deFocusControl, deParentScroll, deConnectChange, deReconcileError, deDisabledStateChange)`

Table 11.5: Enumeration values for type `TDataEvent`

Value	Explanation
<code>deCheckBrowseMode</code>	The browse mode is being checked
<code>deConnectChange</code>	Unused
<code>deDataSetChange</code>	The dataset property changed
<code>deDataSetScroll</code>	The dataset scrolled to another record
<code>deDisabledStateChange</code>	Unused
<code>deFieldChange</code>	A field value changed
<code>deFieldListChange</code>	Event sent when the list of fields of a dataset changes
<code>deFocusControl</code>	Event sent whenever a control connected to a field should be focused
<code>deLayoutChange</code>	The layout properties of one of the fields changed
<code>deParentScroll</code>	Unused
<code>dePropertyChange</code>	Unused
<code>deReconcileError</code>	Unused
<code>deRecordChange</code>	The current record changed
<code>deUpdateRecord</code>	The record is being updated
<code>deUpdateState</code>	The dataset state is updated

`TDataEvent` describes the various events that can be sent to `TDataSource` (409) instances connected to a `TDataset` (371) instance.

`TDataOperation = procedure of object`

`TDataOperation` is a prototype handler used internally in `TDataset`. It can be changed at any time, so it should not be used in end-user code.

`TDatasetClass = Class of TDataSet`

`TDatasetClass` is the class type for the `TDataset` (371) class. It is currently unused in the DB unit and is defined for the benefit of other units.

```

TDataSetErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError
;
                                var DataAction: TDataAction) of object

```

`TDataSetErrorEvent` is used by the `TDataSet.OnEditError` (406), `TDataSet.OnPostError` (408) and `TDataSet.OnDeleteError` (406) event handlers to allow the programmer to specify what should be done if an update operation fails with an exception: The `DataSet` parameter indicates what dataset triggered the event, the `E` parameter contains the exception object. The `DataAction` must be set by the event handler, and based on its return value, the dataset instance will take appropriate action. The default value is `daFail`, i.e. the exception will be raised again. For a list of available return values, see `TDataAction` (319).

```

TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object

```

`TDataSetNotifyEvent` is used in most of the `TDataSet` (371) event handlers. It differs from the more general `TNotifyEvent` (defined in the `Classes` unit) in that the `Sender` parameter of the latter is replaced with the `DataSet` parameter. This avoids typecasts, the available `TDataSet` methods can be used directly.

```

TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
                 dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue
                 ,
                 dsBlockRead, dsInternalCalc, dsOpening, dsRefreshFields
                 )

```

Table 11.6: Enumeration values for type `TDataSetState`

Value	Explanation
<code>dsBlockRead</code>	The dataset is open, but no events are transferred to datasources.
<code>dsBrowse</code>	The dataset is active, and the cursor can be used to navigate the data.
<code>dsCalcFields</code>	The dataset is calculating its calculated fields.
<code>dsCurValue</code>	The dataset is showing the current values of a record.
<code>dsEdit</code>	The dataset is in editing mode: the current record can be modified.
<code>dsFilter</code>	The dataset is filtering records.
<code>dsInactive</code>	The dataset is not active. No data is available.
<code>dsInsert</code>	The dataset is in insert mode: the current record is a new record which can be edited.
<code>dsInternalCalc</code>	The dataset is calculating its internally calculated fields.
<code>dsNewValue</code>	The dataset is showing the new values of a record.
<code>dsOldValue</code>	The dataset is showing the old values of a record.
<code>dsOpening</code>	The dataset is currently opening, but is not yet completely open.
<code>dsRefreshFields</code>	Dataset is refreshing field values from server after an update.
<code>dsSetKey</code>	The dataset is calculating the primary key.

`TDataSetState` describes the current state of the dataset. During its lifetime, the dataset's state is described by these enumerated values.

Some state are not used in the default `TDataSet` implementation, and are only used by certain descendants.

```

TDateTimeAlias = TDateTime

```

TDateTimeAlias is no longer used.

```
TDateTimeRec = record
case TFieldType of
ftDate: (
    Date : LongInt
    ;
);
ftTime: (
    Time : LongInt;
);
ftDateTime: (
    DateTime : TDateTimeAlias
    ;
);
end
```

TDateTimeRec was used by older TDataSet (371) implementations to store date/time values. Newer implementations use the TDateTime. This type should no longer be used.

TDBDatasetClass = Class of TDBDataset

TDBDatasetClass is the class pointer for TDBDataset (415)

TDBTransactionClass = Class of TDBTransaction

TDBTransactionClass is the class pointer for the TDBTransaction (417) class.

```
TFieldAttribute = (faHiddenCol, faReadonly, faRequired, faLink, faUnNamed
,
    faFixed)
```

Table 11.7: Enumeration values for type TFieldAttribute

Value	Explanation
faFixed	Fixed length field
faHiddenCol	Field is a hidden column (used to construct a unique key)
faLink	Field is a link field for other datasets
faReadonly	Field is read-only
faRequired	Field is required
faUnNamed	Field has no original name

TFieldAttribute is used to denote some attributes of a field in a database. It is used in the Attributes (450) property of TFieldDef (447).

TFieldAttributes = Set of TFieldAttribute

TFieldAttributes is used in the TFieldDef.Attributes (450) property to denote additional attributes of the underlying field.

`TFieldChars = Set of Char`

`TFieldChars` is a type used in the `TField.ValidChars` (438) property. It's a simple set of characters.

`TFieldClass = Class of TField`

`TFieldDefClass = Class of TFieldDef`

`TFieldDefClass` is used to be able to customize the actual `TDataset.FieldDefs` (395) items class.

`TFieldDefsClass = Class of TFieldDefs`

`TFieldDefClass` is used to be able to customize the actual `TDataset.FieldDefs` (395) class used in a `TDataset` (371) descendent.

`TFieldGetTextEvent = procedure(Sender: TField; var aText: string;  
DisplayText: Boolean) of object`

`TFieldGetTextEvent` is the prototype for the `TField.OnGetText` (446) event handler. It should be used when the text of a field requires special formatting. The event handler should return the contents of the field in formatted form in the `aText` parameter. The `DisplayText` is `True` if the text is used for displaying purposes or is `False` if it will be used for editing purposes.

`TFieldKind = (fkData, fkCalculated, fkLookup, fkInternalCalc)`

Table 11.8: Enumeration values for type `TFieldKind`

Value	Explanation
<code>fkCalculated</code>	The field is calculated on the fly.
<code>fkData</code>	Field represents actual data in the underlying data structure.
<code>fkInternalCalc</code>	Field is calculated but stored in an underlying buffer.
<code>fkLookup</code>	The field is a lookup field.

`TFieldKind` indicates the type of a `TField` instance. Besides `TField` instances that represent fields present in the underlying data records, there can also be calculated or lookup fields. To distinguish between these kind of fields, `TFieldKind` is introduced.

`TFieldKinds = Set of TFieldKind`

`TFieldKinds` is a set of `TFieldKind` (323) values. It is used internally by the classes of the DB unit.

`Array = Array[TFieldType] of Byte`

`TFieldMap` is no longer used.

`TFieldNotifyEvent = procedure(Sender: TField) of object`

TFieldNotifyEvent is a prototype for the event handlers in the TField (421) class. Its Sender parameter is the field instance that triggered the event.

```
TFieldRef = ^TField
```

Pointer to a TField instance

```
TFieldsClass = Class of TFields
```

TFieldsClass is needed to be able to specify the class of fields used in TDataSet.Fields (398);

```
TFieldSetTextEvent = procedure(Sender: TField; const aText: string
    )
                        of object
```

TFieldSetTextEvent is the prototype for an event handler used to set the contents of a field based on a user-edited text. It should be used when the text of a field is entered with special formatting. The event handler should set the contents of the field based on the formatted text in the AText parameter.

```
TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger, ftWord, ftBoolean
    ,
                ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime
    , ftBytes,
                ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic
    , ftFmtMemo,
                ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor
    ,
                ftFixedChar, ftWideString, ftLargeint, ftADT, ftArray
    ,
                ftReference, ftDataSet, ftOraBlob, ftOraClob, ftVariant
    ,
                ftInterface, ftIDispatch, ftGuid, ftTimeStamp, ftFMTBcd
    ,
                ftFixedWideChar, ftWideMemo)
```

Table 11.9: Enumeration values for type TFieldType

Value	Explanation
ftADT	ADT value
ftArray	Array data
ftAutoInc	Auto-increment integer value (4 bytes)
ftBCD	Binary Coded Decimal value (DECIMAL and NUMERIC SQL types)
ftBlob	Binary data value (no type, no size)
ftBoolean	Boolean value
ftBytes	Array of bytes value, fixed size (untyped)
ftCurrency	Currency value (4 decimal points)
ftCursor	Cursor data value (no size)
ftDataSet	Dataset data (blob)
ftDate	Date value
ftDateTime	Date/Time (timestamp) value
ftDBaseOle	Paradox OLE field data
ftFixedChar	Fixed character array (string)
ftFixedWideChar	Fixed wide character data (2 bytes per character)
ftFloat	Floating point value (double)
ftFMTBcd	Formatted BCD (Binary Coded Decimal) value.
ftFmtMemo	Formatted memo data value (no size)
ftGraphic	Graphical data value (no size)
ftGuid	GUID data value
ftIDispatch	Dispatch data value
ftInteger	Regular integer value (4 bytes, signed)
ftInterface	interface data value
ftLargeint	Large integer value (8-byte)
ftMemo	Binary text data (no size)
ftOraBlob	Oracle BLOB data
ftOraClob	Oracle CLOB data
ftParadoxOle	Paradox OLE field data (no size)
ftReference	Reference data
ftSmallint	Small integer value(1 byte, signed)
ftString	String data value (ansistring)
ftTime	Time value
ftTimeStamp	Timestamp data value
ftTypedBinary	Binary typed data (no size)
ftUnknown	Unknown data type
ftVarBytes	Array of bytes value, variable size (untyped)
ftVariant	Variant data value
ftWideMemo	Widestring memo data
ftWideString	Widestring (2 bytes per character)
ftWord	Word-sized value(2 bytes, unsigned)

TFieldType indicates the type of a TField ([421](#)) underlying data, in the DataType ([435](#)) property.

TFilterOption = (foCaseInsensitive, foNoPartialCompare)

Table 11.10: Enumeration values for type TFilterOption

Value	Explanation
foCaseInsensitive	Filter case insensitively.
foNoPartialCompare	Do not compare values partially, always compare completely.

TFilterOption enumerates the various options available when filtering a dataset. The TFilterOptions (326) set is used in the TDataset.FilterOptions (399) property to indicate which of the options should be used when filtering the data.

TFilterOptions = Set of TFilterOption

TFilterOption is the set of filter options to use when filtering a dataset. This set type is used in the TDataset.FilterOptions (399) property. The available values are described in the TFilterOption (325) type.

```
TFilterRecordEvent = procedure (DataSet: TDataSet; var Accept: Boolean
    )
                        of object
```

TFilterRecordEvent is the prototype for the TDataset.OnFilterRecord (407) event handler. The DataSet parameter indicates which dataset triggered the event, and the Accept parameter must be set to true if the current record should be shown, False should be used when the record should be hidden.

TGetMode = (gmCurrent, gmNext, gmPrior)

Table 11.11: Enumeration values for type TGetMode

Value	Explanation
gmCurrent	Retrieve the current record
gmNext	Retrieve the next record.
gmPrior	Retrieve the previous record.

TGetMode is used internally by TDataset (371) when it needs to fetch more data for its buffers (using GetRecord). It tells the descendent dataset what operation must be performed.

TGetResult = (grOK, grBOF, grEOF, grError)

Table 11.12: Enumeration values for type TGetResult

Value	Explanation
grBOF	The beginning of the recordset is reached
grEOF	The end of the recordset is reached.
grError	An error occurred
grOK	The operation was completed successfully

`TGetResult` is used by descendents of `TDataset` (371) when they have to communicate the result of the `GetRecord` operation back to the `TDataset` record.

```
TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive
,
                ixExpression, ixNonMaintained)
```

Table 11.13: Enumeration values for type `TIndexOption`

Value	Explanation
<code>ixCaseInsensitive</code>	The values in the index are sorted case-insensitively
<code>ixDescending</code>	The values in the index are sorted descending.
<code>ixExpression</code>	The values in the index are based on a calculated expression.
<code>ixNonMaintained</code>	The index is non-maintained, i.e. changing the data will not update the index.
<code>ixPrimary</code>	The index is the primary index for the data
<code>ixUnique</code>	The index is a unique index, i.e. each index value can occur only once

`TIndexOption` describes the various properties that an index can have. It is used in the `TIndexOptions` (327) set type to describe all properties of an index definition as in `TIndexDef` (467).

```
TIndexOptions = Set of TIndexOption
```

`TIndexOptions` contains the set of properties that an index can have. It is used in the `TIndexDef.Options` (469) property to describe all properties of an index definition as in `TIndexDef` (467).

```
TLocateOption = (loCaseInsensitive, loPartialKey)
```

Table 11.14: Enumeration values for type `TLocateOption`

Value	Explanation
<code>loCaseInsensitive</code>	Perform a case-insensitive search
<code>loPartialKey</code>	Accept partial key matches for string fields

`TLocateOption` is used in the `TDataset.Locate` (387) call to enumerate the possible options available when locating a record in the dataset.

For string-type fields, this option indicates that fields starting with the search value are considered a match. For other fields (e.g. integer, date/time), this option is ignored and only equal field values are considered a match.

```
TLocateOptions = Set of TLocateOption
```

`TLocateOptions` is used in the `TDataset.Locate` (387) call: It should contain the actual options to use when locating a record in the dataset.

```
TLoginEvent = procedure(Sender: TObject; Username: string;
    Password: string) of object
```



`TLoginEvent` is the prototype for the `TCustomConnection.OnLogin` (361) event handler. It gets passed the `TCustomConnection` instance that is trying to login, and the initial username and password.

`Array = Array of Integer`

`TParamBinding` is an auxiliary type used when parsing and binding parameters in SQL statements. It should never be used directly in application code.

`TParamClass = Class of TParam`

`TParamClass` is needed to be able to specify the type of parameters when instantiating a `TParams` (497) collection.

`TParamStyle = (psInterbase, psPostgreSQL, psSimulated)`

Table 11.15: Enumeration values for type `TParamStyle`

Value	Explanation
<code>psInterbase</code>	Parameters are specified by a ? character
<code>psPostgreSQL</code>	Parameters are specified by a \$N character.
<code>psSimulated</code>	Parameters are specified by a \$N character.

`TParamStyle` denotes the style in which parameters are specified in a query. It is used in the `TParams.ParseSQL` (500) method, and can have the following values:

**`psInterbase`** Parameters are specified by a ? character

**`psPostgreSQL`** Parameters are specified by a \$N character.

**`psSimulated`** Parameters are specified by a \$N character.

`TParamType = (ptUnknown, ptInput, ptOutput, ptInputOutput, ptResult)`

Table 11.16: Enumeration values for type `TParamType`

Value	Explanation
<code>ptInput</code>	Input parameter
<code>ptInputOutput</code>	Input/output parameter
<code>ptOutput</code>	Output parameter, filled on result
<code>ptResult</code>	Result parameter
<code>ptUnknown</code>	Unknown type

`TParamType` indicates the kind of parameter represented by a `TParam` (484) instance. it has one of the following values:

**`ptUnknown`** Unknown type

**`ptInput`** Input parameter

**ptOutput** Output parameter, filled on result

**ptInputOutput** Input/output parameter

**ptResult** Result parameter

`TParamTypes = Set of TParamType`

`TParamTypes` is defined for completeness: a set of `TParamType` (328) values.

`TProviderFlag = (pfInUpdate, pfInWhere, pfInKey, pfHidden, pfRefreshOnInsert, pfRefreshOnUpdate)`

Table 11.17: Enumeration values for type `TProviderFlag`

Value	Explanation
<code>pfHidden</code>	
<code>pfInKey</code>	Field is a key field and used in the WHERE clause of an update statement
<code>pfInUpdate</code>	Changes to the field should be propagated to the database.
<code>pfInWhere</code>	Field should be used in the WHERE clause of an update statement in case of <code>upWhereChanged</code> .
<code>pfRefreshOnInsert</code>	This field's value should be refreshed after insert.
<code>pfRefreshOnUpdate</code>	This field's value should be refreshed after update.

`TProviderFlag` describes how the field should be used when applying updates from a dataset to the database. Each field of a `TDataset` (371) has one or more of these flags.

`TProviderFlags = Set of TProviderFlag`

`TProviderFlags` is used for the `TField.ProviderFlags` (445) property to describe the role of the field when applying updates to a database.

`TPSCommandType = (ctUnknown, ctQuery, ctTable, ctStoredProc, ctSelect, ctInsert, ctUpdate, ctDelete, ctDDL)`

Table 11.18: Enumeration values for type `TPSCommandType`

Value	Explanation
<code>ctDDL</code>	SQL DDL statement
<code>ctDelete</code>	SQL DELETE Statement
<code>ctInsert</code>	SQL INSERT Statement
<code>ctQuery</code>	General SQL statement
<code>ctSelect</code>	SQL SELECT Statement
<code>ctStoredProc</code>	Stored procedure statement
<code>ctTable</code>	Table contents (select * from table)
<code>ctUnknown</code>	Unknown SQL type or not SQL based
<code>ctUpdate</code>	SQL UPDATE statement

`TPSCommandType` is used in the `IProviderSupport.PSGetCommandType` (339) call to determine the type of SQL command that the provider is exposing. It is meaningless for datasets that are not SQL based.

`TRecordBuffer = PAnsiChar`

`TRecordBuffer` is the type used by `TDataset` (371) to point to a record's data buffer. It is used in several internal `TDataset` routines.

`TRecordBufferBaseType = AnsiChar`

`TRecordBufferBaseType` should not be used directly. It just serves as an (opaque) base type to `TRecordBuffer` (330)

`TResolverResponse = (rrSkip, rrAbort, rrMerge, rrApply, rrIgnore)`

Table 11.19: Enumeration values for type `TResolverResponse`

Value	Explanation
<code>rrAbort</code>	Abort the whole update process, no error message is displayed (no <code>EAbort</code> exception raised).
<code>rrApply</code>	Replace the update with new values applied by the event handler
<code>rrIgnore</code>	Ignore the error and remove update from change log
<code>rrMerge</code>	Merge the update with existing changes on the server.
<code>rrSkip</code>	Skip the current update, leave it in the change log.

`TResolverResponse` is used to indicate what should happen to a pending change that could not be resolved. It is used in callbacks.

`TResyncMode = Set of (rmExact, rmCenter)`

Table 11.20: Enumeration values for type

Value	Explanation
<code>rmCenter</code>	Try to position the cursor in the middle of the buffer
<code>rmExact</code>	Reposition at exact the same location in the buffer

`TResyncMode` is used internally by various `TDataset` (371) navigation and data manipulation methods such as the `TDataset.Refresh` (390) method when they need to reset the cursor position in the dataset's buffer.

`Array = Array[0..dsMaxStringSize] of AnsiChar`

Type to access string field content buffers as an array of characters

`TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied)`

Table 11.21: Enumeration values for type TUpdateAction

Value	Explanation
uaAbort	The whole update operation should abort
uaApplied	Consider the update as applied
uaFail	Update operation should fail
uaRetry	Retry the update operation
uaSkip	The update of the current record should be skipped. (but not discarded)

TUpdateAction indicates what action must be taken in case the applying of updates on the underlying database fails. This type is not used in the TDataset (371) class, but is defined on behalf of TDataset descendents that implement caching of updates: It indicates what should be done when the (delayed) applying of the updates fails. This event occurs long after the actual post or delete operation.

TUpdateKind = (ukModify, ukInsert, ukDelete)

Table 11.22: Enumeration values for type TUpdateKind

Value	Explanation
ukDelete	Delete a record in the database.
ukInsert	insert a new record in the database.
ukModify	Modify an existing record in the database.

TUpdateKind indicates what kind of update operation is in progress when applying updates.

TUpdateMode = (upWhereAll, upWhereChanged, upWhereKeyOnly)

Table 11.23: Enumeration values for type TUpdateMode

Value	Explanation
upWhereAll	Use all old field values
upWhereChanged	Use only old field values of modified fields
upWhereKeyOnly	Only use key fields in the where clause.

TUpdateMode determines how the WHERE clause of update queries for SQL databases should be constructed.

TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted)

Table 11.24: Enumeration values for type TUpdateStatus

Value	Explanation
usDeleted	Record exists in the database, but is locally deleted.
usInserted	Record does not yet exist in the database, but is locally inserted
usModified	Record exists in the database but is locally modified
usUnmodified	Record is unmodified

TUpdateStatus determines the current state of the record buffer, if updates have not yet been applied to the database.

TUpdateStatusSet = Set of TUpdateStatus

TUpdateStatusSet is a set of TUpdateStatus (331) values.

### 11.3.3 Variables

```
LoginDialogExProc : function(const ADatabaseName: string; var AUserName
    : string;
    var APassword: string; UserNameReadOnly: Boolean
    ) : Boolean = Nil
```

LoginDialogExProc is a procedural variable that can be set to handle login dialogs: if a database connection component needs to collect login data (typically when LoginPrompt is True), then if this callback is set it can e.g. be used to show a dialog used to fetch the data.

## 11.4 Procedures and functions

### 11.4.1 BuffersEqual

Synopsis: Check whether 2 memory buffers are equal

Declaration: `function BuffersEqual(Buf1: Pointer; Buf2: Pointer; Size: Integer) : Boolean`

Visibility: default

Description: BuffersEqual compares the memory areas pointed to by the Buf1 and Buf2 pointers and returns True if the contents are equal. The memory areas are compared for the first Size bytes. If all bytes in the indicated areas are equal, then True is returned, otherwise False is returned.

Errors: If Buf1 or Buf2 do not point to a valid memory area or Size is too large, then an exception may occur

See also: #rtl.sysutils.Comparemem (??)

### 11.4.2 DatabaseError

Synopsis: Raise an EDatabaseError exception.

Declaration: `procedure DatabaseError(const Msg: string); Overload`  
`procedure DatabaseError(const Msg: string; Comp: TComponent); Overload`

Visibility: default

Description: DatabaseError raises an EDatabaseError (335) exception, passing it Msg. If Comp is specified, the name of the component is prepended to the message.

See also: DatabaseErrorFmt (333), EDatabaseError (335)

### 11.4.3 DatabaseErrorFmt

Synopsis: Raise an `EDatabaseError` exception with a formatted message

Declaration: 

```
procedure DatabaseErrorFmt(const Fmt: string;
                           const Args: Array of const); Overload
procedure DatabaseErrorFmt(const Fmt: string;
                           const Args: Array of const; Comp: TComponent)
                           ; Overload
```

Visibility: default

Description: `DatabaseErrorFmt` raises an `EDatabaseError` (335) exception, passing it a message made by calling `#rtl.sysutils.format` (??) with the `fmt` and `Args` arguments. If `Comp` is specified, the name of the component is prepended to the message.

See also: `DatabaseError` (332), `EDatabaseError` (335)

### 11.4.4 DateTimeRecToDateTime

Synopsis: Convert `TDateTimeRec` record to a `TDateTime` value.

Declaration: 

```
function DateTimeRecToDateTime(DT: TFieldType; Data: TDateTimeRec)
                               : TDateTime
```

Visibility: default

Description: `DateTimeRecToDateTime` examines `Data` and `Dt` and uses `dt` to convert the timestamp in `Data` to a `TDateTime` value.

See also: `TFieldType` (324), `TDateTimeRec` (322), `DateTimeToDateTimeRec` (333)

### 11.4.5 DateTimeToDateTimeRec

Synopsis: Convert `TDateTime` value to a `TDateTimeRec` record.

Declaration: 

```
function DateTimeToDateTimeRec(DT: TFieldType; Data: TDateTime)
                               : TDateTimeRec
```

Visibility: default

Description: `DateTimeToDateTimeRec` examines `Data` and `Dt` and uses `dt` to convert the date/time value in `Data` to a `TDateTimeRec` record.

See also: `TFieldType` (324), `TDateTimeRec` (322), `DateTimeRecToDateTime` (333)

### 11.4.6 DisposeMem

Synopsis: Dispose of a heap memory block and `Nil` the pointer (deprecated)

Declaration: 

```
procedure DisposeMem(var Buffer; Size: Integer)
```

Visibility: default

Description: `DisposeMem` disposes of the heap memory area pointed to by `Buffer` (`Buffer` must be of type `Pointer`). The `Size` parameter indicates the size of the memory area (it is, in fact, ignored by the heap manager). The pointer `Buffer` is set to `Nil`. If `Buffer` is `Nil`, then nothing happens. Do not use `DisposeMem` on objects, because their destructor will not be called.

**Errors:** If `Buffer` is not pointing to a valid heap memory block, then memory corruption may occur.

See also: `#rtl.system.FreeMem` (??), `#rtl.sysutils.freeandnil` (??)

### 11.4.7 enumerator(TDataSet):TDataSetEnumerator

**Synopsis:** Operator to return dataset enumerator

**Declaration:** `operator enumerator(ADataset: TDataSet) : TDataSetEnumerator`

**Visibility:** default

**Description:** This operator allows to use the `TDataSetEnumerator` (408) as an enumerator for a `TDataset` (371)

See also: `TDataSetEnumerator` (408), `TDataset` (371)

### 11.4.8 ExtractFieldName

**Synopsis:** Extract the field name at position

**Declaration:** `function ExtractFieldName(const Fields: string; var Pos: Integer)  
: string`

**Visibility:** default

**Description:** `ExtractFieldName` returns the string starting at position `Pos` till the next semicolon (;) character or the end of the string. On return, `Pos` contains the position of the first character after the semicolon character (or one more than the length of the string).

See also: `TFields.GetFieldList` (454)

### 11.4.9 SkipComments

**Synopsis:** Skip SQL comments

**Declaration:** `function SkipComments(var p: PChar; EscapeSlash: Boolean;  
EscapeRepeat: Boolean) : Boolean`

**Visibility:** default

**Description:** `SkipComments` examines the null-terminated string in `P` and skips any SQL comment or string literal found at the start. It returns `P` the first non-comment or non-string literal position. The `EscapeSlash` parameter determines whether the backslash character (\) functions as an escape character (i.e. the following character is not considered a delimiter). `EscapeRepeat` must be set to `True` if the quote character is repeated to indicate itself.

The function returns `True` if a comment was found and skipped, `False` otherwise.

**Errors:** No checks are done on the validity of `P`.

See also: `TParams.ParseSQL` (500)

## 11.5 TLookupListRec

```
TLookupListRec = record
  Key : Variant;
  Value : Variant;
end
```

TLookupListRec is used by lookup fields to store lookup results, if the results should be cached. Its two fields keep the key value and associated lookup value.

## 11.6 EDatabaseError

### 11.6.1 Description

EDatabaseError is the base class from which database-related exception classes should derive. It is raised by the DatabaseError (332) call.

See also: DatabaseError (332), DatabaseErrorFmt (333)

## 11.7 EUpdateError

### 11.7.1 Description

EupdateError is an exception used by the TProvider database support. It should never be raised directly.

See also: EDatabaseError (335)

### 11.7.2 Method overview

Page	Method	Description
<a href="#">335</a>	Create	Create a new EUpdateError instance
<a href="#">336</a>	Destroy	Free the EupdateError instance

### 11.7.3 Property overview

Page	Properties	Access	Description
<a href="#">336</a>	Context	r	Context in which exception occurred.
<a href="#">336</a>	ErrorCode	r	Numerical error code.
<a href="#">337</a>	OriginalException	r	Originally raised exception
<a href="#">337</a>	PreviousError	r	Previous error number

### 11.7.4 EUpdateError.Create

Synopsis: Create a new EUpdateError instance

Declaration: constructor Create (NativeError: string; Context: string;  
ErrCode: Integer; PrevError: Integer; E: Exception)

Visibility: public



**Description:** Create instantiates a new `EUpdateError` object and populates the various properties with the `NativeError`, `Context`, `ErrCode` and `PrevError` parameters. The `E` parameter is the actual exception that occurred while the update operation was attempted. The exception object `E` will be freed if the `EUpdateError` instance is freed.

See also: `EDatabaseError` ([335](#))

### 11.7.5 `EUpdateError.Destroy`

**Synopsis:** Free the `EupdateError` instance

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

**Visibility:** `public`

**Description:** `Destroy` frees the original exception object (if there was one) and then calls the inherited destructor.

**Errors:** If the original exception object was already freed, an error will occur.

See also: `EUpdateError.OriginalException` ([337](#))

### 11.7.6 `EUpdateError.Context`

**Synopsis:** Context in which exception occurred.

**Declaration:** `Property Context : string`

**Visibility:** `public`

**Access:** `Read`

**Description:** A description of the context in which the original exception was raised.

See also: `EUpdateError.OriginalException` ([337](#)), `EUpdateError.ErrorCode` ([336](#)), `EUpdateError.PreviousError` ([337](#))

### 11.7.7 `EUpdateError.ErrorCode`

**Synopsis:** Numerical error code.

**Declaration:** `Property ErrorCode : Integer`

**Visibility:** `public`

**Access:** `Read`

**Description:** `ErrorCode` is a numerical error code, provided by the native data access layer, to describe the error. It may or not be filled.

See also: `EUpdateError.OriginalException` ([337](#)), `EUpdateError.Context` ([336](#)), `EUpdateError.PreviousError` ([337](#))

### 11.7.8 EUpdateError.OriginalException

Synopsis: Originally raised exception

Declaration: `Property OriginalException : Exception`

Visibility: public

Access: Read

Description: `OriginalException` is the originally raised exception that is transformed to an `EUpdateError` exception.

See also: `DB.EDatabaseError` ([335](#))

### 11.7.9 EUpdateError.PreviousError

Synopsis: Previous error number

Declaration: `Property PreviousError : Integer`

Visibility: public

Access: Read

Description: `PreviousError` is used to order the errors which occurred during an update operation.

See also: `EUpdateError.ErrorCode` ([336](#)), `EUpdateError.Context` ([336](#)), `EUpdateError.OriginalException` ([337](#))

## 11.8 IProviderSupport

### 11.8.1 Description

`IProviderSupport` is an interface used by Delphi's TProvider (datasnap) technology. It is currently not used in Free Pascal, but is provided for Delphi compatibility. The `TDataset` ([371](#)) class implements all the methods of this interface for the benefit of descendent classes, but does not publish the interface in it's declaration.

See also: `TDataset` ([371](#))

### 11.8.2 Method overview

Page	Method	Description
<a href="#">338</a>	<code>PSEndTransaction</code>	End an active transaction
<a href="#">338</a>	<code>PSExecute</code>	Execute the current command-text.
<a href="#">338</a>	<code>PSExecuteStatement</code>	Execute a SQL statement.
<a href="#">339</a>	<code>PSGetAttributes</code>	Get a list of attributes (metadata)
<a href="#">339</a>	<code>PSGetCommandText</code>	Return the SQL command executed for getting data.
<a href="#">339</a>	<code>PSGetCommandType</code>	Return SQL command type
<a href="#">340</a>	<code>PSGetDefaultOrder</code>	Default order index definition
<a href="#">340</a>	<code>PSGetIndexDefs</code>	Return a list of index definitions
<a href="#">340</a>	<code>PSGetKeyFields</code>	Return a list of key fields in the dataset
<a href="#">340</a>	<code>PSGetParams</code>	Get the parameters in the commandtext
<a href="#">341</a>	<code>PSGetQuoteChar</code>	Quote character for quoted strings
<a href="#">341</a>	<code>PSGetTableName</code>	Name of database table which must be updated
<a href="#">341</a>	<code>PSGetUpdateException</code>	Transform exception to <code>UpdateError</code>
<a href="#">341</a>	<code>PSInTransaction</code>	Is the dataset in an active transaction.
<a href="#">342</a>	<code>PSIsSQLBased</code>	Is the dataset SQL based
<a href="#">342</a>	<code>PSIsSQLSupported</code>	Can the dataset support SQL statements
<a href="#">342</a>	<code>PSReset</code>	Position the dataset on the first record
<a href="#">342</a>	<code>PSSetCommandText</code>	Set the command-text of the dataset
<a href="#">343</a>	<code>PSSetParams</code>	Set the parameters for the command text
<a href="#">343</a>	<code>PSStartTransaction</code>	Start a new transaction
<a href="#">343</a>	<code>PSUpdateRecord</code>	Update a record

### 11.8.3 `IProviderSupport.PSEndTransaction`

Synopsis: End an active transaction

Declaration: `procedure PSEndTransaction (ACommit: Boolean)`

Visibility: default

Description: `PSEndTransaction` ends an active transaction if an transaction is active. (`PSInTransaction` ([316](#)) returns `True`). If `ACommit` is `True` then the transaction is committed, else it is rolled back.

See also: `PSInTransaction` ([316](#)), `PSStartTransaction` ([316](#))

### 11.8.4 `IProviderSupport.PSExecute`

Synopsis: Execute the current command-text.

Declaration: `procedure PSExecute`

Visibility: default

Description: `PSExecute` executes the current SQL statement: the command as it is returned by `PSGetCommandText` ([316](#)).

See also: `PSGetCommandText` ([316](#)), `PSExecuteStatement` ([316](#))

### 11.8.5 `IProviderSupport.PSExecuteStatement`

Synopsis: Execute a SQL statement.

**Declaration:** `function PSExecuteStatement(const ASQL: string; AParams: TParams;  
ResultSet: Pointer) : Integer`

**Visibility:** default

**Description:** `PSExecuteStatement` will execute the ASQL SQL statement in the current transaction. The SQL statement can have parameters embedded in it (in the form `:ParamName`), values for these parameters will be taken from `AParams`. If the SQL statement returns a result-set, then the result set can be returned in `ResultSet`. The function returns `True` if the statement was executed successful. `PSExecuteStatement` does not modify the content of `CommandText`: `PSGetCommandText` (316) returns the same value before and after a call to `PSExecuteStatement`.

See also: `PSGetCommandText` (316), `PSSetCommandText` (316), `PSExecuteStatement` (316)

### 11.8.6 IProviderSupport.PSGetAttributes

**Synopsis:** Get a list of attributes (metadata)

**Declaration:** `procedure PSGetAttributes(List: TList)`

**Visibility:** default

**Description:** `PSGetAttributes` returns a set of name=value pairs which is included in the data packet sent to a client.

See also: `PSGetCommandText` (316)

### 11.8.7 IProviderSupport.PSGetCommandText

**Synopsis:** Return the SQL command executed for getting data.

**Declaration:** `function PSGetCommandText : string`

**Visibility:** default

**Description:** `PSGetCommandText` returns the SQL command that is executed when the `PSExecute` (316) function is called (for a `TSQLQuery` this would be the SQL property) or when the dataset is opened.

See also: `PSExecute` (316), `PSSetCommandText` (316)

### 11.8.8 IProviderSupport.PSGetCommandType

**Synopsis:** Return SQL command type

**Declaration:** `function PSGetCommandType : TPSCCommandType`

**Visibility:** default

**Description:** `PSGetCommandType` should return the kind of SQL statement that is executed by the command (as returned by `PSGetCommandText` (316)). The list of possible command types is enumerated in `TPSCCommandType` (329).

See also: `PSGetCommandText` (316), `TPSCCommandType` (329), `PSExecute` (316)

### 11.8.9 IProviderSupport.PSGetDefaultOrder

Synopsis: Default order index definition

Declaration: `function PSGetDefaultOrder : TIndexDef`

Visibility: default

Description: `PSGetDefaultOrder` should return the index definition from the list of indexes (as returned by `PSGetIndexDefs` (316)) that represents the default sort order.

See also: `PSGetIndexDefs` (316), `PSGetKeyFields` (316)

### 11.8.10 IProviderSupport.PSGetIndexDefs

Synopsis: Return a list of index definitions

Declaration: `function PSGetIndexDefs(IndexTypes: TIndexOptions) : TIndexDefs`

Visibility: default

Description: `PSGetIndexDefs` should return a list of index definitions, limited to the types of indexes in `IndexTypes`.

See also: `PSGetDefaultOrder` (316), `PSGetKeyFields` (316)

### 11.8.11 IProviderSupport.PSGetKeyFields

Synopsis: Return a list of key fields in the dataset

Declaration: `function PSGetKeyFields : string`

Visibility: default

Description: `PSGetKeyFields` returns a semicolon-separated list of fieldnames that make up the unique key for a record. Normally, these are the names of the fields that have `pfInKey` in their `ProviderOptions` (421) property.

See also: `PSGetIndexDefs` (316), `PSGetDefaultOrder` (316), `TField.ProviderOptions` (421), `TProviderFlags` (329)

### 11.8.12 IProviderSupport.PSGetParams

Synopsis: Get the parameters in the commandtext

Declaration: `function PSGetParams : TParams`

Visibility: default

Description: `PSGetParams` returns the list of parameters in the command-text (as returned by `PSGetCommandText` (316)). This is usually the `Params` property of a `TDataset` (371) descendant.

See also: `PSGetCommandText` (316), `PSSetParams` (316)

### 11.8.13 IProviderSupport.PSGetQuoteChar

Synopsis: Quote character for quoted strings

Declaration: `function PSGetQuoteChar : string`

Visibility: default

Description: `PSGetQuoteChar` returns the quote character needed to enclose string literals in an SQL statement for the underlying database.

See also: `PSGetTableName` (316)

### 11.8.14 IProviderSupport.PSGetTableName

Synopsis: Name of database table which must be updated

Declaration: `function PSGetTableName : string`

Visibility: default

Description: `PSGetTableName` returns the name of the table for which update SQL statements must be constructed. The provider can create and execute SQL statements to update the underlying database by itself. For this, it uses `PSGetTableName` as the name of the table to update.

See also: `PSGetQuoteChar` (316)

### 11.8.15 IProviderSupport.PSGetUpdateException

Synopsis: Transform exception to `UpdateError`

Declaration: `function PSGetUpdateException(E: Exception; Prev: EUpdateError)  
: EUpdateError`

Visibility: default

Description: `PSGetUpdateException` is called to transform and chain exceptions that occur during an `ApplyUpdates` operation. The exception `E` must be transformed to an `EUpdateError` (335) exception. The previous `EUpdateError` exception in the update batch is passed in `Prev`.

See also: `EUpdateError` (335)

### 11.8.16 IProviderSupport.PSInTransaction

Synopsis: Is the dataset in an active transaction.

Declaration: `function PSInTransaction : Boolean`

Visibility: default

Description: `PSInTransaction` returns `True` if the dataset is in an active transaction or `False` if no transaction is active.

See also: `PSEndTransaction` (316), `PSStartTransaction` (316)

### 11.8.17 IProviderSupport.PSIsSQLBased

Synopsis: Is the dataset SQL based

Declaration: `function PSIsSQLBased : Boolean`

Visibility: default

Description: `PSIsSQLBased` returns `True` if the dataset is SQL based or not. Note that this is different from `PSIsSQLSupported` (316) which indicates whether SQL statements can be executed using `PSExecuteCommand` (316)

See also: `PSIsSQLSupported` (316), `PSExecuteCommand` (316)

### 11.8.18 IProviderSupport.PSIsSQLSupported

Synopsis: Can the dataset support SQL statements

Declaration: `function PSIsSQLSupported : Boolean`

Visibility: default

Description: `PSIsSQLSupported` returns `True` if `PSExecuteCommand` (316) can be used to execute SQL statements on the underlying database.

See also: `PSExecuteCommand` (316)

### 11.8.19 IProviderSupport.PSReset

Synopsis: Position the dataset on the first record

Declaration: `procedure PSReset`

Visibility: default

Description: `PSReset` repositions the dataset on the first record. For bi-directional datasets, this usually means that first is called, but for unidirectional datasets this may result in re-fetching the data from the underlying database.

See also: `TDataset.First` (384), `TDataset.Open` (389)

### 11.8.20 IProviderSupport.PSSetCommandText

Synopsis: Set the command-text of the dataset

Declaration: `procedure PSSetCommandText (const CommandText: string)`

Visibility: default

Description: `PSSetCommandText` sets the `commandtext` (SQL) statement that is executed by `PSExecute` or that is used to open the dataset.

See also: `PSExecute` (316), `PSGetCommandText` (316), `PSSetParams` (316)

### 11.8.21 IProviderSupport.PSSetParams

Synopsis: Set the parameters for the command text

Declaration: `procedure PSSetParams (AParams: TParams)`

Visibility: default

Description: `PSSetParams` sets the values of the parameters that should be used when executing the command-text SQL statement.

See also: `PSSetCommandText` (316), `PSGetParams` (316)

### 11.8.22 IProviderSupport.PSStartTransaction

Synopsis: Start a new transaction

Declaration: `procedure PSStartTransaction`

Visibility: default

Description: `PSStartTransaction` is used by the provider to start a new transaction. It will only be called if no transaction was active yet (i.e. `PSIntransaction` (316) returned `False`).

See also: `PSEndTransaction` (316), `PSIntransaction` (316)

### 11.8.23 IProviderSupport.PSUpdateRecord

Synopsis: Update a record

Declaration: `function PSUpdateRecord (UpdateKind: TUpdateKind; Delta: TDataSet)  
: Boolean`

Visibility: default

Description: `PSUpdateRecord` is called before attempting to update the records through generated SQL statements. The update to be performed is passed in `UpdateKind` parameter. The `Delta` Dataset's current record contains all data for the record that must be updated.

The function returns `True` if the update was successfully applied, `False` if not. In that case the provider will attempt to update the record using SQL statements if the dataset allows it.

See also: `PSIsSQLSupported` (316), `PSExecuteCommand` (316)

## 11.9 TAutoIncField

### 11.9.1 Description

`TAutoIncField` is the class created when a dataset must manage 32-bit signed integer data, of datatype `ftAutoInc`: This field gets its data automatically by the database engine. It exposes no new properties, but simply overrides some methods to manage 32-bit signed integer data.

It should never be necessary to create an instance of `TAutoIncField` manually, a field of this class will be instantiated automatically for each auto-incremental field when a dataset is opened.

See also: `TField` (421)



### 11.9.2 Method overview

Page	Method	Description
<a href="#">344</a>	Create	Create a new instance of the TAutoIncField class.

### 11.9.3 TAutoIncField.Create

**Synopsis:** Create a new instance of the TAutoIncField class.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** public

**Description:** Create initializes a new instance of the TAutoIncField class. It simply calls the inherited constructor and then sets up some of the TField ([421](#)) class' fields.

See also: TField ([421](#))

## 11.10 TBCDField

### 11.10.1 Description

TBCDField is the class used when a dataset must manage data of Binary Coded Decimal type. (TField.DataType ([435](#)) equals ftBCD). It initializes some of the properties of the TField ([421](#)) class, and overrides some of its methods to be able to work with BCD fields.

TBCDField assumes that the field's contents can be stored in a currency type, i.e. the maximum number of decimals after the decimal separator that can be stored in a TBCDField is 4. Fields that need to store a larger amount of decimals should be represented by a TFMTBCDField ([463](#)) instance.

It should never be necessary to create an instance of TBCDField manually, a field of this class will be instantiated automatically for each BCD field when a dataset is opened.

See also: TDataSet ([371](#)), TField ([421](#)), TFMTBCDField ([463](#))

### 11.10.2 Method overview

Page	Method	Description
<a href="#">345</a>	CheckRange	Check whether a values falls within the allowed range
<a href="#">344</a>	Create	Create a new instance of a TBCDField class.

### 11.10.3 Property overview

Page	Properties	Access	Description
<a href="#">346</a>	Currency	rw	Does the field represent a currency amount
<a href="#">346</a>	MaxValue	rw	Maximum value for the field
<a href="#">346</a>	MinValue	rw	Minimum value for the field
<a href="#">345</a>	Precision	rw	Precision of the BCD field
<a href="#">347</a>	Size		Number of decimals after the decimal separator
<a href="#">345</a>	Value	rw	Value of the field contents as a Currency type

### 11.10.4 TBCDField.Create

**Synopsis:** Create a new instance of a TBCDField class.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** `public`

**Description:** `Create` initializes a new instance of the `TBCDField` class. It calls the inherited destructor, and then sets some `TField` (421) properties to configure the instance for working with BCD data values.

See also: `TField` (421)

### 11.10.5 TBCDField.CheckRange

**Synopsis:** Check whether a values falls within the allowed range

**Declaration:** `function CheckRange(AValue: Currency) : Boolean`

**Visibility:** `public`

**Description:** `CheckRange` returns `True` if `AValue` lies within the range defined by the `MinValue` (346) and `MaxValue` (346) properties. If the value lies outside of the allowed range, then `False` is returned.

See also: `MaxValue` (346), `MinValue` (346)

### 11.10.6 TBCDField.Value

**Synopsis:** Value of the field contents as a `Currency` type

**Declaration:** `Property Value : Currency`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `Value` is overridden from the `TField.Value` (439) property to a `currency` type field. It returns the same value as the `TField.AsCurrency` (429) field.

See also: `TField.Value` (439), `TField.AsCurrency` (429)

### 11.10.7 TBCDField.Precision

**Synopsis:** Precision of the BCD field

**Declaration:** `Property Precision : LongInt`

**Visibility:** `published`

**Access:** `Read,Write`

**Description:** `Precision` is the total number of decimals in the BCD value. It is not the same as `TBCDField.Size` (347), which is the number of decimals after the decimal point. The `Precision` property should be set by the descendent classes when they initialize the field, and should be considered read-only. Changing the value will influence the values returned by the various `AsXXX` properties.

See also: `TBCDField.Size` (347), `TBCDField.Value` (345)

### 11.10.8 TBCDField.Currency

Synopsis: Does the field represent a currency amount

Declaration: `Property Currency : Boolean`

Visibility: published

Access: Read,Write

Description: `Currency` can be set to `True` to indicate that the field contains data representing an amount of currency. This affects the way the `TField.DisplayText` (435) and `TField.Text` (438) properties format the value of the field: if the `Currency` property is `True`, then these properties will format the value as a currency value (generally appending the currency sign) and if the `Currency` property is `False`, then they will format it as a normal floating-point value.

See also: `TField.DisplayText` (435), `TField.Text` (438)

### 11.10.9 TBCDField.MaxValue

Synopsis: Maximum value for the field

Declaration: `Property MaxValue : Currency`

Visibility: published

Access: Read,Write

Description: `MaxValue` can be set to a value different from zero, it is then the maximum value for the field if set to any value different from zero. When setting the field's value, the value may not be larger than `MaxValue`. Any attempt to write a larger value as the field's content will result in an exception. By default `MaxValue` equals 0, i.e. any floating-point value is allowed.

If `MaxValue` is set, `MinValue` (346) should also be set, because it will also be checked.

See also: `TBCDField.MinValue` (346), `TBCDField.CheckRange` (345)

### 11.10.10 TBCDField.MinValue

Synopsis: Minimum value for the field

Declaration: `Property MinValue : Currency`

Visibility: published

Access: Read,Write

Description: `MinValue` can be set to a value different from zero, then it is the minimum value for the field. When setting the field's value, the value may not be less than `MinValue`. Any attempt to write a smaller value as the field's content will result in an exception. By default `MinValue` equals 0, i.e. any floating-point value is allowed.

If `MinValue` is set, `TBCDField.MaxValue` (346) should also be set, because it will also be checked.

See also: `TBCDField.MaxValue` (346), `TBCDField.CheckRange` (345)

### 11.10.11 TBCDField.Size

Synopsis: Number of decimals after the decimal separator

Declaration: `Property Size :`

Visibility: `published`

Access:

Description: `Size` is the number of decimals after the decimal separator. It is not the total number of decimals, which is stored in the `TBCDField.Precision` (345) field.

See also: `TBCDField.Precision` (345)

## 11.11 TBinaryField

### 11.11.1 Description

`TBinaryField` is an abstract class, designed to handle binary data of variable size. It overrides some of the properties and methods of the `TField` (421) class to be able to work with binary field data, such as retrieving the contents as a string or as a variant.

One must never create an instance of `TBinaryField` manually, it is an abstract class. Instead, a descendent class such as `TBytesField` (353) or `TVarBytesField` (508) should be created.

See also: `TDataset` (371), `TField` (421), `TBytesField` (353), `TVarBytesField` (508)

### 11.11.2 Method overview

Page	Method	Description
<a href="#">347</a>	<code>Create</code>	Create a new instance of a <code>TBinaryField</code> class.

### 11.11.3 Property overview

Page	Properties	Access	Description
<a href="#">348</a>	<code>Size</code>		Size of the binary data

### 11.11.4 TBinaryField.Create

Synopsis: Create a new instance of a `TBinaryField` class.

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initializes a new instance of the `TBinaryField` class. It simply calls the inherited destructor.

See also: `TField` (421)

### 11.11.5 TBinaryField.Size

Synopsis: Size of the binary data

Declaration: `Property Size :`

Visibility: published

Access:

Description: `Size` is simply redeclared published with a default value of 16.

See also: `TField.Size` ([438](#))

## 11.12 TBlobField

### 11.12.1 Description

`TBlobField` is the class used when a dataset must manage BLOB data. (`TField.DataType` ([435](#)) equals `ftBLOB`). It initializes some of the properties of the `TField` ([421](#)) class, and overrides some of its methods to be able to work with BLOB fields. It also serves as parent class for some specialized blob-like field types such as `TMemoField` ([481](#)), `TWideMemoField` ([509](#)) or `TGraphicField` ([465](#))

It should never be necessary to create an instance of `TBlobField` manually, a field of this class will be instantiated automatically for each BLOB field when a dataset is opened.

See also: `TDataset` ([371](#)), `TField` ([421](#)), `TMemoField` ([481](#)), `TWideMemoField` ([509](#)), `TGraphicField` ([465](#))

### 11.12.2 Method overview

Page	Method	Description
<a href="#">349</a>	<code>Clear</code>	Clear the BLOB field's contents
<a href="#">348</a>	<code>Create</code>	Create a new instance of a <code>TBlobField</code> class.
<a href="#">349</a>	<code>IsBlob</code>	Is the field a blob field
<a href="#">349</a>	<code>LoadFromFile</code>	Load the contents of the field from a file
<a href="#">349</a>	<code>LoadFromStream</code>	Load the field's contents from stream
<a href="#">350</a>	<code>SaveToFile</code>	Save field contents to a file
<a href="#">350</a>	<code>SaveToStream</code>	Save the field's contents to stream
<a href="#">350</a>	<code>SetFieldType</code>	Set field type

### 11.12.3 Property overview

Page	Properties	Access	Description
<a href="#">350</a>	<code>BlobSize</code>	r	Size of the current blob
<a href="#">351</a>	<code>BlobType</code>	rw	Type of blob
<a href="#">351</a>	<code>Modified</code>	rw	Has the field's contents been modified.
<a href="#">352</a>	<code>Size</code>		Size of the blob field
<a href="#">351</a>	<code>Transliterate</code>	rw	Should the contents of the field be transliterated
<a href="#">351</a>	<code>Value</code>	rw	Return the field's contents as a string

### 11.12.4 TBlobField.Create

Synopsis: Create a new instance of a `TBlobField` class.

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` initializes a new instance of the `TBlobField` class. It calls the inherited destructor, and then sets some `TField` (421) properties to configure the instance for working with BLOB data.

See also: `TField` (421)

### 11.12.5 `TBlobField.Clear`

Synopsis: Clear the BLOB field's contents

Declaration: `procedure Clear; Override`

Visibility: public

Description: `Clear` overrides the `TField` implementation of `TField.Clear` (426). It creates and immediately releases an empty blob stream in write mode, effectively clearing the contents of the BLOB field.

See also: `TField.Clear` (426), `TField.IsNull` (437)

### 11.12.6 `TBlobField.IsBlob`

Synopsis: Is the field a blob field

Declaration: `class function IsBlob : Boolean; Override`

Visibility: public

Description: `IsBlob` is overridden by `TBlobField` to return `True`

See also: `TField.IsBlob` (427)

### 11.12.7 `TBlobField.LoadFromFile`

Synopsis: Load the contents of the field from a file

Declaration: `procedure LoadFromFile(const FileName: string)`

Visibility: public

Description: `LoadFromFile` creates a file stream with `FileName` as the name of the file to open, en then calls `LoadFromStream` (349) to read the contents of the blob field from the file. The file is opened in read-only mode.

Errors: If the file does not exist or is nor available for reading, an exception will be raised.

See also: `LoadFromStream` (349), `SaveToFile` (350)

### 11.12.8 `TBlobField.LoadFromStream`

Synopsis: Load the field's contents from stream

Declaration: `procedure LoadFromStream(Stream: TStream)`

Visibility: public

Description: `LoadFromStream` can be used to load the contents of the field from a `TStream` (??) descendent. The entire data of the stream will be copied, and the stream will be positioned on the first byte of data, so it must be seekable.

Errors: If the stream is not seekable, an exception will be raised.

See also: [SaveToStream \(350\)](#), [LoadFromFile \(349\)](#)

### 11.12.9 TBlobField.SaveToFile

Synopsis: Save field contents to a file

Declaration: `procedure SaveToFile(const FileName: string)`

Visibility: public

Description: `SaveToFile` creates a file stream with `FileName` as the name of the file to open, en then calls [SaveToStream \(350\)](#) to write the contents of the blob field to the file. The file is opened in write mode and is created if it does not yet exist.

Errors: If the file cannot be created or is not available for writing, an exception will be raised.

See also: [LoadFromFile \(349\)](#), [SaveToStream \(350\)](#)

### 11.12.10 TBlobField.SaveToStream

Synopsis: Save the field's contents to stream

Declaration: `procedure SaveToStream(Stream: TStream)`

Visibility: public

Description: `SaveToStream` can be used to save the contents of the field to a `TStream` (??) descendent. The entire data of the field will be copied. The stream must of course support writing.

Errors: If the stream is not writable, an exception will be raised.

See also: [SaveToFile \(350\)](#), [LoadFromStream \(349\)](#)

### 11.12.11 TBlobField.SetFieldType

Synopsis: Set field type

Declaration: `procedure SetFieldType(AValue: TFieldType); Override`

Visibility: public

Description: `SetFieldType` is overridden by `TBlobField` to check whether a valid Blob field type is set. If so, it calls the inherited method.

See also: `TField.DataType` ([435](#))

### 11.12.12 TBlobField.BlobSize

Synopsis: Size of the current blob

Declaration: `Property BlobSize : LongInt`

Visibility: public

Access: Read

Description: `BlobSize` is the size (in bytes) of the current contents of the field. It will vary as the dataset's current record moves from record to record.

See also: `TField.Size` ([438](#)), `TField.DataSize` ([435](#))

### 11.12.13 TBlobField.Modified

Synopsis: Has the field's contents been modified.

Declaration: `Property Modified : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `Modified` indicates whether the field's contents have been modified for the current record.

See also: `TBlobField.LoadFromStream` ([349](#))

### 11.12.14 TBlobField.Value

Synopsis: Return the field's contents as a string

Declaration: `Property Value : string`

Visibility: `public`

Access: `Read,Write`

Description: `Value` is redefined by `TBlobField` as a string value: getting or setting this value will convert the BLOB data to a string, it will return the same value as the `TField.AsString` ([431](#)) property.

See also: `TField.Value` ([439](#)), `TField.AsString` ([431](#))

### 11.12.15 TBlobField.Transliterate

Synopsis: Should the contents of the field be transliterated

Declaration: `Property Transliterate : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `Transliterate` indicates whether the contents of the field should be transliterated (i.e. changed from OEM to non OEM codepage and vice versa) when reading or writing the value. The actual transliteration must be done in the `TDataset.Translate` ([391](#)) method of the dataset to which the field belongs. By default this property is `False`, but it can be set to `True` for BLOB data which contains text in another codepage.

See also: `TStringField.Transliterate` ([506](#)), `TDataset.Translate` ([391](#))

### 11.12.16 TBlobField.BlobType

Synopsis: Type of blob

Declaration: `Property BlobType : TBlobType`

Visibility: `published`

Access: `Read,Write`

Description: `BlobType` is an alias for `TField.DataType` ([435](#)), but with a restricted set of values. Setting `BlobType` is equivalent to setting the `TField.DataType` ([435](#)) property.

See also: `TField.DataType` ([435](#))



### 11.12.17 TBlobField.Size

Synopsis: Size of the blob field

Declaration: `Property Size :`

Visibility: `published`

Access:

Description: `Size` is the size of the blob in the internal memory buffer. It defaults to 0, as the BLOB data is not stored in the internal memory buffer. To get the size of the data in the current record, use the `BlobSize` (350) property instead.

See also: `BlobSize` (350)

## 11.13 TBooleanField

### 11.13.1 Description

`TBooleanField` is the field class used by `TDataset` (371) whenever it needs to manage boolean data (`TField.DataType` (435) equals `ftBoolean`). It overrides some properties and methods of `TField` (421) to be able to work with boolean data.

It should never be necessary to create an instance of `TBooleanField` manually, a field of this class will be instantiated automatically for each boolean field when a dataset is opened.

See also: `TDataset` (371), `TField` (421)

### 11.13.2 Method overview

Page	Method	Description
<a href="#">352</a>	<code>Create</code>	Create a new instance of the <code>TBooleanField</code> class.

### 11.13.3 Property overview

Page	Properties	Access	Description
<a href="#">353</a>	<code>DisplayValues</code>	<code>rw</code>	Textual representation of the true and false values
<a href="#">353</a>	<code>Value</code>	<code>rw</code>	Value of the field as a boolean value

### 11.13.4 TBooleanField.Create

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

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initializes a new instance of the `TBooleanField` class. It calls the inherited constructor and then sets some `TField` (421) properties to configure it for working with boolean values.

See also: `TField` (421)

### 11.13.5 TBooleanField.Value

Synopsis: Value of the field as a boolean value

Declaration: `Property Value : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `Value` is redefined from `TField.Value` (439) by `TBooleanField` as a boolean value. It returns the same value as the `TField.AsBoolean` (429) property.

See also: `TField.AsBoolean` (429), `TField.Value` (439)

### 11.13.6 TBooleanField.DisplayValues

Synopsis: Textual representation of the true and false values

Declaration: `Property DisplayValues : string`

Visibility: `published`

Access: `Read,Write`

Description: `DisplayValues` contains 2 strings, separated by a semicolon (;) which are used to display the `True` and `False` values of the fields. The first string is used for `True` values, the second value is used for `False` values. If only one value is given, it will serve as the representation of the `True` value, the `False` value will be represented as an empty string.

A value of `Yes;No` will result in `True` values being displayed as 'Yes', and `False` values as 'No'. When writing the value of the field as a string, the string will be compared (case insensitively) with the value for `True`, and if it matches, the field's value will be set to `True`. After this it will be compared to the value for `False`, and if it matches, the field's value will be set to `False`. If the text matches neither of the two values, an exception will be raised.

See also: `TField.AsString` (431), `TField.Text` (438)

## 11.14 TBytesField

### 11.14.1 Description

`TBytesField` is the class used when a dataset must manage data of fixed-size binary type. (`TField.DataType` (435) equals `ftBytes`). It initializes some of the properties of the `TField` (421) class to be able to work with fixed-size byte fields.

It should never be necessary to create an instance of `TBytesField` manually, a field of this class will be instantiated automatically for each binary data field when a dataset is opened.

See also: `TDataset` (371), `TField` (421), `TVarBytesField` (508)

### 11.14.2 Method overview

Page	Method	Description
<a href="#">354</a>	Create	Create a new instance of a <code>TBytesField</code> class.

### 11.14.3 TBytesField.Create

**Synopsis:** Create a new instance of a `TBytesField` class.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** public

**Description:** `Create` initializes a new instance of the `TBytesField` class. It calls the inherited destructor, and then sets some `TField` (421) properties to configure the instance for working with binary data values.

See also: `TField` (421)

## 11.15 TCheckConstraint

### 11.15.1 Description

`TCheckConstraint` can be used to store the definition of a record-level constraint. It does not enforce the constraint, it only stores the constraint's definition. The constraint can come from several sources: an imported constraints from the database, usually stored in the `TCheckConstraint.ImportedConstraint` (355) property, or a constraint enforced by the user on a particular dataset instance stored in `TCheckConstraint.CustomConstraint` (355)

See also: `TCheckConstraints` (356), `TCheckConstraint.ImportedConstraint` (355), `TCheckConstraint.CustomConstraint` (355)

### 11.15.2 Method overview

Page	Method	Description
354	<code>Assign</code>	Assign one constraint to another

### 11.15.3 Property overview

Page	Properties	Access	Description
355	<code>CustomConstraint</code>	rw	User-defined constraint
355	<code>ErrorMessage</code>	rw	Message to display when the constraint is violated
355	<code>FromDictionary</code>	rw	True if the constraint is imported from a datadictionary
355	<code>ImportedConstraint</code>	rw	Constraint imported from the database engine

### 11.15.4 TCheckConstraint.Assign

**Synopsis:** Assign one constraint to another

**Declaration:** `procedure Assign(Source: TPersistent); Override`

**Visibility:** public

**Description:** `Assign` is overridden by `TCheckConstraint` to copy all published properties if `Source` is also a `TCheckConstraint` instance.

**Errors:** If `Source` is not an instance of `TCheckConstraint`, an exception may be thrown.

See also: `TCheckConstraint.ImportedConstraint` (355), `TCheckConstraint.CustomConstraint` (355)

### 11.15.5 TCheckConstraint.CustomConstraint

Synopsis: User-defined constraint

Declaration: `Property CustomConstraint : string`

Visibility: published

Access: Read,Write

Description: `CustomConstraint` is an SQL expression with an additional user-defined constraint. The expression should be enforced by a `TDataset` (371) descendent when data is posted to the dataset. If the constraint is violated, then the dataset should raise an exception, with message as specified in `TCheckConstraint.ErrorMessage` (355)

See also: `TCheckConstraint.ErrorMessage` (355)

### 11.15.6 TCheckConstraint.ErrorMessage

Synopsis: Message to display when the constraint is violated

Declaration: `Property ErrorMessage : string`

Visibility: published

Access: Read,Write

Description: `ErrorMessage` is used as the message when the dataset instance raises an exception if the constraint is violated.

See also: `TCheckConstraint.CustomConstraint` (355)

### 11.15.7 TCheckConstraint.FromDictionary

Synopsis: True if the constraint is imported from a datadictionary

Declaration: `Property FromDictionary : Boolean`

Visibility: published

Access: Read,Write

Description: `FromDictionary` indicates whether a constraint is imported from a data dictionary. This can be set by `TDataset` (371) descendents to indicate the source of the constraint, but is otherwise ignored.

See also: `TCheckConstraint.ImportedConstraint` (355)

### 11.15.8 TCheckConstraint.ImportedConstraint

Synopsis: Constraint imported from the database engine

Declaration: `Property ImportedConstraint : string`

Visibility: published

Access: Read,Write

Description: `ImportedConstraint` is a constraint imported from the database engine: it will not be enforced locally by the `TDataset` (371) descendent.

See also: `TCheckConstraint.CustomConstraint` (355)

## 11.16 TCheckConstraints

### 11.16.1 Description

`TCheckConstraints` is a `TCollection` descendent which keeps a collection of `TCheckConstraint` (354) items. It overrides the `Add` (356) method to return a `TCheckConstraint` instance.

See also: `TCheckConstraint` (354)

### 11.16.2 Method overview

Page	Method	Description
<a href="#">356</a>	<code>Add</code>	Add new <code>TCheckConstraint</code> item to the collection
<a href="#">356</a>	<code>Create</code>	Create a new instance of the <code>TCheckConstraints</code> class.

### 11.16.3 Property overview

Page	Properties	Access	Description
<a href="#">356</a>	<code>Items</code>	<code>rw</code>	Indexed access to the items in the collection

### 11.16.4 TCheckConstraints.Create

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

Declaration: `constructor Create(AOwner: TPersistent)`

Visibility: `public`

Description: `Create` initializes a new instance of the `TCheckConstraints` class. The `AOwner` argument is usually the `TDataset` (371) instance for which the data is managed. It is kept for future reference. After storing the owner, the inherited constructor is called with the `TCheckConstraint` (354) class pointer.

See also: `TCheckConstraint` (354), `TDataset` (371)

### 11.16.5 TCheckConstraints.Add

Synopsis: Add new `TCheckConstraint` item to the collection

Declaration: `function Add : TCheckConstraint`

Visibility: `public`

Description: `Add` is overridden by `TCheckConstraint` to add a new `TCheckConstraint` (354) instance to the collection. it returns the newly added instance.

See also: `TCheckConstraint` (354), `#rtl.classes.TCollection.Add` (??)

### 11.16.6 TCheckConstraints.Items

Synopsis: Indexed access to the items in the collection

Declaration: `Property Items[Index: LongInt]: TCheckConstraint; default`

Visibility: `public`

Access: Read,Write

Description: `Items` is overridden by `TCheckConstraints` to provide type-safe access to the items in the collection. The `index` is zero-based, so it runs from 0 to `Count-1`.

See also: `#rtl.classes.TCollection.Items` (??)

## 11.17 TCurrencyField

### 11.17.1 Description

`TCurrencyField` is the field class used by `TDataset` (371) when it needs to manage currency-valued data. (`TField.Datatype` (435) equals `ftCurrency`). It simply sets some `Tfield` (421) properties to be able to work with currency data.

It should never be necessary to create an instance of `TCurrencyField` manually, a field of this class will be instantiated automatically for each currency field when a dataset is opened.

See also: `TField` (421), `TDataset` (371)

### 11.17.2 Method overview

Page	Method	Description
<a href="#">357</a>	<code>Create</code>	Create a new instance of a <code>TCurrencyField</code> .

### 11.17.3 Property overview

Page	Properties	Access	Description
<a href="#">357</a>	<code>Currency</code>		Is the field a currency field

### 11.17.4 TCurrencyField.Create

Synopsis: Create a new instance of a `TCurrencyField`.

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initializes a new instance of `TCurrencyField`. It calls the inherited constructor and then sets some properties (`TCurrencyField.Currency` (357)) to be able to work with currency data.

See also: `TField` (421), `TCurrencyField.Currency` (357)

### 11.17.5 TCurrencyField.Currency

Synopsis: Is the field a currency field

Declaration: `Property Currency :`

Visibility: `published`

Access:

Description: `Currency` is inherited from `TFloatField.Currency` (461) but is initialized to `True` by the `TCurrencyField` constructor. It can be set to `False` if the contents of the field is of type currency, but does not represent an amount of currency.

See also: `TFloatField.Currency` ([461](#))

## 11.18 TCustomConnection

### 11.18.1 Description

`TCustomConnection` must be used for all database classes that need a connection to a server. The class introduces some methods and classes to activate the connection (`Open` ([359](#))) and to deactivate the connection (`TCustomConnection.Close` ([358](#))), plus a property to inspect the state (`Connected` ([360](#))) of the connected.

See also: `TCustomConnection.Open` ([359](#)), `TCustomConnection.Close` ([358](#)), `TCustomConnection.Connected` ([360](#))

### 11.18.2 Method overview

Page	Method	Description
<a href="#">358</a>	<code>Close</code>	Close the connection
<a href="#">359</a>	<code>Destroy</code>	Remove the <code>TCustomconnection</code> instance from memory
<a href="#">359</a>	<code>Open</code>	Makes the connection to the server

### 11.18.3 Property overview

Page	Properties	Access	Description
<a href="#">360</a>	<code>AfterConnect</code>	rw	Event triggered after a connection is made.
<a href="#">361</a>	<code>AfterDisconnect</code>	rw	Event triggered after a connection is closed
<a href="#">361</a>	<code>BeforeConnect</code>	rw	Event triggered before a connection is made.
<a href="#">361</a>	<code>BeforeDisconnect</code>	rw	Event triggered before a connection is closed
<a href="#">360</a>	<code>Connected</code>	rw	Is the connection established or not
<a href="#">359</a>	<code>DataSetCount</code>	r	Number of datasets connected to this connection
<a href="#">360</a>	<code>DataSets</code>	r	Datasets linked to this connection
<a href="#">360</a>	<code>LoginPrompt</code>	rw	Should the <code>OnLogin</code> be triggered
<a href="#">361</a>	<code>OnLogin</code>	rw	Event triggered when a login prompt is shown.

### 11.18.4 TCustomConnection.Close

Synopsis: Close the connection

Declaration: `procedure Close(ForceClose: Boolean)`

Visibility: public

Description: `Close` closes the connection with the server if it was connected. Calling this method first triggers the `BeforeDisconnect` ([361](#)) event. If an exception is raised during the execution of that event handler, the disconnect process is aborted. After calling this event, the connection is actually closed. After the connection was closed, the `AfterDisconnect` ([361](#)) event is triggered.

Calling the `Close` method is equivalent to setting the `Connected` ([360](#)) property to `False`.

If `ForceClose` is `True` then the descendent should ignore errors from the underlying connection, allowing all datasets to be closed properly.

Errors: If the connection cannot be broken for some reason, an `EDatabaseError` ([335](#)) exception will be raised.

See also: `TCustomConnection.BeforeDisconnect` (361), `TCustomConnection.AfterDisconnect` (361), `TCustomConnection.Open` (359), `TCustomConnection.Connected` (360)

### 11.18.5 TCustomConnection.Destroy

Synopsis: Remove the `TCustomConnection` instance from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` closes the connection, and then calls the inherited destructor.

Errors: If an exception is raised during the disconnect process, an exception will be raised, and the instance is not removed from memory.

See also: `TCustomConnection.Close` (358)

### 11.18.6 TCustomConnection.Open

Synopsis: Makes the connection to the server

Declaration: `procedure Open`

Visibility: `public`

Description: `Open` establishes the connection with the server if it was not yet connected. Calling this method first triggers the `BeforeConnect` (361) event. If an exception is raised during the execution of that event handler, the connect process is aborted. If `LoginPrompt` (360) is `True`, the `OnLogin` (361) event handler is called. Only after this event, the connection is actually established. After the connection was established, the `AfterConnect` (360) event is triggered.

Calling the `Open` method is equivalent to setting the `Connected` (360) property to `True`.

Errors: If an exception is raised during the `BeforeConnect` or `OnLogin` handlers, the connection is not actually established.

See also: `TCustomConnection.BeforeConnect` (361), `TCustomConnection.LoginPrompt` (360), `TCustomConnection.OnLogin` (361), `TCustomConnection.AfterConnect` (360), `TCustomConnection.Connected` (360)

### 11.18.7 TCustomConnection.DataSetCount

Synopsis: Number of datasets connected to this connection

Declaration: `Property DataSetCount : LongInt`

Visibility: `public`

Access: `Read`

Description: `DataSetCount` is the number of datasets connected to this connection component. The actual datasets are available through the `Datasets` (360) array property. As implemented in `TCustomConnection`, this property is always zero. Descendent classes implement the actual count.

See also: `TDataset` (371), `TCustomConnection.Datasets` (360)



### 11.18.8 TCustomConnection.DataSets

Synopsis: Datasets linked to this connection

Declaration: `Property DataSets[Index: LongInt]: TDataSet`

Visibility: `public`

Access: `Read`

Description: `DataSets` allows indexed access to the datasets connected to this connection. `Index` is a zero-based indexed, it's maximum value is `DatasetCount-1` ([359](#)).

See also: `DatasetCount` ([359](#))

### 11.18.9 TCustomConnection.Connected

Synopsis: Is the connection established or not

Declaration: `Property Connected : Boolean`

Visibility: `published`

Access: `Read,Write`

Description: `Connected` is `True` if the connection to the server is established, `False` if it is disconnected. The property can be set to `True` to establish a connection (equivalent to calling `TCustomConnection.Open` ([359](#)), or to `False` to break it (equivalent to calling `TCustomConnection.Close` ([358](#))).

See also: `TCustomConnection.Open` ([359](#)), `TCustomConnection.Close` ([358](#))

### 11.18.10 TCustomConnection.LoginPrompt

Synopsis: Should the `OnLogin` be triggered

Declaration: `Property LoginPrompt : Boolean`

Visibility: `published`

Access: `Read,Write`

Description: `LoginPrompt` can be set to `True` if the `OnLogin` handler should be called when the `Open` method is called. If it is not `True`, then the event handler is not called.

See also: `TCustomConnection.OnLogin` ([361](#))

### 11.18.11 TCustomConnection.AfterConnect

Synopsis: Event triggered after a connection is made.

Declaration: `Property AfterConnect : TNotifyEvent`

Visibility: `published`

Access: `Read,Write`

Description: `AfterConnect` is called after a connection is successfully established in `TCustomConnection.Open` ([359](#)). It can be used to open datasets, or indicate a connection status change.

See also: `TCustomConnection.Open` ([359](#)), `TCustomConnection.BeforeConnect` ([361](#)), `TCustomConnection.OnLogin` ([361](#))

### 11.18.12 TCustomConnection.AfterDisconnect

Synopsis: Event triggered after a connection is closed

Declaration: Property AfterDisconnect : TNotifyEvent

Visibility: published

Access: Read,Write

Description: AfterDisconnect is called after a connection is successfully closed in TCustomConnection.Close (358). It can be used for instance to indicate a connection status change.

See also: TCustomConnection.Close (358), TCustomConnection.BeforeDisconnect (361)

### 11.18.13 TCustomConnection.BeforeConnect

Synopsis: Event triggered before a connection is made.

Declaration: Property BeforeConnect : TNotifyEvent

Visibility: published

Access: Read,Write

Description: BeforeConnect is called before a connection is attempted in TCustomConnection.Open (359). It can be used to set connection parameters, or to abort the establishing of the connection: if an exception is raised during this event, the connection attempt is aborted.

See also: TCustomConnection.Open (359), TCustomConnection.AfterConnect (360), TCustomConnection.OnLogin (361)

### 11.18.14 TCustomConnection.BeforeDisconnect

Synopsis: Event triggered before a connection is closed

Declaration: Property BeforeDisconnect : TNotifyEvent

Visibility: published

Access: Read,Write

Description: BeforeDisconnect is called before a connection is closed in TCustomConnection.Close (358). It can be used for instance to check for unsaved changes, to save those changes, or to abort the disconnect operation: if an exception is raised during the event handler, the disconnect operation is aborted entirely.

See also: TCustomConnection.Close (358), TCustomConnection.AfterDisconnect (361)

### 11.18.15 TCustomConnection.OnLogin

Synopsis: Event triggered when a login prompt is shown.

Declaration: Property OnLogin : TLoginEvent

Visibility: published

Access: Read,Write

**Description:** `OnLogin` is triggered when the connection needs a login prompt during the call: it is triggered when the `LoginPrompt` (360) property is `True`, after the `TCustomConnection.BeforeConnect` (361) event, but before the connection is actually established.

See also: `TCustomConnection.BeforeConnect` (361), `TCustomConnection.LoginPrompt` (360), `TCustomConnection.Open` (359)

## 11.19 TDatabase

### 11.19.1 Description

`TDatabase` is a component whose purpose is to provide a connection to an external database engine, not to provide the database itself. This class provides generic methods for attachment to databases and querying their contents; the details of the actual connection are handled by database-specific components (such as `SQLDb` for SQL-based databases, or `DBA` for `DBASE/FoxPro` style databases).

Like `TDataset` (371), `TDatabase` is an abstract class. It provides methods to keep track of datasets connected to the database, and to close these datasets when the connection to the database is closed. To this end, it introduces a `Connected` (365) boolean property, which indicates whether a connection to the database is established or not. The actual logic to establish a connection to a database must be implemented by descendent classes.

See also: `TDataset` (371), `TDatabase` (362)

### 11.19.2 Method overview

Page	Method	Description
363	<code>CloseDataSets</code>	Close all connected datasets
363	<code>CloseTransactions</code>	End all transactions
362	<code>Create</code>	Initialize a new <code>TDatabase</code> class instance.
363	<code>Destroy</code>	Remove a <code>TDatabase</code> instance from memory.
364	<code>EndTransaction</code>	End an active transaction.
363	<code>StartTransaction</code>	Start a new transaction.

### 11.19.3 Property overview

Page	Properties	Access	Description
365	<code>Connected</code>	rw	Is the database connected
365	<code>DatabaseName</code>	rw	Database name or path
364	<code>Directory</code>	rw	Directory for the database
365	<code>IsSQLBased</code>	r	Is the database SQL based.
366	<code>KeepConnection</code>	rw	Should the connection be kept active
366	<code>Params</code>	rw	Connection parameters
364	<code>TransactionCount</code>	r	Number of transaction components connected to this database.
364	<code>Transactions</code>	r	Indexed access to all transaction components connected to this database.

### 11.19.4 TDatabase.Create

**Synopsis:** Initialize a new `TDatabase` class instance.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** `public`

**Description:** `Create` initializes a new instance of the `TDatabase` class. It allocates some resources and then calls the inherited constructor.

See also: `TBDDataset` ([415](#)), `TDBTransaction` ([417](#)), `TDatabase.Destroy` ([363](#))

### 11.19.5 TDatabase.Destroy

**Synopsis:** Remove a `TDatabase` instance from memory.

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

**Visibility:** `public`

**Description:** `Destroy` starts by disconnecting the database (thus closing all datasets and ending all transactions), then notifies all connected datasets and transactions that it is about to be released. After this, it releases all resources used by the `TDatabase` instance

See also: `TDatabase.CloseDatasets` ([363](#))

### 11.19.6 TDatabase.CloseDataSets

**Synopsis:** Close all connected datasets

**Declaration:** `procedure CloseDataSets`

**Visibility:** `public`

**Description:** `CloseDatasets` closes all connected datasets. It is called automatically when the connection is closed.

See also: `TCustomConnection.Close` ([358](#)), `TDatabase.CloseTransactions` ([363](#))

### 11.19.7 TDatabase.CloseTransactions

**Synopsis:** End all transactions

**Declaration:** `procedure CloseTransactions`

**Visibility:** `public`

**Description:** `CloseTransaction` calls `TDBTransaction.EndTransaction` ([417](#)) on all connected transactions. It is called automatically when the connection is closed, after all datasets are closed.

See also: `TCustomConnection.Close` ([358](#)), `TDatabase.CloseDatasets` ([363](#))

### 11.19.8 TDatabase.StartTransaction

**Synopsis:** Start a new transaction.

**Declaration:** `procedure StartTransaction; Virtual; Abstract`

**Visibility:** `public`

**Description:** `StartTransaction` must be implemented by descendent classes to start a new transaction. This method is provided for Delphi compatibility: new applications should use a `TDBTransaction` (417) component instead and invoke the `TDBTransaction.StartTransaction` (417) method.

See also: `TDBTransaction` (417), `TDBTransaction.StartTransaction` (417)

### 11.19.9 TDatabase.EndTransaction

**Synopsis:** End an active transaction.

**Declaration:** `procedure EndTransaction; Virtual; Abstract`

**Visibility:** `public`

**Description:** `EndTransaction` must be implemented by descendent classes to end an active transaction. This method is provided for Delphi compatibility: new applications should use a `TDBTransaction` (417) component instead and invoke the `TDBTransaction.EndTransaction` (417) method.

See also: `TDBTransaction` (417), `TDBTransaction.EndTransaction` (417)

### 11.19.10 TDatabase.TransactionCount

**Synopsis:** Number of transaction components connected to this database.

**Declaration:** `Property TransactionCount : LongInt`

**Visibility:** `public`

**Access:** `Read`

**Description:** `TransactionCount` is the number of transaction components which are connected to this database instance. It is the upper bound for the `TDatabase.Transactions` (364) array property.

See also: `TDatabase.Transactions` (364)

### 11.19.11 TDatabase.Transactions

**Synopsis:** Indexed access to all transaction components connected to this database.

**Declaration:** `Property Transactions[Index: LongInt]: TDBTransaction`

**Visibility:** `public`

**Access:** `Read`

**Description:** `Transactions` provides indexed access to the transaction components connected to this database. The `Index` is zero based: it runs from 0 to `TransactionCount-1`.

See also: `TDatabase.TransactionCount` (364)

### 11.19.12 TDatabase.Directory

**Synopsis:** Directory for the database

**Declaration:** `Property Directory : string`

**Visibility:** `public`

Access: Read,Write

Description: `Directory` is provided for Delphi compatibility: it indicates (for Paradox and dBase based databases) the directory where the database files are located. It is not used in the Free Pascal implementation of `TDatabase` (362).

See also: `TDatabase.Params` (366), `TDatabase.IsSQLBased` (365)

### 11.19.13 `TDatabase.IsSQLBased`

Synopsis: Is the database SQL based.

Declaration: `Property IsSQLBased : Boolean`

Visibility: `public`

Access: Read

Description: `IsSQLBased` is a read-only property which indicates whether a property is SQL-Based, i.e. whether the database engine accepts SQL commands.

See also: `TDatabase.Params` (366), `TDatabase.Directory` (364)

### 11.19.14 `TDatabase.Connected`

Synopsis: Is the database connected

Declaration: `Property Connected : Boolean`

Visibility: `published`

Access: Read,Write

Description: `Connected` is simply promoted to published property from `TCustomConnection.Connected` (360).

See also: `TCustomConnection.Connected` (360)

### 11.19.15 `TDatabase.DatabaseName`

Synopsis: Database name or path

Declaration: `Property DatabaseName : string`

Visibility: `published`

Access: Read,Write

Description: `DatabaseName` specifies the path of the database. For directory-based databases this will be the same as the `Directory` (364) property. For other databases this will be the name of a known pre-configured connection, or the location of the database file.

See also: `TDatabase.Directory` (364), `TDatabase.Params` (366)

### 11.19.16 TDatabase.KeepConnection

Synopsis: Should the connection be kept active

Declaration: `Property KeepConnection : Boolean`

Visibility: published

Access: Read,Write

Description: `KeepConnection` is provided for Delphi compatibility, and is not used in the Free Pascal implementation of `TDatabase`.

See also: `TDatabase.Params` ([366](#))

### 11.19.17 TDatabase.Params

Synopsis: Connection parameters

Declaration: `Property Params : TStrings`

Visibility: published

Access: Read,Write

Description: `Params` is a catch-all storage mechanism for database connection parameters. It is a list of strings in the form of `Name=Value` pairs. Which name/value pairs are supported depends on the `TDatabase` descendent, but the `user_name` and `password` parameters are commonly used to store the login credentials for the database.

See also: `TDatabase.Directory` ([364](#)), `TDatabase.DatabaseName` ([365](#))

## 11.20 TDataLink

### 11.20.1 Description

`TDataLink` is used by GUI controls or datasets in a master-detail relationship to handle data events coming from a `TDataSource` ([409](#)) instance. It is a class that exists for component programmers, application coders should never need to use `TDataLink` or one of its descendents.

DB-Aware Component coders must use a `TDataLink` instance to handle all communication with a `TDataSet` ([371](#)) instance, rather than communicating directly with the dataset. `TDataLink` contains methods which are called by the various events triggered by the dataset. Inversely, it has some methods to trigger actions in the dataset.

`TDataLink` is an abstract class; it is never used directly. Instead, a descendent class is used which overrides the various methods that are called in response to the events triggered by the dataset. Examples are .

See also: `TDataSet` ([371](#)), `TDataSource` ([409](#)), `TDetailDataLink` ([421](#)), `TMasterDataLink` ([477](#))

### 11.20.2 Method overview

Page	Method	Description
<a href="#">367</a>	Create	Initialize a new instance of TDataLink
<a href="#">367</a>	Destroy	Remove an instance of TDataLink from memory
<a href="#">367</a>	Edit	Set the dataset in edit mode, if possible
<a href="#">368</a>	ExecuteAction	Execute action
<a href="#">368</a>	UpdateAction	Update handler for actions
<a href="#">368</a>	UpdateRecord	Called when the data in the dataset must be updated

### 11.20.3 Property overview

Page	Properties	Access	Description
<a href="#">369</a>	Active	r	Is the link active
<a href="#">369</a>	ActiveRecord	rw	Currently active record
<a href="#">369</a>	BOF	r	Is the dataset at the first record
<a href="#">369</a>	BufferCount	rw	Set to the number of record buffers this datalink needs.
<a href="#">370</a>	DataSet	r	Dataset this datalink is connected to
<a href="#">370</a>	DataSource	rw	Datasource this datalink is connected to
<a href="#">370</a>	DataSourceFixed	rw	Can the datasource be changed
<a href="#">370</a>	Editing	r	Is the dataset in edit mode
<a href="#">371</a>	Eof	r	
<a href="#">371</a>	ReadOnly	rw	Is the link readonly
<a href="#">371</a>	RecordCount	r	Number of records in the buffer of the dataset

### 11.20.4 TDataLink.Create

Synopsis: Initialize a new instance of TDataLink

Declaration: `constructor Create`

Visibility: `public`

Description: `Create` calls the inherited constructor and then initializes some fields. In particular, it sets the `buffercount` to 1.

See also: `TDataLink.Destroy` ([367](#))

### 11.20.5 TDataLink.Destroy

Synopsis: Remove an instance of TDataLink from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` cleans up the TDataLink instance (in particular, it removes itself from the datasource it is coupled to), and then calls the inherited destructor.

See also: `TDataLink.Destroy` ([367](#))

### 11.20.6 TDataLink.Edit

Synopsis: Set the dataset in edit mode, if possible

Declaration: `function Edit : Boolean`



Visibility: public

**Description:** Edit attempts to put the dataset in edit mode. It returns `True` if this operation succeeded, `False` if not. To this end, it calls the `Edit` (410) method of the `DataSource` (370) to which the datalink instance is coupled. If the `TDatasource.AutoEdit` (411) property is `False` then this operation will not succeed, unless the dataset is already in edit mode. GUI controls should always respect the result of this function, and not allow the user to edit data if this function returned `false`.

See also: `TDatasource` (409), `TDatalink.DataSource` (370), `TDatasource.Edit` (410), `TDatasource.AutoEdit` (411)

### 11.20.7 TDataLink.UpdateRecord

**Synopsis:** Called when the data in the dataset must be updated

**Declaration:** `procedure UpdateRecord`

Visibility: public

**Description:** `Updaterecord` is called when the dataset expects the GUI controls to post any pending changes to the dataset. This method guards against recursive behaviour: while an `UpdateRecord` is in progress, the `TDatalink.RecordChange` (366) notification (which could result from writing data to the dataset) will be blocked.

See also: `TDatalink.RecordChange` (366)

### 11.20.8 TDataLink.ExecuteAction

**Synopsis:** Execute action

**Declaration:** `function ExecuteAction(Action: TBasicAction) : Boolean; Virtual`

Visibility: public

**Description:** `ExecuteAction` implements action support. It should never be necessary to call `ExecuteAction` from program code, as it is called automatically whenever a target control needs to handle an action. This method must be overridden in case any additional action must be taken when the action must be executed. The implementation in `TDatalink` checks if the action handles the datasource, and then calls `Action.ExecuteTarget`, passing it the datasource. If so, it returns `True`.

See also: `TDatalink.UpdateAction` (368)

### 11.20.9 TDataLink.UpdateAction

**Synopsis:** Update handler for actions

**Declaration:** `function UpdateAction(Action: TBasicAction) : Boolean; Virtual`

Visibility: public

**Description:** `UpdateAction` implements action update support. It should never be necessary to call `UpdateAction` from program code, as it is called automatically whenever a target control needs to update an action. This method must be overridden in case any specific action must be taken when the action must be updated. The implementation in `TDatalink` checks if the action handles the datasource, and then calls `Action.UpdateTarget`, passing it the datasource. If so, it returns `True`.

See also: `TDataLink.ExecuteAction` (368)

### 11.20.10 TDataLink.Active

Synopsis: Is the link active

Declaration: `Property Active : Boolean`

Visibility: `public`

Access: `Read`

Description: `Active` determines whether the events of the dataset are passed on to the control connected to the actionlink. If it is set to `False`, then no events are passed between control and dataset. It is set to `TDataset.Active` (400) whenever the `DataSource` (370) property is set.

See also: `TDatalink.DataSource` (370), `TDatalink.ReadOnly` (371), `TDataset.Active` (400)

### 11.20.11 TDataLink.ActiveRecord

Synopsis: Currently active record

Declaration: `Property ActiveRecord : Integer`

Visibility: `public`

Access: `Read,Write`

Description: `ActiveRecord` returns the index of the active record in the dataset's record buffer for this datalink.

See also: `TDatalink.BOF` (369), `TDatalink.EOF` (371)

### 11.20.12 TDataLink.BOF

Synopsis: Is the dataset at the first record

Declaration: `Property BOF : Boolean`

Visibility: `public`

Access: `Read`

Description: `BOF` returns `TDataset.BOF` (392) if the dataset is available, `True` otherwise.

See also: `TDatalink.EOF` (371), `TDataset.BOF` (392)

### 11.20.13 TDataLink.BufferCount

Synopsis: Set to the number of record buffers this datalink needs.

Declaration: `Property BufferCount : Integer`

Visibility: `public`

Access: `Read,Write`

Description: `BufferCount` can be set to the number of buffers that the dataset should manage on behalf of the control connected to this datalink. By default, this is 1. Controls that must display more than 1 buffer (such as grids) can set this to a higher value.

See also: `TDataset.ActiveBuffer` (376), `TDatalink.ActiveRecord` (369)

#### 11.20.14 TDataLink.DataSet

Synopsis: Dataset this datalink is connected to

Declaration: `Property DataSet : TDataSet`

Visibility: `public`

Access: `Read`

Description: `DataSet` equals `Datasource.Dataset` if the `datasource` is set, or `Nil` otherwise.

See also: `TDatalink.DataSource` (370), `TDataSet` (371)

#### 11.20.15 TDataLink.DataSource

Synopsis: Datasource this datalink is connected to

Declaration: `Property DataSource : TDataSource`

Visibility: `public`

Access: `Read,Write`

Description: `DataSource` should be set to a `TDataSource` (409) instance to get access to the dataset it is connected to. A datalink never points directly to a `TDataSet` (371) instance, always to a `datasource`. When the `datasource` is enabled or disabled, all `TDatalink` instances connected to it are enabled or disabled at once.

See also: `TDataSet` (371), `TDataSource` (409)

#### 11.20.16 TDataLink.DataSourceFixed

Synopsis: Can the datasource be changed

Declaration: `Property DataSourceFixed : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `DataSourceFixed` can be set to `True` to prevent changing of the `DataSource` (370) property. When lengthy operations are in progress, this can be done to prevent user code (e.g. event handlers) from changing the `datasource` property which might interfere with the operation in progress.

See also: `TDatalink.DataSource` (370)

#### 11.20.17 TDataLink.Editing

Synopsis: Is the dataset in edit mode

Declaration: `Property Editing : Boolean`

Visibility: `public`

Access: `Read`

Description: `Editing` determines whether the dataset is in one of the edit states (`dsEdit`, `dsInsert`). It can be set into this mode by calling the `TDatalink.Edit` (367) method. Never attempt to set the dataset in editing mode directly. The `Edit` method will perform the needed checks prior to setting the dataset in edit mode and will return `True` if the dataset was successfully set in the editing state.

See also: `TDatalink.Edit` (367), `TDataSet.Edit` (381)

### 11.20.18 TDataLink.Eof

Synopsis:

Declaration: `Property Eof : Boolean`

Visibility: `public`

Access: `Read`

Description: `EOF` returns `TDataset.EOF` (394) if the dataset is available, `True` otherwise.

See also: `TDatalink.BOF` (369), `TDataset.EOF` (394)

### 11.20.19 TDataLink.ReadOnly

Synopsis: Is the link readonly

Declaration: `Property ReadOnly : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `ReadOnly` can be set to `True` to indicate that the link is read-only, i.e. the connected control will not modify the dataset. Methods as `TDatalink.Edit` (367) will check this property and fail if the link is read-only. This setting has no effect on the communication of dataset events to the datalink: the `TDatalink.Active` (369) property can be used to disable delivery of events to the datalink.

See also: `TDatalink.Active` (369), `TDatalink.edit` (367)

### 11.20.20 TDataLink.RecordCount

Synopsis: Number of records in the buffer of the dataset

Declaration: `Property RecordCount : Integer`

Visibility: `public`

Access: `Read`

Description: `RecordCount` returns the number of records in the dataset's buffer. It is limited by the `TDatalink.BufferCount` (369) property: `RecordCount` is always less than `Buffercount`.

See also: `TDatalink.BufferCount` (369)

## 11.21 TDataSet

### 11.21.1 Description

`TDataset` is the main class of the `db` unit. This abstract class provides all basic functionality to access data stored in tabular format: The data consists of records, and the data in each record is organised in several fields.

`TDataset` has a buffer to cache a few records in memory, this buffer is used by `TDatasource` to create the ability to use data-aware components.

`TDataset` is an abstract class, which provides the basic functionality to access, navigate through the data and - in case read-write access is available, edit existing or add new records.

`TDataset` is an abstract class: it does not have the knowledge to store or load the records from whatever medium the records are stored on. Descendants add the functionality to load and save the data. Therefore `TDataset` is never used directly, one always instantiates a descendent class.

Initially, no data is available: the dataset is inactive. The `Open` (389) method must be used to fetch data into memory. After this command, the data is available in memory for browsing or editing purposes: The dataset is active (indicated by the `TDataset.Active` (400) property). Likewise, the `Close` (379) method can be used to remove the data from memory. Any changes not yet saved to the underlying medium will be lost.

Data is expected to be in tabular format, where each row represents a record. The dataset has an idea of a cursor: this is the current position of the data cursor in the set of rows. Only the data of the current record is available for display or editing purposes. Through the `Next` (388), `Prev` (371), `First` (384) and `Last` (387) methods, it is possible to navigate through the records. The `EOF` (394) property will be `True` if the last row has been reached. Likewise, the `BOF` (392) property will return `True` if the first record in the dataset has been reached when navigating backwards. If both properties are empty, then there is no data available. For dataset descendants that support counting the number of records, the `RecordCount` (397) will be zero.

The `Append` (377) and `Insert` (386) methods can be used to insert new records to the set of records. The `TDataset.Delete` (380) statement is used to delete the current record, and the `Edit` (381) command must be used to set the dataset in editing mode: the contents of the current record can then be changed. Any changes made to the current record (be it a new or existing record) must be saved by the `Post` (389) method, or can be undone using the `Cancel` (378) method.

The data in the various fields properties is available through the `Fields` (398) array property, giving indexed access to all the fields in a record. The contents of a field is always readable. If the dataset is in one of the editing modes, then the fields can also be written to.

See also: `TField` (421)



**11.21.2 Method overview**

Page	Method	Description
376	ActiveBuffer	Currently active memory buffer
377	Append	Append a new record to the data
377	AppendRecord	Append a new record to the dataset and fill with data
378	BookmarkValid	Test whether ABookMark is a valid bookmark.
378	Cancel	Cancel the current editing operation
378	CheckBrowseMode	Check whether the dataset is in browse mode.
378	ClearFields	Clear the values of all fields
379	Close	Close the dataset
379	CompareBookmarks	Compare two bookmarks
379	ControlsDisabled	Check whether the controls are disabled
376	Create	Create a new TDataset instance
380	CreateBlobStream	Create blob stream
380	CursorPosChanged	Indicate a change in cursor position
380	DataConvert	Convert data from/to native format
380	Delete	Delete the current record.
376	Destroy	Free a TDataset instance
381	DisableControls	Disable event propagation of controls
381	Edit	Set the dataset in editing mode.
382	EnableControls	Enable event propagation of controls
382	FieldByName	Search a field by name
382	FindField	Find a field by name
383	FindFirst	Find the first active record (deprecated)
383	FindLast	Find the last active record (deprecated)
383	FindNext	Find the next active record (deprecated)
383	FindPrior	Find the previous active record (deprecated)
384	First	Position the dataset on the first record.
384	FreeBookmark	Free a bookmark obtained with GetBookmark (deprecated)
384	GetBookmark	Get a bookmark pointer (deprecated)
385	GetCurrentRecord	Copy the data for the current record in a memory buffer
376	GetFieldData	Get the data for a field
385	GetFieldList	Return field instances in a list
385	GetFieldNames	Return a list of all available field names
385	GotoBookmark	Jump to bookmark
386	Insert	Insert a new record at the current position.
386	InsertRecord	Insert a new record with given values.
386	IsEmpty	Check if the dataset contains no data
386	IsLinkedTo	Check whether a datasource is linked to the dataset
387	IsSequenced	Is the data sequenced
387	Last	Navigate forward to the last record
387	Locate	Locate a record based on some key values
388	Lookup	Search for a record and return matching values.
388	MoveBy	Move the cursor position
388	Next	Go to the next record in the dataset.
389	Open	Activate the dataset: Fetch data into memory.
389	Post	Post pending edits to the database.
390	Prior	Go to the previous record
390	Refresh	Refresh the records in the dataset
390	Resync	Resynchronize the data buffer
377	SetFieldData	Store the data for a field
390	SetFields	Set a number of field values at once
391	Translate	Transliterate a buffer
391	UpdateCursorPos	Update cursor position
391	UpdateRecord	Indicate that the record contents have changed
392	UpdateStatus	Get the update status for the current record

### 11.21.3 Property overview

Page	Properties	Access	Description
400	Active	rw	Is the dataset open or closed.
404	AfterCancel	rw	Event triggered after a Cancel operation.
401	AfterClose	rw	Event triggered after the dataset is closed
404	AfterDelete	rw	Event triggered after a successful Delete operation.
402	AfterEdit	rw	Event triggered after the dataset is put in edit mode.
402	AfterInsert	rw	Event triggered after the dataset is put in insert mode.
401	AfterOpen	rw	Event triggered after the dataset is opened.
403	AfterPost	rw	Event called after changes have been posted to the underlying database
405	AfterRefresh	rw	Event triggered after the data has been refreshed.
405	AfterScroll	rw	Event triggered after the cursor has changed position.
400	AutoCalcFields	rw	How often should the value of calculated fields be calculated
403	BeforeCancel	rw	Event triggered before a Cancel operation.
401	BeforeClose	rw	Event triggered before the dataset is closed.
404	BeforeDelete	rw	Event triggered before a Delete operation.
402	BeforeEdit	rw	Event triggered before the dataset is put in edit mode.
401	BeforeInsert	rw	Event triggered before the dataset is put in insert mode.
400	BeforeOpen	rw	Event triggered before the dataset is opened.
403	BeforePost	rw	Event called before changes are posted to the underlying database
405	BeforeRefresh	rw	Event triggered before the data is refreshed.
404	BeforeScroll	rw	Event triggered before the cursor changes position.
392	BlockReadSize	rw	Number of records to read
392	BOF	r	Is the cursor at the beginning of the data (on the first record)
392	Bookmark	rw	Get or set the current cursor position
393	CanModify	r	Can the data in the dataset be modified
394	DataSource	r	Datasource this dataset is connected to.
394	DefaultFields	r	Is the dataset using persistent fields or not.
394	EOF	r	Indicates whether the last record has been reached.
395	FieldCount	r	Number of fields
395	FieldDefs	rw	Definitions of available fields in the underlying database
398	Fields	r	Indexed access to the fields of the dataset.
398	FieldValues	rw	Access to field values based on the field names.
399	Filter	rw	Filter to apply to the data in memory.
399	Filtered	rw	Is the filter active or not.
399	FilterOptions	rw	Options to apply when filtering
396	Found	r	Check success of one of the Find methods
396	IsUniDirectional	r	Is the dataset unidirectional (i.e. forward scrolling only)
396	Modified	r	Was the current record modified ?
406	OnCalcFields	rw	Event triggered when values for calculated fields must be computed.
406	OnDeleteError	rw	Event triggered when a delete operation fails.
406	OnEditError	rw	Event triggered when an edit operation fails.
407	OnFilterRecord	rw	Event triggered to filter records.
407	OnNewRecord	rw	Event triggered when a new record is created.
408	OnPostError	rw	Event triggered when a post operation fails.
397	RecNo	rw	Current record number
397	RecordCount	r	Number of records in the dataset
397	RecordSize	r	Size of the record in memory
398	State	r	Current operational state of the dataset



#### 11.21.4 TDataSet.Create

Synopsis: Create a new TDataSet instance

Declaration: constructor Create(AOwner: TComponent); Override

Visibility: public

Description: Create initializes a new TDataSet (371) instance. It calls the inherited constructor, and then initializes the internal structures needed to manage the dataset (fielddefs, fieldlist, constraints etc.).

See also: TDataSet.Destroy (376)

#### 11.21.5 TDataSet.Destroy

Synopsis: Free a TDataSet instance

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy removes a TDataSet instance from memory. It closes the dataset if it was open, clears all internal structures and then calls the inherited destructor.

Errors: An exception may occur during the close operation, in that case, the dataset will not be removed from memory.

See also: TDataSet.Close (379), TDataSet.Create (376)

#### 11.21.6 TDataSet.ActiveBuffer

Synopsis: Currently active memory buffer

Declaration: function ActiveBuffer : TRecordBuffer

Visibility: public

Description: ActiveBuffer points to the currently active memory buffer. It should not be used in application code.

#### 11.21.7 TDataSet.GetFieldData

Synopsis: Get the data for a field

```
Declaration: function GetFieldData(Field: TField; Buffer: Pointer) : Boolean
                ; Virtual; Overload
function GetFieldData(Field: TField; Buffer: Pointer;
                NativeFormat: Boolean) : Boolean; Virtual
                ; Overload
```

Visibility: public

Description: GetFieldData should copy the data for field Field from the internal dataset memory buffer into the memory pointed to by Buffer. This function is not intended for use by end-user applications, and should be used only in descendent classes, where it can be overridden. The function should return True if data was available and has been copied, or False if no data was available (in which case the field has value Null). The NativeFormat determines whether the data should be in native format (e.g. whether the date/time values should be in TDateTime format).

Errors: No checks are performed on the validity of the memory buffer

See also: `TField.DisplayText` ([435](#))

### 11.21.8 `TDataSet.SetFieldData`

Synopsis: Store the data for a field

Declaration: `procedure SetFieldData(Field: TField; Buffer: Pointer); Virtual  
; Overload  
procedure SetFieldData(Field: TField; Buffer: Pointer;  
NativeFormat: Boolean); Virtual; Overload`

Visibility: public

Description: `SetFieldData` should copy the data from field `Field`, stored in the memory pointed to by `Buffer` to the dataset memory buffer for the current record. This function is not intended for use by end-user applications, and should be used only in descendent classes, where it can be overridden. The `NativeFormat` determines whether the data is in native format (e.g. whether the date/time values are in `TDateTime` format).

See also: `TField.DisplayText` ([435](#))

### 11.21.9 `TDataSet.Append`

Synopsis: Append a new record to the data

Declaration: `procedure Append`

Visibility: public

Description: `Append` appends a new record at the end of the dataset. It is functionally equal to the `TDataSet.Insert` ([386](#)) call, but the cursor is positioned at the end of the dataset prior to performing the insert operation. The same events occur as when the `Insert` call is made.

See also: `TDataSet.Insert` ([386](#)), `TDataSet.Edit` ([381](#))

### 11.21.10 `TDataSet.AppendRecord`

Synopsis: Append a new record to the dataset and fill with data

Declaration: `procedure AppendRecord(const Values: Array of const)`

Visibility: public

Description: `AppendRecord` first calls `Append` to add a new record to the dataset. It then copies the values in `Values` to the various fields (using `TDataSet.SetFields` ([390](#))) and attempts to post the record using `TDataSet.Post` ([389](#)). If all went well, the result is that the values in `Values` have been added as a new record to the dataset.

Errors: Various errors may occur (not supplying a value for all required fields, invalid values) and may cause an exception. This may leave the dataset in editing mode.

See also: `TDataSet.Append` ([377](#)), `TDataSet.SetFields` ([390](#)), `TDataSet.Post` ([389](#))

**11.21.11 TDataSet.BookmarkValid**

Synopsis: Test whether ABookMark is a valid bookmark.

Declaration: `function BookmarkValid(ABookmark: TBookMark) : Boolean; Virtual`

Visibility: `public`

Description: `BookmarkValid` returns `True` if `ABookMark` is a valid bookmark for the dataset. Various operations can render a bookmark invalid: changing the sort order, closing and re-opening the dataset. `BookmarkValid` always returns `False` in `TDataSet`. Descendent classes must override this method to do an actual test.

Errors: If the bookmark is a completely arbitrary pointer, an exception may be raised.

See also: `TDataSet.GetBookmark` (384), `TDataSet.SetBookmark` (371), `TDataSet.FreeBookmark` (384), `TDataSet.BookmarkAvailable` (371)

**11.21.12 TDataSet.Cancel**

Synopsis: Cancel the current editing operation

Declaration: `procedure Cancel; Virtual`

Visibility: `public`

Description: `Cancel` cancels the current editing operation and sets the dataset again in browse mode. This operation triggers the `TDataSet.BeforeCancel` (403) and `TDataSet.AfterCancel` (404) events. If the dataset was in insert mode, then the `TDataSet.BeforeScroll` (404) and `TDataSet.AfterScroll` (405) events are triggered after and respectively before the `BeforeCancel` and `AfterCancel` events. If the dataset was not in one of the editing modes when `Cancel` is called, then nothing will happen.

See also: `TDataSet.State` (398), `TDataSet.Append` (377), `TDataSet.Insert` (386), `TDataSet.Edit` (381)

**11.21.13 TDataSet.CheckBrowseMode**

Synopsis: Check whether the dataset is in browse mode.

Declaration: `procedure CheckBrowseMode`

Visibility: `public`

Description: `CheckBrowseMode` will force the dataset to browse mode (`State=dsBrowse`) if it is active. If it is not active, an `EDatabaseError` (335) exception is raised. If it is active, but in an edit state, then `TDataSet.UpdateRecord` (391) is called, and if the `TDataSet.Modified` (396) property is true, a `TDataSet.Post` (389) is performed, else `TDataSet.Cancel` (378) is called.

See also: `TDataSet.State` (398), `TDataSet.Post` (389), `TDataSet.Cancel` (378), `TDataSet.UpdateRecord` (391), `TDataSet.Modified` (396)

**11.21.14 TDataSet.ClearFields**

Synopsis: Clear the values of all fields

Declaration: `procedure ClearFields`

Visibility: `public`

**Description:** `ClearFields` clears the values of all fields.

**Errors:** If the dataset is not in editing mode (`State` in `dsEditmodes`), then an `EDatabaseError` (335) exception will be raised.

See also: `TDataset.State` (398), `TField.Clear` (426)

### 11.21.15 `TDataset.Close`

**Synopsis:** Close the dataset

**Declaration:** `procedure Close`

**Visibility:** `public`

**Description:** `Close` closes the dataset if it is open (`Active=True`). This action triggers the `TDataset.BeforeClose` (401) and `TDataset.AfterClose` (401) events. If the dataset is not active, nothing happens.

**Errors:** If an exception occurs during the closing of the dataset, the `AfterClose` event will not be triggered.

See also: `TDataset.Active` (400), `TDataset.Open` (389)

### 11.21.16 `TDataset.ControlsDisabled`

**Synopsis:** Check whether the controls are disabled

**Declaration:** `function ControlsDisabled : Boolean`

**Visibility:** `public`

**Description:** `ControlsDisabled` returns `True` if the controls are disabled, i.e. no events are propagated to the controls connected to this dataset. The `TDataset.DisableControls` (381) call can be used to disable sending of data events to the controls. The sending can be re-enabled with `TDataset.EnableControls` (382). This mechanism has a counting mechanism: in order to enable sending of events to the controls, `EnableControls` must be called as much as `DisableControls` was called. The `ControlsDisabled` function will return `true` as long as the internal counter is not zero.

See also: `TDataset.DisableControls` (381), `TDataset.EnableControls` (382)

### 11.21.17 `TDataset.CompareBookmarks`

**Synopsis:** Compare two bookmarks

**Declaration:** `function CompareBookmarks(Bookmark1: TBookmark; Bookmark2: TBookmark)  
: LongInt; Virtual`

**Visibility:** `public`

**Description:** `CompareBookmarks` can be used to compare the relative positions of 2 bookmarks. It returns a negative value if `Bookmark1` is located before `Bookmark2`, zero if they refer to the same record, and a positive value if the second bookmark appears before the first bookmark. This function must be overridden by descendent classes of `TDataset`. The implementation in `TDataset` always returns zero.

**Errors:** No checks are performed on the validity of the bookmarks.

See also: `TDataset.BookmarkValid` (378), `TDataset.GetBookmark` (384), `TDataset.SetBookmark` (371)

**11.21.18 TDataSet.CreateBlobStream**

Synopsis: Create blob stream

Declaration: `function CreateBlobStream(Field: TField; Mode: TBlobStreamMode)  
: TStream; Virtual`

Visibility: public

Description: `CreateBlobStream` is not intended for use by application programmers. It creates a stream object which can be used to read or write data from a blob field. Instead, application programmers should use the `TBlobField.LoadFromStream` (349) and `TBlobField.SaveToStream` (350) methods when reading and writing data from/to BLOB fields. Which operation must be performed on the stream is indicated in the `Mode` parameter, and the `Field` parameter contains the field whose data should be read. The caller is responsible for freeing the stream created by this function.

See also: `TBlobField.LoadFromStream` (349), `TBlobField.SaveToStream` (350)

**11.21.19 TDataSet.CursorPosChanged**

Synopsis: Indicate a change in cursor position

Declaration: `procedure CursorPosChanged`

Visibility: public

Description: `CursorPosChanged` is not intended for internal use only, and serves to indicate that the current cursor position has changed. (it clears the internal cursor position).

**11.21.20 TDataSet.DataConvert**

Synopsis: Convert data from/to native format

Declaration: `procedure DataConvert(aField: TField; aSource: Pointer; aDest: Pointer;  
aToNative: Boolean); Virtual`

Visibility: public

Description: `DataConvert` converts the data from field `AField` in buffer `ASource` to native format and puts the result in `ADest`. If the `aToNative` parameter equals `False`, then the data is converted from native format to non-native format. Currently, only date/time/datetime and BCD fields are converted from/to native data. This means the routine handles conversion between `TDateTime` (the native format) and `TDateTimeRec`, and between `TBCD` and currency (the native format) for BCD fields. `DataConvert` is used internally by `TDataset` and descendent classes. There should be no need to use this routine in application code.

Errors: No checking on the validity of the buffer pointers is performed. If an invalid pointer is passed, an exception may be raised.

See also: `TDataset.GetFieldData` (376), `TDataset.SetFieldData` (377)

**11.21.21 TDataSet.Delete**

Synopsis: Delete the current record.

Declaration: `procedure Delete; Virtual`

Visibility: public

**Description:** `Delete` will delete the current record. This action will trigger the `TDataset.BeforeDelete` (404), `TDataset.BeforeScroll` (404), `TDataset.AfterDelete` (404) and `TDataset.AfterScroll` (405) events. If the dataset was in edit mode, the edits will be canceled before the delete operation starts.

**Errors:** If the dataset is empty or read-only, then an `EDatabaseError` (335) exception will be raised.

**See also:** `TDataset.Cancel` (378), `TDataset.BeforeDelete` (404), `TDataset.BeforeScroll` (404), `TDataset.AfterDelete` (404), `TDataset.AfterScroll` (405)

### 11.21.22 `TDataset.DisableControls`

**Synopsis:** Disable event propagation of controls

**Declaration:** `procedure DisableControls`

Visibility: public

**Description:** `DisableControls` tells the dataset to stop sending data-related events to the controls. This can be used before starting operations that will cause the current record to change a lot, or before any other lengthy operation that may cause a lot of events to be sent to the controls that show data from the dataset: each event will cause the control to update itself, which is a time-consuming operation that may also cause a lot of flicker on the screen.

The sending of events to the controls can be re-enabled with `Tdataset.EnableControls` (382). Note that for each call to `DisableControls`, a matching call to `EnableControls` must be made: an internal count is kept and only when the count reaches zero, the controls are again notified of changes to the dataset. It is therefore essential that the call to `EnableControls` is put in a `Finally` block:

```
MyDataset.DisableControls;
Try
    // Do some intensive stuff
Finally
    MyDataset.EnableControls
end;
```

**Errors:** Failure to call `enablecontrols` will prevent the controls from receiving updates. The state can be checked with `TDataset.ControlsDisabled` (379).

**See also:** `TDataset.EnableControls` (382), `TDataset.ControlsDisabled` (379)

### 11.21.23 `TDataset.Edit`

**Synopsis:** Set the dataset in editing mode.

**Declaration:** `procedure Edit`

Visibility: public

**Description:** `Edit` will set the dataset in edit mode: the contents of the current record can then be changed. This action will call the `TDataset.BeforeEdit` (402) and `TDataset.AfterEdit` (402) events. If the dataset was already in insert or edit mode, nothing will happen (the events will also not be triggered). If the dataset is empty, this action will execute `TDataset.Append` (377) instead.

**Errors:** If the dataset is read-only or not opened, then an `EDatabaseError` (335) exception will be raised.

**See also:** `TDataset.State` (398), `TDataset.EOF` (394), `TDataset.BOF` (392), `TDataset.Append` (377), `TDataset.BeforeEdit` (402), `TDataset.AfterEdit` (402)

### 11.21.24 TDataSet.EnableControls

Synopsis: Enable event propagation of controls

Declaration: `procedure EnableControls`

Visibility: `public`

Description: `EnableControls` tells the dataset to resume sending data-related events to the controls. This must be used after a call to `TDataSet.DisableControls` (381) to re-enable updating of controls.

Note that for each call to `DisableControls`, a matching call to `EnableControls` must be made: an internal count is kept and only when the count reaches zero, the controls are again notified of changes to the dataset. It is therefore essential that the call to `EnableControls` is put in a `Finally` block:

```
MyDataset.DisableControls;
Try
    // Do some intensive stuff
Finally
    MyDataset.EnableControls
end;
```

Errors: Failure to call `enablecontrols` will prevent the controls from receiving updates. The state can be checked with `TDataSet.ControlsDisabled` (379).

See also: `TDataSet.DisableControls` (381), `TDataSet.ControlsDisabled` (379)

### 11.21.25 TDataSet.FieldByName

Synopsis: Search a field by name

Declaration: `function FieldByName(const FieldName: string) : TField`

Visibility: `public`

Description: `FieldByName` is a shortcut for `Fields.FieldByName` (457): it searches for the field with `fieldname` equalling `FieldName`. The case is performed case-insensitive. The matching field instance is returned.

Errors: If the field is not found, an `EDatabaseError` (335) exception will be raised.

See also: `TFields.FieldByName` (457), `TDataSet.FindField` (382)

### 11.21.26 TDataSet.FindField

Synopsis: Find a field by name

Declaration: `function FindField(const FieldName: string) : TField`

Visibility: `public`

Description: `FindField` is a shortcut for `Fields.FindField` (456): it searches for the field with `fieldname` equalling `FieldName`. The case is performed case-insensitive. The matching field instance is returned, and if no match is found, `Nil` is returned.

See also: `TDataSet.FieldByName` (382), `TFields.FindField` (456)

### 11.21.27 TDataSet.FindFirst

Synopsis: Find the first active record (deprecated)

Declaration: `function FindFirst : Boolean; Virtual`

Visibility: `public`

Description: `FindFirst` positions the cursor on the first record (taking into account filtering), and returns `True` if the cursor position was changed. This method must be implemented by descendents of `TDataSet`: The implementation in `TDataSet` always returns `False`, indicating that the position was not changed.

This method is deprecated, use `TDataSet.First` (384) instead.

See also: `TDataSet.First` (384), `TDataSet.FindLast` (383), `TDataSet.FindNext` (383), `TDataSet.FindPrior` (383)

### 11.21.28 TDataSet.FindLast

Synopsis: Find the last active record (deprecated)

Declaration: `function FindLast : Boolean; Virtual`

Visibility: `public`

Description: `FindLast` positions the cursor on the last record (taking into account filtering), and returns `True` if the cursor position was changed. This method must be implemented by descendents of `TDataSet`: The implementation in `TDataSet` always returns `False`, indicating that the position was not changed.

This method is deprecated, use `TDataSet.Last` (387) instead.

See also: `TDataSet.Last` (387), `TDataSet.FindFirst` (383), `TDataSet.FindNext` (383), `TDataSet.FindPrior` (383)

### 11.21.29 TDataSet.FindNext

Synopsis: Find the next active record (deprecated)

Declaration: `function FindNext : Boolean; Virtual`

Visibility: `public`

Description: `FindNext` positions the cursor on the next record (taking into account filtering), and returns `True` if the cursor position was changed. This method must be implemented by descendents of `TDataSet`: The implementation in `TDataSet` always returns `False`, indicating that the position was not changed.

This method is deprecated, use `TDataSet.Next` (388) instead.

See also: `TDataSet.Next` (388), `TDataSet.FindFirst` (383), `TDataSet.FindLast` (383), `TDataSet.FindPrior` (383)

### 11.21.30 TDataSet.FindPrior

Synopsis: Find the previous active record (deprecated)

Declaration: `function FindPrior : Boolean; Virtual`

Visibility: `public`



**Description:** `FindPrior` positions the cursor on the previous record (taking into account filtering), and returns `True` if the cursor position was changed. This method must be implemented by descendents of `TDataset`: The implementation in `TDataset` always returns `False`, indicating that the position was not changed.

This method is deprecated, use `TDataset.Prior` (390) instead.

See also: `TDataset.Prior` (390), `TDataset.FindFirst` (383), `TDataset.FindLast` (383), `TDataset.FindPrior` (383)

### 11.21.31 `TDataset.First`

**Synopsis:** Position the dataset on the first record.

**Declaration:** `procedure First`

**Visibility:** `public`

**Description:** `First` positions the dataset on the first record. This action will trigger the `TDataset.BeforeScroll` (404) and `TDataset.AfterScroll` (405) events. After the action is completed, the `TDataset.BOF` (392) property will be `True`.

**Errors:** If the dataset is unidirectional or is closed, an `EDatabaseError` (335) exception will be raised.

See also: `TDataset.Prior` (390), `TDataset.Last` (387), `TDataset.Next` (388), `TDataset.BOF` (392), `TDataset.BeforeScroll` (404), `TDataset.AfterScroll` (405)

### 11.21.32 `TDataset.FreeBookmark`

**Synopsis:** Free a bookmark obtained with `GetBookmark` (deprecated)

**Declaration:** `procedure FreeBookmark(ABookmark: TBookmark); Virtual`

**Visibility:** `public`

**Description:** `FreeBookmark` must be used to free a bookmark obtained by `TDataset.GetBookmark` (384). It should not be used on bookmarks obtained with the `TDataset.Bookmark` (392) property. Both `GetBookmark` and `FreeBookmark` are deprecated. Use the `Bookmark` property instead: it uses a string type, which is automatically disposed of when the string variable goes out of scope.

See also: `TDataset.GetBookmark` (384), `TDataset.Bookmark` (392)

### 11.21.33 `TDataset.GetBookmark`

**Synopsis:** Get a bookmark pointer (deprecated)

**Declaration:** `function GetBookmark : TBookmark; Virtual`

**Visibility:** `public`

**Description:** `GetBookmark` gets a bookmark pointer to the current cursor location. The `TDataset.SetBookmark` (371) call can be used to return to the current record in the dataset. After use, the bookmark must be disposed of with the `TDataset.FreeBookmark` (384) call. The bookmark will be `Nil` if the dataset is empty or not active.

This call is deprecated. Use the `TDataset.Bookmark` (392) property instead to get a bookmark.

See also: `TDataset.SetBookmark` (371), `TDataset.FreeBookmark` (384), `TDataset.Bookmark` (392)

**11.21.34 TDataSet.GetCurrentRecord**

Synopsis: Copy the data for the current record in a memory buffer

Declaration: `function GetCurrentRecord(Buffer: TRecordBuffer) : Boolean; Virtual`

Visibility: public

Description: `GetCurrentRecord` can be overridden by `TDataSet` descendents to copy the data for the current record to `Buffer`. `Buffer` must point to a memory area, large enough to contain the data for the record. If the data is copied successfully to the buffer, the function returns `True`. The `TDataSet` implementation is empty, and returns `False`.

See also: `TDataSet.ActiveBuffer` (376)

**11.21.35 TDataSet.GetFieldList**

Synopsis: Return field instances in a list

Declaration: `procedure GetFieldList(List: TList; const FieldNames: string)`

Visibility: public

Description: `GetfieldList` parses `FieldNames` for names of fields, and returns the field instances that match the names in `list`. `FieldNames` must be a list of field names, separated by semicolons. The list is cleared prior to filling with the requested field instances.

Errors: If `FieldNames` contains a name of a field that does not exist in the dataset, then an `EDatabaseError` (335) exception will be raised.

See also: `TDataSet.GetFieldNames` (385), `TDataSet.FieldByName` (382), `TDataSet.FindField` (382)

**11.21.36 TDataSet.GetFieldNames**

Synopsis: Return a list of all available field names

Declaration: `procedure GetFieldNames(List: TStrings)`

Visibility: public

Description: `GetFieldNames` returns in `List` the names of all available fields, one field per item in the list. The dataset must be open for this function to work correctly.

See also: `TDataSet.GetFieldNameList` (371), `TDataSet.FieldByName` (382), `TDataSet.FindField` (382)

**11.21.37 TDataSet.GotoBookmark**

Synopsis: Jump to bookmark

Declaration: `procedure GotoBookmark(const ABookmark: TBookmark)`

Visibility: public

Description: `GotoBookmark` positions the dataset to the bookmark position indicated by `ABookMark`. `ABookmark` is a bookmark obtained by the `TDataSet.GetBookmark` (384) function.

This function is deprecated, use the `TDataSet.Bookmark` (392) property instead.

Errors: if `ABookmark` does not contain a valid bookmark, then an exception may be raised.

See also: `TDataSet.Bookmark` (392), `TDataSet.GetBookmark` (384), `TDataSet.FreeBookmark` (384)

**11.21.38 TDataSet.Insert**

Synopsis: Insert a new record at the current position.

Declaration: `procedure Insert`

Visibility: `public`

Description: `Insert` will insert a new record at the current position. When this function is called, any pending modifications (when the dataset already is in insert or edit mode) will be posted. After that, the `BeforeInsert` (401), `BeforeScroll` (404), `OnNewRecord` (407), `AfterInsert` (402) and `AfterScroll` (405) events are triggered in the order indicated here. The dataset is in the `dsInsert` state after this method is called, and the contents of the various fields can be set. To write the new record to the underlying database `TDataSet.Post` (389) must be called.

Errors: If the dataset is read-only, calling `Insert` will result in an `EDatabaseError` (335).

See also: `BeforeInsert` (401), `BeforeScroll` (404), `OnNewRecord` (407), `AfterInsert` (402), `AfterScroll` (405), `TDataSet.Post` (389), `TDataSet.Append` (377)

**11.21.39 TDataSet.InsertRecord**

Synopsis: Insert a new record with given values.

Declaration: `procedure InsertRecord(const Values: Array of const)`

Visibility: `public`

Description: `InsertRecord` is not yet implemented in Free Pascal. It does nothing.

See also: `TDataSet.Insert` (386), `TDataSet.SetFieldValues` (371)

**11.21.40 TDataSet.IsEmpty**

Synopsis: Check if the dataset contains no data

Declaration: `function IsEmpty : Boolean`

Visibility: `public`

Description: `IsEmpty` returns `True` if the dataset is empty, i.e. if `EOF` (394) and `TDataSet.BOF` (392) are both `True`, and the dataset is not in insert mode.

See also: `TDataSet.EOF` (394), `TDataSet.BOF` (392), `TDataSet.State` (398)

**11.21.41 TDataSet.IsLinkedTo**

Synopsis: Check whether a datasource is linked to the dataset

Declaration: `function IsLinkedTo(ADatasource: TDataSource) : Boolean`

Visibility: `public`

Description: `IsLinkedTo` returns `True` if `ADatasource` is linked to this dataset, either directly (the `ADatasource.Dataset`" (412) points to the current dataset instance, or indirectly.

See also: `TDataSource.Dataset` (412)

**11.21.42 TDataSet.IsSequenced**

Synopsis: Is the data sequenced

Declaration: `function IsSequenced : Boolean; Virtual`

Visibility: `public`

Description: `IsSequenced` indicates whether it is safe to use the `TDataSet.RecNo` (397) property to navigate in the records of the data. By default, this property is set to `True`, but `TDataSet` descendants may set this property to `False` (for instance, unidirectional datasets), in which case `RecNo` should not be used to navigate through the data.

See also: `TDataSet.RecNo` (397)

**11.21.43 TDataSet.Last**

Synopsis: Navigate forward to the last record

Declaration: `procedure Last`

Visibility: `public`

Description: `Last` puts the cursor at the last record in the dataset, fetching more records from the underlying database if needed. It is equivalent to moving to the last record and calling `TDataSet.Next` (388). After a call to `Last`, the `TDataSet.EOF` (394) property will be `True`.

Calling this method will trigger the `TDataSet.BeforeScroll` (404) and `TDataSet.AfterScroll` (405) events.

See also: `TDataSet.First` (384), `TDataSet.Next` (388), `TDataSet.EOF` (394), `TDataSet.BeforeScroll` (404), `TDataSet.AfterScroll` (405)

**11.21.44 TDataSet.Locate**

Synopsis: Locate a record based on some key values

Declaration: `function Locate(const KeyFields: string; const KeyValues: Variant;  
Options: TLocateOptions) : Boolean; Virtual`

Visibility: `public`

Description: `Locate` attempts to locate a record in the dataset. There are 2 possible cases when using `Locate`.

1. `Keyvalues` is a single value. In that case, `KeyFields` is the name of the field whose value must be matched to the value in `KeyValues`
2. `Keyvalues` is a variant array. In that case, `KeyFields` must contain a list of names of fields (separated by semicolons) whose values must be matched to the values in the `KeyValues` array

The matching always happens according to the `Options` parameter. For a description of the possible values, see `TLocateOption` (327).

If a record is found that matches the criteria, then the `locate` operation positions the cursor on this record, and returns `True`. If no record is found to match the criteria, `False` is returned, and the position of the cursor is unchanged.

The implementation in `TDataSet` always returns `False`. It is up to `TDataSet` descendants to implement this method and return an appropriate value.

See also: `TDataSet.Find` (371), `TDataSet.Lookup` (388), `TLocateOption` (327)

### 11.21.45 TDataSet.Lookup

Synopsis: Search for a record and return matching values.

Declaration: `function Lookup(const KeyFields: string; const KeyValues: Variant;  
const ResultFields: string) : Variant; Virtual`

Visibility: public

Description: `Lookup` always returns `Null` in `TDataSet`. Descendents of `TDataSet` can override this method to call `TDataSet.Locate` (387) to locate the record with fields `KeyFields` matching `KeyValues` and then to return the values of the fields in `ResultFields`. If `ResultFields` contains more than one fieldname (separated by semicolons), then the function returns an array. If there is only 1 fieldname, the value is returned directly.

Errors: If the dataset is unidirectional, then a `EDatabaseError` (335) exception will be raised.

See also: `TDataSet.Locate` (387)

### 11.21.46 TDataSet.MoveBy

Synopsis: Move the cursor position

Declaration: `function MoveBy(Distance: LongInt) : LongInt`

Visibility: public

Description: `MoveBy` moves the current record pointer with `Distance` positions. `Distance` may be a positive number, in which case the cursor is moved forward, or a negative number, in which case the cursor is moved backward. The move operation will stop as soon as the beginning or end of the data is reached. The `TDataSet.BeforeScroll` (404) and `TDataSet.AfterScroll` (405) events are triggered (once) when this method is called. The function returns the distance which was actually moved by the cursor.

Errors: A negative distance will result in an `EDatabaseError` (335) exception on unidirectional datasets.

See also: `TDataSet.RecNo` (397), `TDataSet.BeforeScroll` (404), `TDataSet.AfterScroll` (405)

### 11.21.47 TDataSet.Next

Synopsis: Go to the next record in the dataset.

Declaration: `procedure Next`

Visibility: public

Description: `Next` positions the cursor on the next record in the dataset. It is equivalent to a `MoveBy(1)` operation. Calling this method triggers the `TDataSet.BeforeScroll` (404) and `TDataSet.AfterScroll` (405) events. If the dataset is located on the last known record (`EOF` (394) is true), then no action is performed, and the events are not triggered.

Errors: Calling this method on a closed dataset will result in an `EDatabaseError` (335) exception.

See also: `TDataSet.MoveBy` (388), `TDataSet.Prior` (390), `TDataSet.Last` (387), `TDataSet.BeforeScroll` (404), `TDataSet.AfterScroll` (405), `TDataSet.EOF` (394)

**11.21.48 TDataSet.Open**

Synopsis: Activate the dataset: Fetch data into memory.

Declaration: `procedure Open`

Visibility: `public`

Description: `Open` must be used to make the `TDataSet` Active. It does nothing if the dataset is already active. `Open` initializes the `TDataSet` and brings the dataset in a browsable state:

Effectively the following happens:

1. The `BeforeOpen` event is triggered.
2. The descendents `InternalOpen` method is called to actually fetch data and initialize field-defs and field instances.
3. `BOF` (392) is set to `True`
4. Internal buffers are allocated and filled with data
5. If the dataset is empty, `EOF` (394) is set to `true`
6. `State` (398) is set to `dsBrowse`
7. The `AfterOpen` (401) event is triggered

Errors: If the descendent class cannot fetch the data, or the data does not match the field definitions present in the dataset, then an exception will be raised.

See also: `TDataSet.Active` (400), `TDataSet.State` (398), `TDataSet.BOF` (392), `TDataSet.EOF` (394), `TDataSet.BeforeOpen` (400), `TDataSet.AfterOpen` (401)

**11.21.49 TDataSet.Post**

Synopsis: Post pending edits to the database.

Declaration: `procedure Post; Virtual`

Visibility: `public`

Description: `Post` attempts to save pending edits when the dataset is in one of the edit modes: that is, after a `Insert` (386), `Append` (377) or `TDataSet.Edit` (381) operation. The changes will be committed to memory - and usually immediately to the underlying database as well. Prior to saving the data to memory, it will check some constraints: in `TDataSet`, the presence of a value for all required fields is checked. if for a required field no value is present, an exception will be raised. A call to `Post` results in the triggering of the `BeforePost` (403), `AfterPost` (403) events. After the call to `Post`, the `State` (398) of the dataset is again `dsBrowse`, i.e. the dataset is again in browse mode.

Errors: Invoking the `post` method when the dataset is not in one of the editing modes (`dsEditModes` (317)) will result in an `EdatabaseError` (335) exception. If an exception occurs during the save operation, the `OnPostError` (408) event is triggered to handle the error.

See also: `Insert` (386), `Append` (377), `Edit` (381), `OnPostError` (408), `BeforePost` (403), `AfterPost` (403), `State` (398)

**11.21.50 TDataSet.Prior**

Synopsis: Go to the previous record

Declaration: `procedure Prior`

Visibility: `public`

Description: `Prior` moves the cursor to the previous record. It is equivalent to a `MoveBy(-1)` operation. Calling this method triggers the `TDataSet.BeforeScroll` (404) and `TDataSet.AfterScroll` (405) events. If the dataset is located on the first record, (`BOF` (392) is true) then no action is performed, and the events are not triggered.

Errors: Calling this method on a closed dataset will result in an `EDatabaseError` (335) exception.

See also: `TDataSet.MoveBy` (388), `TDataSet.Next` (388), `TDataSet.First` (384), `TDataSet.BeforeScroll` (404), `TDataSet.AfterScroll` (405), `TDataSet.BOF` (392)

**11.21.51 TDataSet.Refresh**

Synopsis: Refresh the records in the dataset

Declaration: `procedure Refresh`

Visibility: `public`

Description: `Refresh` posts any pending edits, and refetches the data in the dataset from the underlying database, and attempts to reposition the cursor on the same record as it was. This operation is not supported by all datasets, and should be used with care. The repositioning may not always succeed, in which case the cursor will be positioned on the first record in the dataset. This is in particular true for unidirectional datasets. Calling `Refresh` results in the triggering of the `BeforeRefresh` (405) and `AfterRefresh` (405) events.

Errors: Refreshing may fail if the underlying dataset descendent does not support it.

See also: `TDataSet.Close` (379), `TDataSet.Open` (389), `BeforeRefresh` (405), `AfterRefresh` (405)

**11.21.52 TDataSet.Resync**

Synopsis: Resynchronize the data buffer

Declaration: `procedure Resync (Mode: TResyncMode); Virtual`

Visibility: `public`

Description: `Resync` refetches the records around the cursor position. It should not be used by application code, instead `TDataSet.Refresh` (390) should be used. The `Resync` parameter indicates how the buffers should be refreshed.

See also: `TDataSet.Refresh` (390)

**11.21.53 TDataSet.SetFields**

Synopsis: Set a number of field values at once

Declaration: `procedure SetFields (const Values: Array of const)`

Visibility: `public`

**Description:** `SetFields` sets the values of the fields with the corresponding values in the array. It starts with the first field in the `TDataset.Fields` (398) property, and works its way down the array.

**Errors:** If the dataset is not in edit mode, then an `EDatabaseError` (335) exception will be raised. If there are more values than fields, an `EListError` exception will be raised.

See also: `TDataset.Fields` (398)

#### 11.21.54 `TDataset.Translate`

**Synopsis:** Transliterate a buffer

**Declaration:** `function Translate(Src: PChar; Dest: PChar; ToOem: Boolean) : Integer; Virtual`

**Visibility:** public

**Description:** `Translate` is called for all string fields for which the `TStringField.Transliterate` (506) property is set to `True`. The `toOEM` parameter is set to `True` if the transliteration must happen from the used codepage to the codepage used for storage, and if it is set to `False` then the transliteration must happen from the native codepage to the storage codepage. This call must be overridden by descendants of `TDataset` to provide the necessary transliteration: `TDataset` just copies the contents of the `Src` buffer to the `Dest` buffer. The result must be the number of bytes copied to the destination buffer.

**Errors:** No checks are performed on the buffers.

See also: `TStringField.Transliterate` (506)

#### 11.21.55 `TDataset.UpdateCursorPos`

**Synopsis:** Update cursor position

**Declaration:** `procedure UpdateCursorPos`

**Visibility:** public

**Description:** `UpdateCursorPos` should not be used in application code. It is used to ensure that the logical cursor position is the correct (physical) position.

See also: `TDataset.Refresh` (390)

#### 11.21.56 `TDataset.UpdateRecord`

**Synopsis:** Indicate that the record contents have changed

**Declaration:** `procedure UpdateRecord`

**Visibility:** public

**Description:** `UpdateRecord` notifies controls that the contents of the current record have changed. It triggers the event. This should never be called by application code, and is intended only for descendants of `TDataset`.

See also: `OnUpdateRecord` (371)



### 11.21.57 TDataSet.UpdateStatus

Synopsis: Get the update status for the current record

Declaration: `function UpdateStatus : TUpdateStatus; Virtual`

Visibility: `public`

Description: `UpdateStatus` always returns `usUnModified` in the `TDataSet` implementation. Descendent classes should override this method to indicate the status for the current record in case they support cached updates: the function should return the status of the current record: has the record been locally inserted, modified or deleted, or none of these. `UpdateStatus` is not used in `TDataSet` itself, but is provided so applications have a unique API to work with datasets that have support for cached updates.

### 11.21.58 TDataSet.BlockReadSize

Synopsis: Number of records to read

Declaration: `Property BlockReadSize : Integer`

Visibility: `public`

Access: `Read,Write`

Description: `BlockReadSize` can be set to a positive number to prevent the dataset from sending notifications to DB-Aware controls while scrolling through the data. Setting it to zero will re-enable sending of notifications, as will putting the dataset in another state (edit etc.).

See also: `EnableControls` ([316](#)), `DisableControls` ([316](#))

### 11.21.59 TDataSet.BOF

Synopsis: Is the cursor at the beginning of the data (on the first record)

Declaration: `Property BOF : Boolean`

Visibility: `public`

Access: `Read`

Description: `BOF` returns `True` if the first record is the first record in the dataset, `False` otherwise. It will always be `True` if the dataset is just opened, or after a call to `TDataSet.First` ([384](#)). As soon as `TDataSet.Next` ([388](#)) is called, `BOF` will no longer be true.

See also: `TDataSet.EOF` ([394](#)), `TDataSet.Next` ([388](#)), `TDataSet.First` ([384](#))

### 11.21.60 TDataSet.Bookmark

Synopsis: Get or set the current cursor position

Declaration: `Property Bookmark : TBookMark`

Visibility: `public`

Access: `Read,Write`

**Description:** `Bookmark` can be read to obtain a bookmark to the current position in the dataset. The obtained value can be used to return to current position at a later stage. Writing the `Bookmark` property with a value previously obtained like this, will reposition the dataset on the same position as it was when the property was read.

This is often used when scanning all records, like this:

```
Var
  B : TBookmarkStr;

begin
  With MyDataset do
    begin
      B:=Bookmark;
      DisableControls;
    try
      First;
      While Not EOF do
        begin
          DoSomething;
        end;
      end;
    finally
      EnableControls;
      Bookmark:=B;
    end;
  end;
```

At the end of this code, the dataset will be positioned on the same record as when the code was started. The `TDataset.DisableControls` (381) and `TDataset.EnableControls` (382) calls prevent the controls from receiving update notifications as the dataset scrolls through the records, thus reducing flicker on the screen.

Note that bookmarks become invalid as soon as the dataset closes. A call to refresh may also destroy the bookmarks.

See also: `TDataset.DisableControls` (381), `TDataset.EnableControls` (382)

### 11.21.61 TDataSet.CanModify

**Synopsis:** Can the data in the dataset be modified

**Declaration:** `Property CanModify : Boolean`

**Visibility:** `public`

**Access:** `Read`

**Description:** `CanModify` indicates whether the dataset allows editing. `Unidirectional` datasets do not allow editing. Descendent datasets can impose additional conditions under which the data can not be modified (read-only datasets, for instance). If the `CanModify` property is `False`, then the edit, append or insert methods will fail.

See also: `TDataset.Insert` (386), `TDataset.Append` (377), `TDataset.Delete` (380), `Tdataset.Edit` (381)

### 11.21.62 TDataSet.DataSource

Synopsis: Datasource this dataset is connected to.

Declaration: Property DataSource : TDataSource

Visibility: public

Access: Read

Description: Datasource is the datasource this dataset is connected to, and from which it can get values for parameters. In TDataSet, the DataSource property is not used, and is always Nil. It is up to descendent classes that actually support a datasource to implement getter and setter routines for the DataSource property.

See also: TDataSource ([409](#))

### 11.21.63 TDataSet.DefaultFields

Synopsis: Is the dataset using persistent fields or not.

Declaration: Property DefaultFields : Boolean

Visibility: public

Access: Read

Description: DefaultFields is True if the fields were generated dynamically when the dataset was opened. If it is False then the field instances are persistent, i.e. they were created at design time with the fields editor. If DefaultFields is True, then for each item in the TDataSet.FieldDefs ([395](#)) property, a field instance is created. These fields instances are freed again when the dataset is closed. If DefaultFields is False, then there may be less field instances than there are items in the FieldDefs property. This can be the case for instance when opening a DBF file at runtime which has more fields than the file used at design time.

See also: TDataSet.FieldDefs ([395](#)), TDataSet.Fields ([398](#)), TField ([421](#))

### 11.21.64 TDataSet.EOF

Synopsis: Indicates whether the last record has been reached.

Declaration: Property EOF : Boolean

Visibility: public

Access: Read

Description: EOF is True if the cursor is on the last record in the dataset, and no more records are available. It is also True for an empty dataset. The EOF property will be set to True in the following cases:

- 1.The cursor is on the last record, and the TDataSet.Next ([388](#)) method is called.
- 2.The TDataSet.Last ([387](#)) method is called (which is equivalent to moving to the last record and calling TDataSet.Next ([388](#))).
- 3.The dataset is empty when opened.

In all other cases, `EOF` is `False`. Note: when the cursor is on the last-but-one record, and `Next` is called (moving the cursor to the last record), `EOF` will not yet be `True`. Only if both the cursor is on the last record **and** `Next` is called, will `EOF` become `True`.

This means that the following loop will stop after the last record was visited:

```
With MyDataset do
  While not EOF do
    begin
      DoSomething;
      Next;
    end;
```

See also: `TDataset.BOF` (392), `TDataset.Next` (388), `TDataset.Last` (387), `TDataset.IsEmpty` (386)

### 11.21.65 `TDataset.FieldCount`

Synopsis: Number of fields

Declaration: `Property FieldCount : LongInt`

Visibility: `public`

Access: `Read`

Description: `FieldCount` is the same as `Fields.Count` (458), i.e. the number of fields. For a dataset with persistent fields (when `DefaultFields` (394) is `False`) then this number will be always the same every time the dataset is opened. For a dataset with dynamically created fields, the number of fields may be different each time the dataset is opened.

See also: `TFields` (454)

### 11.21.66 `TDataset.FieldDefs`

Synopsis: Definitions of available fields in the underlying database

Declaration: `Property FieldDefs : TFieldDefs`

Visibility: `public`

Access: `Read, Write`

Description: `FieldDefs` is filled by the `TDataset` descendent when the dataset is opened. It represents the fields as they are returned by the particular database when the data is initially fetched from the engine. If the dataset uses dynamically created fields (when `DefaultFields` (394) is `True`), then for each item in this list, a field instance will be created with default properties available in the field definition. If the dataset uses persistent fields, then the fields in the field list will be checked against the items in the `FieldDefs` property. If no matching item is found for a persistent field, then an exception will be raised. Items that exist in the `fielddefs` property but for which there is no matching field instance, are ignored.

See also: `TDataset.Open` (389), `TDataset.DefaultFields` (394), `TDataset.Fields` (398)

**11.21.67 TDataSet.Found**

Synopsis: Check success of one of the `Find` methods

Declaration: `Property Found : Boolean`

Visibility: `public`

Access: `Read`

Description: `Found` is `True` if the last of one of the `TDataSet.FindFirst` (383), `TDataSet.FindLast` (383), `TDataSet.FindNext` (383) or `TDataSet.FindPrior` (383) operations was successful.

See also: `TDataSet.FindFirst` (383), `TDataSet.FindLast` (383), `TDataSet.FindNext` (383), `TDataSet.FindPrior` (383)

**11.21.68 TDataSet.Modified**

Synopsis: Was the current record modified ?

Declaration: `Property Modified : Boolean`

Visibility: `public`

Access: `Read`

Description: `Modified` is `True` if the current record was modified after a call to `Tdataset.Edit` (381) or `Tdataset.Insert` (386). It becomes `True` if a value was written to one of the fields of the dataset.

See also: `Tdataset.Edit` (381), `TDataSet.Insert` (386), `TDataSet.Append` (377), `TDataSet.Cancel` (378), `TDataSet.Post` (389)

**11.21.69 TDataSet.IsUniDirectional**

Synopsis: Is the dataset unidirectional (i.e. forward scrolling only)

Declaration: `Property IsUniDirectional : Boolean`

Visibility: `public`

Access: `Read`

Description: `IsUniDirectional` is `True` if the dataset is unidirectional. By default it is `False`, i.e. scrolling backwards is allowed. If the dataset is unidirectional, then any attempt to scroll backwards (using one of `TDataSet.Prior` (390) or `TDataSet.Last` (387)), random positioning of the cursor, editing or filtering will result in an `EDatabaseError` (335). Unidirectional datasets are also not suitable for display in a grid, as they have only 1 record in memory at any given time: they are only useful for performing an action on all records:

```
With MyDataset do
  While not EOF do
    begin
      DoSomething;
    Next;
  end;
```

See also: `TDataSet.Prior` (390), `TDataSet.Next` (388)

### 11.21.70 TDataSet.RecordCount

Synopsis: Number of records in the dataset

Declaration: Property RecordCount : LongInt

Visibility: public

Access: Read

Description: RecordCount is the number of records in the dataset. This number is not necessarily equal to the number of records returned by a query. For optimization purposes, a TDataSet descendent may choose not to fetch all records from the database when the dataset is opened. If this is the case, then the RecordCount will only reflect the number of records that have actually been fetched at the current time, and therefor the value will change as more records are fetched from the database.

Only when Last has been called (and the dataset has been forced to fetch all records returned by the database), will the value of RecordCount be equal to the number of records returned by the query.

In general, datasets based on in-memory data or flat files, will return the correct number of records in RecordCount.

See also: TDataSet.RecNo ([397](#))

### 11.21.71 TDataSet.RecNo

Synopsis: Current record number

Declaration: Property RecNo : LongInt

Visibility: public

Access: Read,Write

Description: RecNo returns the current position in the dataset. It can be written to set the cursor to the indicated position. This property must be implemented by TDataSet descendents, for TDataSet the property always returns -1.

This property should not be used if exact positioning is required. it is inherently unreliable.

See also: TDataSet.RecordCount ([397](#))

### 11.21.72 TDataSet.RecordSize

Synopsis: Size of the record in memory

Declaration: Property RecordSize : Word

Visibility: public

Access: Read

Description: RecordSize is the total size of the memory buffer used for the records. This property returns always 0 in the TDataSet implementation. Descendent classes should implement this property. Note that this property does not necessarily reflect the actual data size for the records. that may be more or less, depending on how the TDataSet descendent manages it's data.

See also: TField.Datasize ([435](#)), TDataSet.RecordCount ([397](#)), TDataSet.RecNo ([397](#))

### 11.21.73 TDataSet.State

Synopsis: Current operational state of the dataset

Declaration: `Property State : TDataSetState`

Visibility: `public`

Access: `Read`

Description: `State` determines the current operational state of the dataset. During its lifetime, the dataset is in one of many states, depending on which operation is currently in progress:

- If a dataset is closed, the `State` is `dsInactive`.
- As soon as it is opened, it is in `dsBrowse` mode, and remains in this state while changing the cursor position.
- If the `Edit` or `Insert` or `Append` methods is called, the `State` changes to `dsEdit` or `dsInsert`, respectively.
- As soon as edits have been posted or cancelled, the state is again `dsBrowse`.
- Closing the dataset sets the state again to `dsInactive`.

There are some other states, mainly connected to internal operations, but which can become visible in some of the dataset's events.

See also: `TDataSet.Active` (400), `TDataSet.Edit` (381), `TDataSet.Insert` (386), `TDataSet.Append` (377), `TDataSet.Post` (389), `TDataSet.Cancel` (378)

### 11.21.74 TDataSet.Fields

Synopsis: Indexed access to the fields of the dataset.

Declaration: `Property Fields : TFields`

Visibility: `public`

Access: `Read`

Description: `Fields` provides access to the fields of the dataset. It is of type `TFields` (454) and therefore gives indexed access to the fields, but also allows other operations such as searching for fields based on their names or getting a list of fieldnames.

See also: `TFieldDefs` (451), `TField` (421)

### 11.21.75 TDataSet.FieldValues

Synopsis: Access to field values based on the field names.

Declaration: `Property FieldValues[FieldName: string]: Variant; default`

Visibility: `public`

Access: `Read,Write`

Description: `FieldValues` provides array-like access to the values of the fields, based on the names of the fields. The value is read or written as a variant type. It is equivalent to the following:

```
FieldByName(FieldName).AsVariant
```

It can be read as well as written.

See also: `TFields.FieldByName` (457)

### 11.21.76 TDataSet.Filter

Synopsis: Filter to apply to the data in memory.

Declaration: `Property Filter : string`

Visibility: `public`

Access: `Read,Write`

Description: `Filter` is not implemented by `TDataSet`. It is up to descendent classes to implement actual filtering: the filtering happens on in-memory data, and is not applied on the database level. (in particular: setting the filter property will in no way influence the `WHERE` clause of an SQL-based dataset).

In general, the `filter` property accepts a SQL-like syntax usually encountered in the `WHERE` clause of an SQL `SELECT` statement.

The filter is only applied if the `Filtered` property is set to `True`. If the `Filtered` property is `False`, the `Filter` property is ignored.

See also: `TDataSet.Filtered` (399), `TDataSet.FilterOptions` (399)

### 11.21.77 TDataSet.Filtered

Synopsis: Is the filter active or not.

Declaration: `Property Filtered : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `Filtered` determines whether the filter condition in `TDataSet.Filter` (399) is applied or not. The filter is only applied if the `Filtered` property is set to `True`. If the `Filtered` property is `False`, the `Filter` property is ignored.

See also: `TDataSet.Filter` (399), `TDataSet.FilterOptions` (399)

### 11.21.78 TDataSet.FilterOptions

Synopsis: Options to apply when filtering

Declaration: `Property FilterOptions : TFilterOptions`

Visibility: `public`

Access: `Read,Write`

Description: `FilterOptions` determines what options should be taken into account when applying the filter in `TDataSet.Filter` (399), such as case-sensitivity or whether to treat an asterisk as a wildcard: By default, an asterisk (\*) at the end of a literal string in the filter expression is treated as a wildcard. When `FilterOptions` does not include `foNoPartialCompare`, strings that have an asterisk at the end, indicate a partial string match. In that case, the asterisk matches any number of characters. If `foNoPartialCompare` is included in the options, the asterisk is regarded as a regular character.

See also: `TDataSet.Filter` (399), `TDataSet.FilterOptions` (399)



**11.21.79 TDataSet.Active**

Synopsis: Is the dataset open or closed.

Declaration: Property Active : Boolean

Visibility: public

Access: Read,Write

Description: Active is True if the dataset is open, and False if it is closed (TDataSet.State (398) is then dsInactive). Setting the Active property to True is equivalent to calling TDataSet.Open (389), setting it to False is equivalent to calling TDataSet.Close (379)

See also: TDataSet.State (398), TDataSet.Open (389), TDataSet.Close (379)

**11.21.80 TDataSet.AutoCalcFields**

Synopsis: How often should the value of calculated fields be calculated

Declaration: Property AutoCalcFields : Boolean

Visibility: public

Access: Read,Write

Description: AutoCalcFields is by default true, meaning that the values of calculated fields will be computed in the following cases:

- When the dataset is opened
- When the dataset is put in edit mode
- When a data field changed

When AutoCalcFields is False, then the calculated fields are called whenever

- The dataset is opened
- The dataset is put in edit mode

Both proper calculated fields and lookup fields are computed. Calculated fields are computed through the TDataSet.OnCalcFields (406) event.

See also: TField.FieldKind (442), TDataSet.OnCalcFields (406)

**11.21.81 TDataSet.BeforeOpen**

Synopsis: Event triggered before the dataset is opened.

Declaration: Property BeforeOpen : TDataSetNotifyEvent

Visibility: public

Access: Read,Write

Description: BeforeOpen is triggered before the dataset is opened. No actions have been performed yet when this event is called, and the dataset is still in dsInactive state. It can be used to set parameters and options that influence the opening process. If an exception is raised during the event handler, the dataset remains closed.

See also: TDataSet.AfterOpen (401), TDataSet.State (398)

### 11.21.82 TDataSet.AfterOpen

Synopsis: Event triggered after the dataset is opened.

Declaration: Property `AfterOpen` : `TDataSetNotifyEvent`

Visibility: public

Access: Read,Write

Description: `AfterOpen` is triggered after the dataset is opened. The dataset has fetched its data and is in `dsBrowse` state when this event is triggered. If the dataset is not empty, then a `TDataSet.AfterScroll` (405) event will be triggered immediately after the `AfterOpen` event. If an exception is raised during the event handler, the dataset remains open, but the `AfterScroll` event will not be triggered.

See also: `TDataSet.AfterOpen` (401), `TDataSet.State` (398), `TDataSet.AfterScroll` (405)

### 11.21.83 TDataSet.BeforeClose

Synopsis: Event triggered before the dataset is closed.

Declaration: Property `BeforeClose` : `TDataSetNotifyEvent`

Visibility: public

Access: Read,Write

Description: `BeforeClose` is triggered before the dataset is closed. No actions have been performed yet when this event is called, and the dataset is still in `dsBrowse` state or one of the editing states. It can be used to prevent closing of the dataset, for instance if there are pending changes not yet committed to the database. If an exception is raised during the event handler, the dataset remains opened.

See also: `TDataSet.AfterClose` (401), `TDataSet.State` (398)

### 11.21.84 TDataSet.AfterClose

Synopsis: Event triggered after the dataset is closed

Declaration: Property `AfterClose` : `TDataSetNotifyEvent`

Visibility: public

Access: Read,Write

Description: `AfterOpen` is triggered after the dataset is opened. The dataset has discarded its data and has cleaned up its internal memory structures. It is in `dsInactive` state when this event is triggered.

See also: `TDataSet.BeforeClose` (401), `TDataSet.State` (398)

### 11.21.85 TDataSet.BeforeInsert

Synopsis: Event triggered before the dataset is put in insert mode.

Declaration: Property `BeforeInsert` : `TDataSetNotifyEvent`

Visibility: public

Access: Read,Write

**Description:** `BeforeInsert` is triggered at the start of the `TDataset.Append` (377) or `TDataset.Insert` (386) methods. The dataset is still in `dsBrowse` state when this event is triggered. If an exception is raised in the `BeforeInsert` event handler, then the dataset will remain in `dsBrowse` state, and the append or insert operation is cancelled.

See also: `TDataset.AfterInsert` (402), `TDataset.Append` (377), `TDataset.Insert` (386)

### 11.21.86 `TDataset.AfterInsert`

**Synopsis:** Event triggered after the dataset is put in insert mode.

**Declaration:** `Property AfterInsert : TDatasetNotifyEvent`

**Visibility:** public

**Access:** Read,Write

**Description:** `AfterInsert` is triggered after the dataset has finished putting the dataset in `dsInsert` state and it has initialized the new record buffer. This event can be used e.g. to set initial field values. After the `AfterInsert` event, the `TDataset.AfterScroll` (405) event is still triggered. Raising an exception in the `AfterInsert` event, will prevent the `AfterScroll` event from being triggered, but does not undo the insert or append operation.

See also: `TDataset.BeforeInsert` (401), `TDataset.AfterScroll` (405), `TDataset.Append` (377), `TDataset.Insert` (386)

### 11.21.87 `TDataset.BeforeEdit`

**Synopsis:** Event triggered before the dataset is put in edit mode.

**Declaration:** `Property BeforeEdit : TDatasetNotifyEvent`

**Visibility:** public

**Access:** Read,Write

**Description:** `BeforeEdit` is triggered at the start of the `TDataset.Edit` (381) method. The dataset is still in `dsBrowse` state when this event is triggered. If an exception is raised in the `BeforeEdit` event handler, then the dataset will remain in `dsBrowse` state, and the edit operation is cancelled.

See also: `TDataset.AfterEdit` (402), `TDataset.Edit` (381), `TDataset.State` (398)

### 11.21.88 `TDataset.AfterEdit`

**Synopsis:** Event triggered after the dataset is put in edit mode.

**Declaration:** `Property AfterEdit : TDatasetNotifyEvent`

**Visibility:** public

**Access:** Read,Write

**Description:** `AfterEdit` is triggered after the dataset has finished putting the dataset in `dsEdit` state and it has initialized the edit buffer for the record. Raising an exception in the `AfterEdit` event does not undo the edit operation.

See also: `TDataset.BeforeEdit` (402), `TDataset.Edit` (381), `TDataset.State` (398)

### 11.21.89 TDataSet.BeforePost

Synopsis: Event called before changes are posted to the underlying database

Declaration: `Property BeforePost : TDataSetNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `BeforePost` is triggered at the start of the `TDataSet.Post` (389) method, when the dataset is still in one of the edit states (`dsEdit,dsInsert`). If the dataset was not in an edit state when `Post` is called, the `BeforePost` event is not triggered. This event can be used to supply values for required fields that have no value yet (the `Post` operation performs the check on required fields only after this event), or it can be used to abort the post operation: if an exception is raised during the `BeforePost` operation, the posting operation is cancelled, and the dataset remains in the editing state it was in before the post operation.

See also: `TDataSet.post` (389), `TDataSet.AfterPost` (403), `TDataSet.State` (398)

### 11.21.90 TDataSet.AfterPost

Synopsis: Event called after changes have been posted to the underlying database

Declaration: `Property AfterPost : TDataSetNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `AfterPost` is triggered when the `TDataSet.Post` (389) operation was successfully completed, and the dataset is again in `dsBrowse` state. If an error occurred during the post operation, then the `AfterPost` event is not called, but the `TDataSet.OnPostError` (408) event is triggered instead.

See also: `TDataSet.BeforePost` (403), `TDataSet.Post` (389), `TDataSet.State` (398), `TDataSet.OnPostError` (408)

### 11.21.91 TDataSet.BeforeCancel

Synopsis: Event triggered before a Cancel operation.

Declaration: `Property BeforeCancel : TDataSetNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `BeforeCancel` is triggered at the start of the `TDataSet.Cancel` (378) operation, when the state is still one of the editing states (`dsEdit,dsInsert`). The event handler can be used to abort the cancel operation: if an exception is raised during the event handler, then the cancel operation stops. If the dataset was not in one of the editing states when the `Cancel` method was called, then the event is not triggered.

See also: `TDataSet.AfterCancel` (404), `TDataSet.Cancel` (378), `TDataSet.State` (398)

### 11.21.92 TDataSet.AfterCancel

Synopsis: Event triggered after a Cancel operation.

Declaration: `Property AfterCancel : TDataSetNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `AfterCancel` is triggered when the `TDataSet.Cancel` (378) operation was successfully completed, and the dataset is again in `dsBrowse` state.

See also: `TDataSet.BeforeCancel` (403), `TDataSet.Cancel` (378), `TDataSet.State` (398)

### 11.21.93 TDataSet.BeforeDelete

Synopsis: Event triggered before a Delete operation.

Declaration: `Property BeforeDelete : TDataSetNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `BeforeDelete` is triggered at the start of the `TDataSet.Delete` (380) operation, when the dataset is still in `dsBrowse` state. The event handler can be used to abort the delete operation: if an exception is raised during the event handler, then the delete operation stops. The event is followed by a `TDataSet.BeforeScroll` (404) event. If the dataset was in insert mode when the `Delete` method was called, then the event will not be called, as `TDataSet.Cancel` (378) is called instead.

See also: `TDataSet.AfterDelete` (404), `TDataSet.Delete` (380), `TDataSet.BeforeScroll` (404), `TDataSet.Cancel` (378), `TDataSet.State` (398)

### 11.21.94 TDataSet.AfterDelete

Synopsis: Event triggered after a successful Delete operation.

Declaration: `Property AfterDelete : TDataSetNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `AfterDelete` is triggered after the successful completion of the `TDataSet.Delete` (380) operation, when the dataset is again in `dsBrowse` state. The event is followed by a `TDataSet.AfterScroll` (405) event.

See also: `TDataSet.BeforeDelete` (404), `TDataSet.Delete` (380), `TDataSet.AfterScroll` (405), `TDataSet.State` (398)

### 11.21.95 TDataSet.BeforeScroll

Synopsis: Event triggered before the cursor changes position.

Declaration: `Property BeforeScroll : TDataSetNotifyEvent`

Visibility: `public`

Access: Read,Write

**Description:** `BeforeScroll` is triggered before the cursor changes position. This can happen with one of the navigation methods: `TDataset.Next` (388), `TDataset.Prior` (390), `TDataset.First` (384), `TDataset.Last` (387), but also with two of the editing operations: `TDataset.Insert` (386) and `TDataset.Delete` (380). Raising an exception in this event handler aborts the operation in progress.

See also: `TDataset.AfterScroll` (405), `TDataset.Next` (388), `TDataset.Prior` (390), `TDataset.First` (384), `TDataset.Last` (387), `TDataset.Insert` (386), `TDataset.Delete` (380)

### 11.21.96 `TDataset.AfterScroll`

**Synopsis:** Event triggered after the cursor has changed position.

**Declaration:** `Property AfterScroll : TDatasetNotifyEvent`

Visibility: public

Access: Read,Write

**Description:** `AfterScroll` is triggered after the cursor has changed position. This can happen with one of the navigation methods: `TDataset.Next` (388), `TDataset.Prior` (390), `TDataset.First` (384), `TDataset.Last` (387), but also with two of the editing operations: `TDataset.Insert` (386) and `TDataset.Delete` (380) and after the dataset was opened. It is suitable for displaying status information or showing a value that needs to be calculated for each record.

See also: `TDataset.AfterScroll` (405), `TDataset.Next` (388), `TDataset.Prior` (390), `TDataset.First` (384), `TDataset.Last` (387), `TDataset.Insert` (386), `TDataset.Delete` (380), `TDataset.Open` (389)

### 11.21.97 `TDataset.BeforeRefresh`

**Synopsis:** Event triggered before the data is refreshed.

**Declaration:** `Property BeforeRefresh : TDatasetNotifyEvent`

Visibility: public

Access: Read,Write

**Description:** `BeforeRefresh` is triggered at the start of the `TDataset.Refresh` (390) method, after the dataset has been put in browse mode. If the dataset cannot be put in browse mode, the `BeforeRefresh` method will not be triggered. If an exception is raised during the `BeforeRefresh` method, then the refresh method is cancelled and the dataset remains in the `dsBrowse` state.

See also: `TDataset.Refresh` (390), `TDataset.AfterRefresh` (405), `TDataset.State` (398)

### 11.21.98 `TDataset.AfterRefresh`

**Synopsis:** Event triggered after the data has been refreshed.

**Declaration:** `Property AfterRefresh : TDatasetNotifyEvent`

Visibility: public

Access: Read,Write

**Description:** `AfterRefresh` is triggered at the end of the `TDataset.Refresh` (390) method, after the dataset has refreshed its data and is again in `dsBrowse` state. This event can be used to react on changes in data in the current record

See also: `TDataset.Refresh` (390), `TDataset.State` (398), `TDataset.BeforeRefresh` (405)

**11.21.99 TDataSet.OnCalcFields**

Synopsis: Event triggered when values for calculated fields must be computed.

Declaration: `Property OnCalcFields : TDataSetNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `OnCalcFields` is triggered whenever the dataset needs to (re)compute the values of any calculated fields in the dataset. It is called very often, so this event should return as quickly as possible. Only the values of the calculated fields should be set, no methods of the dataset that change the data or cursor position may be called during the execution of this event handler. The frequency with which this event is called can be controlled through the `TDataSet.AutoCalcFields` (400) property. Note that the value of lookup fields does not need to be calculated in this event, their value is computed automatically before this event is triggered.

See also: `TDataSet.AutoCalcFields` (400), `TField.Kind` (421)

**11.21.100 TDataSet.OnDeleteError**

Synopsis: Event triggered when a delete operation fails.

Declaration: `Property OnDeleteError : TDataSetErrorEvent`

Visibility: `public`

Access: `Read,Write`

Description: `OnDeleteError` is triggered when the `TDataSet.Delete` (380) method fails to delete the record in the underlying database. The event handler can be used to indicate what the response to the failed delete should be. To this end, it gets the exception object passed to it (parameter `E`), and it can examine this object to return an appropriate action in the `DataAction` parameter. The following responses are supported:

**daFail**The operation should fail (an exception will be raised)

**daAbort**The operation should be aborted (edits are undone, and an `EAbort` exception is raised)

**daRetry**Retry the operation.

For more information, see also the description of the `TDataSetErrorEvent` (321) event handler type.

See also: `TDataSetErrorEvent` (321), `TDataSet.Delete` (380), `TDataSet.OnEditError` (406), `TDataSet.OnPostError` (408)

**11.21.101 TDataSet.OnEditError**

Synopsis: Event triggered when an edit operation fails.

Declaration: `Property OnEditError : TDataSetErrorEvent`

Visibility: `public`

Access: `Read,Write`

**Description:** `OnEditError` is triggered when the `TDataset.Edit` (381) method fails to put the dataset in edit mode because the underlying database engine reported an error. The event handler can be used to indicate what the response to the failed edit operation should be. To this end, it gets the exception object passed to it (parameter `E`), and it can examine this object to return an appropriate action in the `DataAction` parameter. The following responses are supported:

**daFail**The operation should fail (an exception will be raised)

**daAbort**The operation should be aborted (edits are undone, and an `EAbort` exception is raised)

**daRetry**Retry the operation.

For more information, see also the description of the `TDatasetErrorEvent` (321) event handler type.

See also: `TDatasetErrorEvent` (321), `TDataset.Edit` (381), `TDataset.OnDeleteError` (406), `TDataset.OnPostError` (408)

### 11.21.102 TDataSet.OnFilterRecord

**Synopsis:** Event triggered to filter records.

**Declaration:** `Property OnFilterRecord : TFilterRecordEvent`

**Visibility:** public

**Access:** Read,Write

**Description:** `OnFilterRecord` can be used to provide event-based filtering for datasets that support it. This event is only triggered when the `Tdataset.Filtered` (399) property is set to `True`. The event handler should set the `Accept` parameter to `True` if the current record should be accepted, or to `False` if it should be rejected. No methods that change the state of the dataset may be used during this event, and calculated fields or lookup field values are not yet available.

See also: `TDataset.Filter` (399), `TDataset.Filtered` (399), `TDataset.state` (398)

### 11.21.103 TDataSet.OnNewRecord

**Synopsis:** Event triggered when a new record is created.

**Declaration:** `Property OnNewRecord : TDataSetNotifyEvent`

**Visibility:** public

**Access:** Read,Write

**Description:** `OnNewRecord` is triggered by the `TDataset.Append` (377) or `TDataset.Insert` (386) methods when the buffer for the new record's data has been allocated. This event can be used to set default value for some of the fields in the dataset. If an exception is raised during this event handler, the operation is cancelled and the dataset is put again in browse mode (`TDataset.State` (398) is again `dsBrowse`).

See also: `TDataset.Append` (377), `TDataset.Insert` (386), `TDataset.State` (398)



### 11.21.104 TDataSet.OnPostError

**Synopsis:** Event triggered when a post operation fails.

**Declaration:** `Property OnPostError : TDataSetErrorEvent`

**Visibility:** public

**Access:** Read, Write

**Description:** `OnPostError` is triggered when the `TDataSet.Post` (389) method fails to post the changes in the dataset buffer to the underlying database, because the database engine reported an error. The event handler can be used to indicate what the response to the failed post operation should be. To this end, it gets the exception object passed to it (parameter `E`), and it can examine this object to return an appropriate action in the `DataAction` parameter. The following responses are supported:

**daFail**The operation should fail (an exception will be raised)

**daAbort**The operation should be aborted (edits are undone, and an `EAbort` exception is raised)

**daRetry**Retry the operation.

For more information, see also the description of the `TDataSetErrorEvent` (321) event handler type.

See also: `TDataSetErrorEvent` (321), `TDataSet.Post` (389), `TDataSet.OnDeleteError` (406), `TDataSet.OnEditError` (406)

## 11.22 TDataSetEnumerator

### 11.22.1 Description

`TDataSetEnumerator` is an enumerator for the records in a dataset. It returns the `TDataSet.Fields` (398) instance. It navigates from the first till the last record in the dataset. The following is an example of how this can be used, in conjunction with the field enumerator:

```
var
  Rec : TFields;
  Fld : TField;

begin
  for Rec in MyDataset do
    for F in Rec do
      Writeln(F.Name, ' : ', F.AsString);
```

Note that the current record pointer of the dataset is modified as the loop is traversed. If the current record is modified by other code while the loop is running, the result may become unpredictable. Similarly, if 2 enumerators are used simultaneously for the same dataset, the results are unpredictable.

See also: `TDataSet` (371), `TFields` (454), `TFieldsEnumerator` (459)

### 11.22.2 Method overview

Page	Method	Description
409	Create	Create a new instance of the dataset enumerator
409	MoveNext	Attempts to navigate to the next record

### 11.22.3 Property overview

Page	Properties	Access	Description
<a href="#">409</a>	Current	r	Current record

### 11.22.4 TDataSetEnumerator.Create

Synopsis: Create a new instance of the dataset enumerator

Declaration: `constructor Create (ADataset: TDataSet)`

Visibility: `public`

Description: `Create` saves the dataset for later use, and puts the dataset on the first record.

Errors: None.

See also: `TDataSet` ([371](#)), `TDataSet.First` ([384](#)), `TFieldsEnumerator` ([459](#))

### 11.22.5 TDataSetEnumerator.MoveNext

Synopsis: Attempts to navigate to the next record

Declaration: `function MoveNext : Boolean`

Visibility: `public`

Description: `MoveNext` attempts to navigate to the next record. It returns `True` if the attempt was successful, `False` if not (EOF is true).

See also: `TDataSet.Next` ([388](#)), `TDataSet.EOF` ([394](#))

### 11.22.6 TDataSetEnumerator.Current

Synopsis: Current record

Declaration: `Property Current : TFields`

Visibility: `public`

Access: Read

Description: `Current` always returns `TDataSet.Fields` ([398](#)).

See also: `TDataSet.Fields` ([398](#))

## 11.23 TDataSource

### 11.23.1 Description

`TDataSource` is a mediating component: it handles communication between any DB-Aware component (often edit controls on a form) and a `TDataSet` ([371](#)) instance. Any database aware component should never communicate with a dataset directly. Instead, it should communicate with a `TDataSource` ([409](#)) instance. The `TDataSet` instance will communicate with the `TDataSource` instance, which will notify every component attached to it. Vice versa, any component that wishes to make changes to the dataset, will notify the `TDataSource` instance, which will then (if needed) notify the `TDataSet` instance. The datasource can be disabled, in which case all communication between the dataset and the DB-Aware components is suspended until the datasource is again enabled.

See also: [TDataSet \(371\)](#), [TDatalink \(366\)](#)

### 11.23.2 Method overview

Page	Method	Description
<a href="#">410</a>	Create	Create a new instance of <code>TDataSource</code>
<a href="#">410</a>	Destroy	Remove a <code>TDataSource</code> instance from memory
<a href="#">410</a>	Edit	Put the dataset in edit mode, if needed
<a href="#">411</a>	IsLinkedTo	Check if a dataset is linked to a certain dataset

### 11.23.3 Property overview

Page	Properties	Access	Description
<a href="#">411</a>	AutoEdit	rw	Should the dataset be put in edit mode automatically
<a href="#">412</a>	DataSet	rw	Dataset this <code>datasource</code> is connected to
<a href="#">412</a>	Enabled	rw	Enable or disable sending of events
<a href="#">413</a>	OnDataChange	rw	Called whenever data changes in the current record
<a href="#">412</a>	OnStateChange	rw	Called whenever the state of the dataset changes
<a href="#">413</a>	OnUpdateData	rw	Called whenever the data in the dataset must be updated
<a href="#">411</a>	State	r	State of the dataset

### 11.23.4 TDataSource.Create

Synopsis: Create a new instance of `TDataSource`

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initializes a new instance of `TDataSource`. It simply allocates some resources and then calls the inherited constructor.

See also: `TDataSource.Destroy` ([410](#))

### 11.23.5 TDataSource.Destroy

Synopsis: Remove a `TDataSource` instance from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` notifies all `TDataLink` ([366](#)) instances connected to it that the dataset is no longer available, and then removes itself from the `TDatalink` instance. It then cleans up all resources and calls the inherited constructor.

See also: `TDataSource.Create` ([410](#)), `TDatalink` ([366](#))

### 11.23.6 TDataSource.Edit

Synopsis: Put the dataset in edit mode, if needed

Declaration: `procedure Edit`

Visibility: `public`

**Description:** `Edit` will check `AutoEdit` (411): if it is `True`, then it puts the `Dataset` (412) it is connected to in edit mode, if it was in browse mode. If `AutoEdit` is `False`, then nothing happens. Application or component code that deals with GUI development should always attempt to set a dataset in edit mode through this method instead of calling `TDataset.Edit` (381) directly.

**Errors:** An `EDatabaseError` (335) exception can occur if the dataset is read-only or fails to set itself in edit mode. (e.g. unidirectional datasets).

**See also:** `TDatasource.AutoEdit` (411), `TDataset.Edit` (381), `TDataset.State` (398)

### 11.23.7 `TDatasource.IsLinkedTo`

**Synopsis:** Check if a dataset is linked to a certain dataset

**Declaration:** `function IsLinkedTo(ADataset: TDataset) : Boolean`

**Visibility:** public

**Description:** `IsLinkedTo` checks if it is somehow linked to `ADataset`: it checks the `Dataset` (412) property, and returns `True` if it is the same. If not, it continues by checking any detail dataset fields that the dataset possesses (recursively). This function can be used to detect circular links in e.g. master-detail relationships.

**See also:** `TDatasource.Dataset` (412)

### 11.23.8 `TDatasource.State`

**Synopsis:** State of the dataset

**Declaration:** `Property State : TDatasetState`

**Visibility:** public

**Access:** Read

**Description:** `State` contains the `State` (398) of the dataset it is connected to, or `dsInactive` if the dataset property is not set or the `datasource` is not enabled. Components connected to a dataset through a `datasource` property should always check `TDatasource.State` instead of checking `TDataset.State` (398) directly, to take into account the effect of the `Enabled` (412) property.

**See also:** `TDataset.State` (398), `TDatasource.Enabled` (412)

### 11.23.9 `TDatasource.AutoEdit`

**Synopsis:** Should the dataset be put in edit mode automatically

**Declaration:** `Property AutoEdit : Boolean`

**Visibility:** published

**Access:** Read,Write

**Description:** `AutoEdit` can be set to `True` to prevent visual controls from putting the dataset in edit mode. Visual controls use the `TDatasource.Edit` (410) method to attempt to put the dataset in edit mode as soon as the user changes something. If `AutoEdit` is set to `False` then the `Edit` method does nothing. The effect is that the user must explicitly set the dataset in edit mode (by clicking some button or some other action) before the fields can be edited.

**See also:** `TDatasource.Edit` (410), `TDataset.Edit` (381)

### 11.23.10 TDataSource.DataSet

Synopsis: Dataset this datasource is connected to

Declaration: `Property DataSet : TDataSet`

Visibility: published

Access: Read,Write

Description: `DataSet` must be set by the application programmer to the `TDataSet` (371) instance for which this datasource is handling events. Setting it to `Nil` will disable all controls that are connected to this datasource instance. Once it is set and the datasource is enabled, the datasource will start sending data events to the controls or components connected to it.

See also: `TDataSet` (371), `TDataSource.Enabled` (412)

### 11.23.11 TDataSource.Enabled

Synopsis: Enable or disable sending of events

Declaration: `Property Enabled : Boolean`

Visibility: published

Access: Read,Write

Description: `Enabled` is by default set to `True`: the datasource instance communicates events from the dataset to components connected to the datasource, and vice versa: components can interact with the dataset. If the `Enabled` property is set to `False` then no events are communicated to connected components: it is as if the dataset property was set to `Nil`. Reversely, the components cannot interact with the dataset if the `Enabled` property is set to `False`.

See also: `TDataSet` (371), `TDataSource.DataSet` (412), `TDataSource.AutoEdit` (411)

### 11.23.12 TDataSource.OnStateChange

Synopsis: Called whenever the state of the dataset changes

Declaration: `Property OnStateChange : TNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnStateChange` is called whenever the `TDataSet.State` (398) property changes, and the datasource is enabled. It can be used in application code to react to state changes: enabling or disabling non-DB-Aware controls, setting empty values etc.

See also: `TDataSource.OnUpdateData` (413), `TDataSource.OnStateChange` (412), `TDataSet.State` (398), `TDataSource.Enabled` (412)

### 11.23.13 TDataSource.OnDataChange

**Synopsis:** Called whenever data changes in the current record

**Declaration:** Property OnDataChange : TDataChangeEvent

**Visibility:** published

**Access:** Read,Write

**Description:** OnDatachange is called whenever a field value changes: if the `Field` parameter is set, a single field value changed. If the `Field` parameter is `Nil`, then the whole record changed: when the dataset is opened, when the user scrolls to a new record. This event handler can be set to react to data changes: to update the contents of non-DB-aware controls for instance. The event is not called when the datasource is not enabled.

**See also:** TDataSource.OnUpdateData (413), TDataSource.OnStateChange (412), TDataSet.AfterScroll (405), TField.OnChange (446), TDataSource.Enabled (412)

### 11.23.14 TDataSource.OnUpdateData

**Synopsis:** Called whenever the data in the dataset must be updated

**Declaration:** Property OnUpdateData : TNotifyEvent

**Visibility:** published

**Access:** Read,Write

**Description:** OnUpdateData is called whenever the dataset needs the latest data from the controls: usually just before a `TDataSet.Post` (389) operation. It can be used to copy data from non-db-aware controls to the dataset just before the dataset is posting the changes to the underlying database.

**See also:** TDataSource.OnDataChange (413), TDataSource.OnStateChange (412), TDataSet.Post (389)

## 11.24 TDateField

### 11.24.1 Description

`TDateField` is the class used when a dataset must manage data of type date. (`TField.DataType` (435) equals `ftDate`). It initializes some of the properties of the `TField` (421) class to be able to work with date fields.

It should never be necessary to create an instance of `TDateField` manually, a field of this class will be instantiated automatically for each date field when a dataset is opened.

**See also:** TDataSet (371), TField (421), TDateTimeField (414), TTimeField (507)

### 11.24.2 Method overview

Page	Method	Description
414	Create	Create a new instance of a <code>TDateField</code> class.

### 11.24.3 TDateField.Create

**Synopsis:** Create a new instance of a TDateField class.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** public

**Description:** Create initializes a new instance of the TDateField class. It calls the inherited destructor, and then sets some TField (421) properties to configure the instance for working with date values.

See also: TField (421)

## 11.25 TDateTimeField

### 11.25.1 Description

TDateTimeField is the class used when a dataset must manage data of type datetime. (TField.DataType (435) equals ftDateTime). It also serves as base class for the TDateField (413) or TTimeField (507) classes. It overrides some of the properties and methods of the TField (421) class to be able to work with date/time fields.

It should never be necessary to create an instance of TDateTimeField manually, a field of this class will be instantiated automatically for each datetime field when a dataset is opened.

See also: TDataSet (371), TField (421), TDateField (413), TTimeField (507)

### 11.25.2 Method overview

Page	Method	Description
414	Create	Create a new instance of a TDateTimeField class.

### 11.25.3 Property overview

Page	Properties	Access	Description
415	DisplayFormat	rw	Formatting string for textual representation of the field
415	EditMask		Specify an edit mask for an edit control
415	Value	rw	Contents of the field as a TDateTime value

### 11.25.4 TDateTimeField.Create

**Synopsis:** Create a new instance of a TDateTimeField class.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** public

**Description:** Create initializes a new instance of the TDateTimeField class. It calls the inherited destructor, and then sets some TField (421) properties to configure the instance for working with date/time values.

See also: TField (421)

### 11.25.5 TDateTimeField.Value

Synopsis: Contents of the field as a TDateTime value

Declaration: `Property Value : TDateTime`

Visibility: public

Access: Read,Write

Description: `Value` is redefined from `TField.Value` (439) by `TDateTimeField` as a `TDateTime` value. It returns the same value as the `TField.AsDateTime` (430) property.

See also: `TField.AsDateTime` (430), `TField.Value` (439)

### 11.25.6 TDateTimeField.DisplayFormat

Synopsis: Formatting string for textual representation of the field

Declaration: `Property DisplayFormat : string`

Visibility: published

Access: Read,Write

Description: `DisplayFormat` can be set to a formatting string that will then be used by the `TField.DisplayText` (435) property to format the value with the `DateTimeToString` (??)function.

See also: `DateTimeToString` (??), `FormatDateTime` (??), `TField.DisplayText` (435)

### 11.25.7 TDateTimeField.EditMask

Synopsis: Specify an edit mask for an edit control

Declaration: `Property EditMask :`

Visibility: published

Access:

Description: `EditMask` can be used to specify an edit mask for controls that allow to edit this field. It has no effect on the field value, and serves only to ensure that the user can enter only correct data for this field.

`TDateTimeField` just changes the visibility of the `EditMask` property, it is introduced in `TField`. For more information on valid edit masks, see the documentation of the GUI controls.

See also: `TField.EditMask` (436)

## 11.26 TDBDataset

### 11.26.1 Description

`TDBDataset` is a `TDataset` descendent which introduces the concept of a database: a central component (`TDatabase` (362)) which represents a connection to a database. This central component is exposed in the `TDBDataset.Database` (416) property. When the database is no longer connected, or is no longer in memory, all `TDBDataset` instances connected to it are disabled.



`TDBDataset` also introduces the notion of a transaction, exposed in the `Transaction` (417) property.

`TDBDataset` is an abstract class, it should never be used directly.

Dataset component writers should descend their component from `TDBDataset` if they wish to introduce a central database connection component. The database connection logic will be handled automatically by `TDBDataset`.

See also: `TDatabase` (362), `TDBTransaction` (417)

### 11.26.2 Method overview

Page	Method	Description
416	<code>destroy</code>	Remove the <code>TDBDataset</code> instance from memory.

### 11.26.3 Property overview

Page	Properties	Access	Description
416	<code>DataBase</code>	rw	Database this dataset is connected to
417	<code>Transaction</code>	rw	Transaction in which this dataset is running.

### 11.26.4 TDBDataset.destroy

Synopsis: Remove the `TDBDataset` instance from memory.

Declaration: `destructor destroy; Override`

Visibility: `public`

Description: `Destroy` will disconnect the `TDBDataset` from its `Database` (416) and `Transaction` (417). After this it calls the inherited destructor.

See also: `TDBDataset.Database` (416), `TDatabase` (362)

### 11.26.5 TDBDataset.Database

Synopsis: Database this dataset is connected to

Declaration: `Property DataBase : TDataBase`

Visibility: `public`

Access: `Read,Write`

Description: `Database` should be set to the `TDatabase` (362) instance this dataset is connected to. It can only be set when the dataset is closed.

Descendent classes should check in the property setter whether the database instance is of the correct class.

Errors: If the property is set when the dataset is active, an `EDatabaseError` (335) exception will be raised.

See also: `TDatabase` (362), `TDBDataset.Transaction` (417)

### 11.26.6 TDBDataset.Transaction

Synopsis: Transaction in which this dataset is running.

Declaration: `Property Transaction : TDBTransaction`

Visibility: public

Access: Read,Write

Description: `Transaction` points to a `TDBTransaction` (417) component that represents the transaction this dataset is active in. This property should only be used for databases that support transactions.

The property can only be set when the dataset is disabled.

See also: `TDBTransaction` (417), `TDBDataset.Database` (416)

## 11.27 TDBTransaction

### 11.27.1 Description

`TDBTransaction` encapsulates a SQL transaction. It is an abstract class, and should be used by component creators that wish to encapsulate transactions in a class. The `TDBTransaction` class offers functionality to refer to a `TDatabase` (362) instance, and to keep track of `TDataset` instances which are connected to the transaction.

See also: `TDatabase` (362), `TDataset` (371)

### 11.27.2 Method overview

Page	Method	Description
418	<code>CloseDataSets</code>	Close all connected datasets
417	<code>Create</code>	Transaction property
418	<code>Destroy</code>	Remove a <code>TDBTransaction</code> instance from memory.

### 11.27.3 Property overview

Page	Properties	Access	Description
418	<code>Active</code>	rw	Is the transaction active or not
418	<code>DataBase</code>	rw	Database this transaction is connected to

### 11.27.4 TDBTransaction.Create

Synopsis: Transaction property

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` initializes a new `TDBTransaction` instance. It sets up the necessary resources, after having called the inherited constructor.

See also: `TDBTransaction.Destroy` (418)

### 11.27.5 TDBTransaction.Destroy

Synopsis: Remove a `TDBTransaction` instance from memory.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` first disconnects all connected `TBDDataset` (415) instances and then cleans up the resources allocated in the `Create` (417) constructor. After that it calls the inherited destructor.

See also: `TDBTransaction.Create` (417)

### 11.27.6 TDBTransaction.CloseDataSets

Synopsis: Close all connected datasets

Declaration: `procedure CloseDataSets`

Visibility: `public`

Description: `CloseDataSets` closes all connected datasets (All `TBDDataset` (415) instances whose `Transaction` (417) property points to this `TDBTransaction` instance).

See also: `TBDDataset` (415), `TBDDataset.Transaction` (417)

### 11.27.7 TDBTransaction.DataBase

Synopsis: Database this transaction is connected to

Declaration: `Property DataBase : TDataBase`

Visibility: `public`

Access: `Read,Write`

Description: `DataBase` points to the database that this transaction is part of. This property can be set only when the transaction is not active.

Errors: Setting this property to a new value when the transaction is active will result in an `EDatabaseError` (335) exception.

See also: `TDBTransaction.Active` (418), `TDataBase` (362)

### 11.27.8 TDBTransaction.Active

Synopsis: Is the transaction active or not

Declaration: `Property Active : Boolean`

Visibility: `published`

Access: `Read,Write`

Description: `Active` is `True` if a transaction was started using `TDBTransaction.StartTransaction` (417). Reversely, setting `Active` to `True` will call `StartTransaction`, setting it to `False` will call `TDBTransaction.EndTransaction` (417).

See also: `TDBTransaction.StartTransaction` (417), `TDBTransaction.EndTransaction` (417)

## 11.28 TDefCollection

### 11.28.1 Description

`TDefCollection` is a parent class for the `TFieldDefs` (451) and `TIndexDefs` (469) collections: It holds a set of named definitions on behalf of a `TDataset` (371) component. To this end, it introduces a dataset (420) property, and a mechanism to notify the dataset of any updates in the collection. It is supposed to hold items of class `TNamedItem` (482), so the `TDefCollection.Find` (419) method can find items by named.

### 11.28.2 Method overview

Page	Method	Description
419	<code>create</code>	Instantiate a new <code>TDefCollection</code> instance.
419	<code>Find</code>	Find an item by name
420	<code>GetItemNames</code>	Return a list of all names in the collection
420	<code>IndexOf</code>	Find location of item by name

### 11.28.3 Property overview

Page	Properties	Access	Description
420	<code>Dataset</code>	r	Dataset this collection manages definitions for.
420	<code>Updated</code>	rw	Has one of the items been changed

### 11.28.4 TDefCollection.create

Synopsis: Instantiate a new `TDefCollection` instance.

Declaration: `constructor create(ADataset: TDataset; AOwner: TPersistent;  
AClass: TCollectionItemClass)`

Visibility: public

Description: `Create` saves the `ADataset` and `AOwner` components in local variables for later reference, and then calls the inherited `Create` with `AClass` as a parameter. `AClass` should at least be of type `TNamedItem`. `ADataset` is the dataset on whose behalf the collection is managed. `AOwner` is the owner of the collection, normally this is the form or datamodule on which the dataset is dropped.

See also: `TDataset` (371), `TNamedItem` (482)

### 11.28.5 TDefCollection.Find

Synopsis: Find an item by name

Declaration: `function Find(const AName: string) : TNamedItem`

Visibility: public

Description: `Find` searches for an item in the collection with name `AName` and returns the item if it is found. If no item with the requested name is found, `Nil` is returned. The search is performed case-insensitive.

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

See also: `TNamedItem.Name` (483), `TDefCollection.IndexOf` (420)

### 11.28.6 TDefCollection.GetItemNames

Synopsis: Return a list of all names in the collection

Declaration: `procedure GetItemNames(List: TStrings)`

Visibility: public

Description: `GetItemNames` fills `List` with the names of all items in the collection. It clears the list first.

Errors: If `List` is not a valid `TStrings` instance, an exception will occur.

See also: `TNamedItem.Name` ([483](#))

### 11.28.7 TDefCollection.IndexOf

Synopsis: Find location of item by name

Declaration: `function IndexOf(const AName: string) : LongInt`

Visibility: public

Description: `IndexOf` searches in the collection for an item whose `Name` property matches `AName` and returns the index of the item if it finds one. If no item is found, -1 is returned. The search is performed case-insensitive.

See also: `TDefCollection.Find` ([419](#)), `TNamedItem.Name` ([483](#))

### 11.28.8 TDefCollection.Dataset

Synopsis: Dataset this collection manages definitions for.

Declaration: `Property Dataset : TDataSet`

Visibility: public

Access: Read

Description: `Dataset` is the dataset this collection manages definitions for. It must be supplied when the collection is created and cannot change during the lifetime of the collection.

### 11.28.9 TDefCollection.Updated

Synopsis: Has one of the items been changed

Declaration: `Property Updated : Boolean`

Visibility: public

Access: Read, Write

Description: `Changed` indicates whether the collection has changed: an item was added or removed, or one of the properties of the items was changed.

## 11.29 TDetailDataLink

### 11.29.1 Description

`TDetailDataLink` handles the communication between a detail dataset and the master datasource in a master-detail relationship between datasets. It should never be used in an application, and should only be used by component writers that wish to provide master-detail functionality for `TDataset` descendents.

See also: `TDataset` ([371](#)), `TDataSource` ([409](#))

### 11.29.2 Property overview

Page	Properties	Access	Description
<a href="#">421</a>	<code>DetailDataSet</code>	<code>r</code>	Detail dataset in Master-detail relation

### 11.29.3 TDetailDataLink.DetailDataSet

Synopsis: Detail dataset in Master-detail relation

Declaration: `Property DetailDataSet : TDataset`

Visibility: `public`

Access: `Read`

Description: `DetailDataset` is the detail dataset in a master-detail relationship between 2 datasets. `DetailDataset` is always `Nil` in `TDetailDataLink` and is only filled in in descendent classes like `TMasterDataLink` ([477](#)). The master dataset is available through the regular `TDataLink.DataSource` ([370](#)) property.

See also: `TDataset` ([371](#)), `TMasterDataLink` ([477](#)), `TDataLink.DataSource` ([370](#))

## 11.30 TField

### 11.30.1 Description

`TField` is an abstract class that defines access methods for a field in a record, controlled by a `TDataset` ([371](#)) instance. It provides methods and properties to access the contents of the field in the current record. Reading one of the `AsXXX` properties of `TField` will access the field contents and return the contents as the desired type. Writing one of the `AsXXX` properties will write a value to the buffer represented by the `TField` instance.

`TField` is an abstract class, meaning that it should never be created directly. `TDataset` instances always create one of the descendent classes of `TField`, depending on the type of the underlying data.

See also: `TDataset` ([371](#)), `TFieldDef` ([447](#)), `TFields` ([454](#))

**11.30.2 Method overview**

Page	Method	Description
<a href="#">425</a>	Assign	Copy properties from one TField instance to another
<a href="#">425</a>	AssignValue	Assign value of a variant record to the field.
<a href="#">426</a>	Clear	Clear the field contents.
<a href="#">425</a>	Create	Create a new TField instance
<a href="#">425</a>	Destroy	Destroy the TField instance
<a href="#">426</a>	FocusControl	Set focus to the first control connected to this field.
<a href="#">426</a>	GetData	Get the data from this field
<a href="#">427</a>	IsBlob	Is the field a BLOB field (untyped data of indeterminate size).
<a href="#">427</a>	IsValidChar	Check whether a character is valid input for the field
<a href="#">427</a>	RefreshLookupList	Refresh the lookup list
<a href="#">427</a>	SetData	Save the field data
<a href="#">428</a>	SetFieldType	Set the field data type
<a href="#">428</a>	Validate	Validate the data buffer





### 11.30.3 Property overview

Page	Properties	Access	Description
440	Alignment	rw	Alignment for this field
432	AsAnsiString	rw	Return field contents as an ANSI string
428	AsBCD	rw	Access the field's contents as a BCD (Binary coded Decimal)
429	AsBoolean	rw	Access the field's contents as a Boolean value.
429	AsBytes	rw	Retrieve the contents of the field as an array of bytes
429	AsCurrency	rw	Access the field's contents as a Currency value.
430	AsDateTime	rw	Access the field's contents as a TDateTime value.
430	AsFloat	rw	Access the field's contents as a floating-point (Double) value.
431	AsInteger	rw	Access the field's contents as a 32-bit signed integer (longint) value.
431	AsLargeInt	rw	Access the field's contents as a 64-bit signed integer (longint) value.
430	AsLongint	rw	Access the field's contents as a 32-bit signed integer (longint) value.
431	AsString	rw	Access the field's contents as an AnsiString value.
432	AsUnicodeString	rw	Field contents as a UnicodeString
432	AsUTF8String	rw	Field contents as a UTF8 String
433	AsVariant	rw	Access the field's contents as a Variant value.
433	AsWideString	rw	Access the field's contents as a WideString value.
433	AttributeSet	rw	Not used: dictionary information
434	Calculated	rw	Is the field a calculated field ?
434	CanModify	r	Can the field's contents be modified.
440	ConstraintErrorMessage	rw	Message to display if the CustomConstraint constraint is violated.
434	CurValue	r	Current value of the field
440	CustomConstraint	rw	Custom constraint for the field's value
434	DataSet	rw	Dataset this field belongs to
435	DataSet	r	Size of the field's data
435	DataType	r	The data type of the field.
441	DefaultExpression	rw	Default value for the field
441	DisplayLabel	rws	Name of the field for display purposes
435	DisplayName	r	User-readable fieldname
435	DisplayText	r	Formatted field value
441	DisplayWidth	rws	Width of the field in characters
436	EditMask	rw	Specify an edit mask for an edit control
436	EditMaskPtr	r	Alias for EditMask
439	FieldDef	r	Fielddef associated with this field
442	FieldKind	rw	The kind of field.
442	FieldName	rw	Name of the field
436	FieldNo	r	Number of the field in the record
442	HasConstraints	r	Does the field have any constraints defined
443	ImportedConstraint	rw	Constraint for the field value on the level of the underlying database
442	Index	rw	Index of the field in the list of fields
437	IsIndexField	r	Is the field an indexed field ?
437	IsNull	r	Is the field empty
443	KeyFields	rw	Key fields to use when looking up a field value.
444	Lookup	rws	Is the field a lookup field
443	LookupCache	rw	Should lookup values be cached
443	LookupDataSet	rw	Dataset with lookup values
444	LookupKeyFields	rw	Names of fields on which to perform a locate
439	LookupList	r	List of lookup values
444	LookupResultField	rw	Name of field to use as lookup value
437	NewValue	rw	The new value of the field
437	Offset	r	Offset of the field's value in the dataset buffer
439	OldValue	r	Old value of the field
446	OnChange	rw	Event triggered when the field's value has changed

### 11.30.4 TField.Create

Synopsis: Create a new TField instance

Declaration: constructor Create(AOwner: TComponent); Override

Visibility: public

Description: Create creates a new TField instance and sets up initial values for the fields. TField is a component, and AOwner will be used as the owner of the TField instance. This usually will be the form or datamodule on which the dataset was placed. There should normally be no need for a programmer to create a Tfield instance manually. The TDataSet.Open (389) method will create the necessary TField instances, if none had been created in the designer.

See also: TDataSet.Open (389)

### 11.30.5 TField.Destroy

Synopsis: Destroy the TField instance

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy cleans up any structures set up by the field instance, and then calls the inherited destructor. There should be no need to call this method under normal circumstances: the dataset instance will free any TField instances it has created when the dataset was opened.

See also: TDataSet.Close (379)

### 11.30.6 TField.Assign

Synopsis: Copy properties from one TField instance to another

Declaration: procedure Assign(Source: TPersistent); Override

Visibility: public

Description: Assign is overridden by TField to copy the field value (not the field properties) from Source if it exists. If Source is Nil then the value of the field is cleared.

Errors: If Source is not a TField instance, then an exception will be raised.

See also: TField.Value (439)

### 11.30.7 TField.AssignValue

Synopsis: Assign value of a variant record to the field.

Declaration: procedure AssignValue(const AValue: TVarRec)

Visibility: public

Description: AssignValue assigns the value of a "array of const" record AValue (of type TVarRec) to the field's value. If the record contains a TPersistent instance, it will be used as argument for the Assign to the field.

The dataset must be in edit mode to execute this method.

**Errors:** If the `AValue` contains an unsupported value (such as a non-nil pointer) then an exception will be raised. If the dataset is not in one of the edit modes, then executing this method will raise an `EDatabaseError` (335) exception.

See also: `TField.Assign` (425), `TField.Value` (439)

### 11.30.8 TField.Clear

**Synopsis:** Clear the field contents.

**Declaration:** `procedure Clear; Virtual`

**Visibility:** `public`

**Description:** `Clear` clears the contents of the field. After calling this method the value of the field is `Null` and `IsNull` (437) returns `True`.

The dataset must be in edit mode to execute this method.

**Errors:** If the dataset is not in one of the edit modes, then executing this method will raise an `EDatabaseError` (335) exception.

See also: `TField.IsNull` (437), `TField.Value` (439)

### 11.30.9 TField.FocusControl

**Synopsis:** Set focus to the first control connected to this field.

**Declaration:** `procedure FocusControl`

**Visibility:** `public`

**Description:** `FocusControl` will set focus to the first control that is connected to this field.

**Errors:** If the control cannot receive focus, then this method will raise an exception.

See also: `TDataset.EnableControls` (382), `TDataset.DisableControls` (381)

### 11.30.10 TField.GetData

**Synopsis:** Get the data from this field

**Declaration:** `function GetData(Buffer: Pointer) : Boolean; Overload`  
`function GetData(Buffer: Pointer; NativeFormat: Boolean) : Boolean`  
`; Overload`

**Visibility:** `public`

**Description:** `GetData` is used internally by `TField` to fetch the value of the data of this field into the data buffer pointed to by `Buffer`. If it returns `False` if the field has no value (i.e. is `Null`). If the `NativeFormat` parameter is true, then date/time formats should use the `TDateTime` format. It should not be necessary to use this method, instead use the various 'AsXXX' methods to access the data.

**Errors:** No validity checks are performed on `Buffer`: it should point to a valid memory area, and should be large enough to contain the value of the field. Failure to provide a buffer that matches these criteria will result in an exception.

See also: `TField.IsNull` (437), `TField.SetData` (427), `TField.Value` (439)

### 11.30.11 TField.IsBlob

Synopsis: Is the field a BLOB field (untyped data of indeterminate size).

Declaration: `class function IsBlob : Boolean; Virtual`

Visibility: public

Description: `IsBlob` returns `True` if the field is one of the blob field types. The `TField` implementation returns `false`. Only one of the blob-type field classes override this function and let it return `True`.

Errors: None.

See also: `TBlobField.IsBlob` (349)

### 11.30.12 TField.IsValidChar

Synopsis: Check whether a character is valid input for the field

Declaration: `function IsValidChar(InputChar: Char) : Boolean; Virtual`

Visibility: public

Description: `IsValidChar` checks whether `InputChar` is a valid characters for the current field. It does this by checking whether `InputChar` is in the set of characters specified by the `TField.ValidChars` (438) property. The `ValidChars` property will be initialized to a correct set of characters by descendent classes. For instance, a numerical field will only accept numerical characters and the sign and decimal separator characters.

Descendent classes can override this method to provide custom checks. The `ValidChars` property can be set to restrict the list of valid characters to a subset of what would normally be available.

See also: `TField.ValidChars` (438)

### 11.30.13 TField.RefreshLookupList

Synopsis: Refresh the lookup list

Declaration: `procedure RefreshLookupList`

Visibility: public

Description: `RefreshLookupList` fills the lookup list for a lookup fields with all key, value pairs found in the lookup dataset. It will open the lookup dataset if needed. The lookup list is only used if the `TField.LookupCache` (443) property is set to `True`.

Errors: If the values of the various lookup properties is not correct or the lookup dataset cannot be opened, then an exception will be raised.

See also: `LookupDataset` (443), `LookupKeyFields` (444), `LookupResultField` (444)

### 11.30.14 TField.SetData

Synopsis: Save the field data

Declaration: `procedure SetData(Buffer: Pointer); Overload`  
`procedure SetData(Buffer: Pointer; NativeFormat: Boolean); Overload`

Visibility: public

**Description:** `SetData` saves the value of the field data in `Buffer` to the dataset internal buffer. The `Buffer` pointer should point to a memory buffer containing the data for the field in the correct format. If the `NativeFormat` parameter is true, then date/time formats should use the `TDateTime` format.

There should normally not be any need to call `SetData` directly: it is called by the various setter methods of the `AsXXX` properties of `TField`.

**Errors:** No validity checks are performed on `Buffer`: it should point to a valid memory area, and should be large enough to contain the value of the field. Failure to provide a buffer that matches these criteria will result in an exception.

See also: `TField.GetData` (426), `TField.Value` (439)

### 11.30.15 TField.SetFieldType

**Synopsis:** Set the field data type

**Declaration:** `procedure SetFieldType(AValue: TFieldType); Virtual`

**Visibility:** public

**Description:** `SetFieldType` does nothing, but it can be overridden by descendent classes to provide special handling when the field type is set.

See also: `TField.DataType` (435)

### 11.30.16 TField.Validate

**Synopsis:** Validate the data buffer

**Declaration:** `procedure Validate(Buffer: Pointer)`

**Visibility:** public

**Description:** `Validate` is called by `SetData` prior to writing the data from `Buffer` to the dataset buffer. It will call the `TField.OnValidate` (447) event handler, if one is set, to allow the application programmer to program additional checks.

See also: `TField.SetData` (427), `TField.OnValidate` (447)

### 11.30.17 TField.AsBCD

**Synopsis:** Access the field's contents as a BCD (Binary coded Decimal)

**Declaration:** `Property AsBCD : TBCD`

**Visibility:** public

**Access:** Read,Write

**Description:** `AsBCD` can be used to read or write the contents of the field as a BCD value (Binary Coded Decimal). If the native type of the field is not BCD, then an attempt will be made to convert the field value from the native format to a BCD value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefore, when reading or writing a field value for a field whose native data type is not a BCD value, an exception may be raised.

See also: `TField.AsCurrency` (429), `TField.Value` (439)

### 11.30.18 TField.AsBoolean

Synopsis: Access the field's contents as a Boolean value.

Declaration: `Property AsBoolean : Boolean`

Visibility: `public`

Access: Read,Write

Description: `AsBoolean` can be used to read or write the contents of the field as a boolean value. If the native type of the field is not Boolean, then an attempt will be made to convert the field value from the native format to a boolean value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefore, when reading or writing a field value for a field whose native data type is not a Boolean value (for instance a string value), an exception may be raised.

See also: `TField.Value` ([439](#)), `TField.AsInteger` ([431](#))

### 11.30.19 TField.AsBytes

Synopsis: Retrieve the contents of the field as an array of bytes

Declaration: `Property AsBytes : TBytes`

Visibility: `public`

Access: Read,Write

Description: `AsBytes` returns the contents of the field as an array of bytes. For blob data this is the actual blob content.

See also: `TBlobField` ([348](#))

### 11.30.20 TField.AsCurrency

Synopsis: Access the field's contents as a Currency value.

Declaration: `Property AsCurrency : Currency`

Visibility: `public`

Access: Read,Write

Description: `AsBoolean` can be used to read or write the contents of the field as a currency value. If the native type of the field is not Boolean, then an attempt will be made to convert the field value from the native format to a currency value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefore, when reading or writing a field value for a field whose native data type is not a currency-compatible value (dates or string values), an exception may be raised.

See also: `TField.Value` ([439](#)), `TField.AsFloat` ([430](#))

**11.30.21 TField.AsDateTime**

Synopsis: Access the field's contents as a `TDateTime` value.

Declaration: `Property AsDateTime : TDateTime`

Visibility: `public`

Access: `Read,Write`

Description: `AsDateTime` can be used to read or write the contents of the field as a `TDateTime` value (for both date and time values). If the native type of the field is not a date or time value, then an attempt will be made to convert the field value from the native format to a `TDateTime` value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefor, when reading or writing a field value for a field whose native data type is not a `TDateTime`-compatible value (dates or string values), an exception may be raised.

See also: `TField.Value` ([439](#)), `TField.AsString` ([431](#))

**11.30.22 TField.AsFloat**

Synopsis: Access the field's contents as a floating-point (`Double`) value.

Declaration: `Property AsFloat : Double`

Visibility: `public`

Access: `Read,Write`

Description: `AsFloat` can be used to read or write the contents of the field as a floating-point value (of type `double`, i.e. with double precision). If the native type of the field is not a floating-point value, then an attempt will be made to convert the field value from the native format to a floating-point value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefor, when reading or writing a field value for a field whose native data type is not a floating-point-compatible value (string values for instance), an exception may be raised.

See also: `TField.Value` ([439](#)), `TField.AsString` ([431](#)), `TField.AsCurrency` ([429](#))

**11.30.23 TField.AsLongint**

Synopsis: Access the field's contents as a 32-bit signed integer (`longint`) value.

Declaration: `Property AsLongint : LongInt`

Visibility: `public`

Access: `Read,Write`

Description: `AsLongint` can be used to read or write the contents of the field as a 32-bit signed integer value (of type `longint`). If the native type of the field is not a `longint` value, then an attempt will be made to convert the field value from the native format to a `longint` value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefor, when reading or writing a field value for a field whose native data type is not a 32-bit signed integer-compatible value (string values for instance), an exception may be raised.

This is an alias for the `TField.AsInteger` ([431](#)).

See also: `TField.Value` ([439](#)), `TField.AsString` ([431](#)), `TField.AsInteger` ([431](#))

**11.30.24 TField.AsLargeInt**

Synopsis: Access the field's contents as a 64-bit signed integer (longint) value.

Declaration: `Property AsLargeInt : LargeInt`

Visibility: public

Access: Read,Write

Description: `AsLargeInt` can be used to read or write the contents of the field as a 64-bit signed integer value (of type `Int64`). If the native type of the field is not an `Int64` value, then an attempt will be made to convert the field value from the native format to an `Int64` value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefor, when reading or writing a field value for a field whose native data type is not a 64-bit signed integer-compatible value (string values for instance), an exception may be raised.

See also: `TField.Value` (439), `TField.AsString` (431), `TField.AsInteger` (431)

**11.30.25 TField.AsInteger**

Synopsis: Access the field's contents as a 32-bit signed integer (longint) value.

Declaration: `Property AsInteger : LongInt`

Visibility: public

Access: Read,Write

Description: `AsInteger` can be used to read or write the contents of the field as a 32-bit signed integer value (of type `Integer`). If the native type of the field is not an integer value, then an attempt will be made to convert the field value from the native format to a integer value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefor, when reading or writing a field value for a field whose native data type is not a 32-bit signed integer-compatible value (string values for instance), an exception may be raised.

See also: `TField.Value` (439), `TField.AsString` (431), `TField.AsLongint` (430), `TField.AsInt64` (421)

**11.30.26 TField.AsString**

Synopsis: Access the field's contents as an `AnsiString` value.

Declaration: `Property AsString : string`

Visibility: public

Access: Read,Write

Description: `AsString` can be used to read or write the contents of the field as an `AnsiString` value. If the native type of the field is not an `ansistring` value, then an attempt will be made to convert the field value from the native format to a `ansistring` value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefor, when reading or writing a field value for a field whose native data type is not an `ansistring`-compatible value, an exception may be raised.

See also: `TField.Value` (439), `TField.AsWideString` (433)



**11.30.27 TField.AsAnsiString**

Synopsis: Return field contents as an ANSI string

Declaration: `Property AsAnsiString : AnsiString`

Visibility: public

Access: Read,Write

Description: `AsAnsiString` returns the field data as an ANSI string (single byte character string). Note that if the field contains unicode data, some characters may get lost when reading.

See also: `TField.AsString` (431), `TField.AsUnicodeString` (432), `TField.AsUTF8String` (432), `TField.CodePage` (421)

**11.30.28 TField.AsUnicodeString**

Synopsis: Field contents as a UnicodeString

Declaration: `Property AsUnicodeString : UnicodeString`

Visibility: public

Access: Read,Write

Description: `AsUnicodeString` returns the field data as a Unicode string (double byte character string). If the field contains an `AnsiString`, the data will be converted to unicode according to the `CodePage` (421) when reading, and when writing the written data will be converted to single-byte string. Note that if the field is an `ansistring` field, some characters may get lost when writing.

See also: `TField.AsString` (431), `TField.AsAnsiString` (432), `TField.AsUTF8String` (432), `TField.CodePage` (421)

**11.30.29 TField.AsUTF8String**

Synopsis: Field contents as a UTF8 String

Declaration: `Property AsUTF8String : UTF8String`

Visibility: public

Access: Read,Write

Description: `AsUTF8String` returns the field data as a UTF8-Encoded string (single byte character string). If the field contains an `AnsiString`, the data will be converted to unicode according to the `CodePage` (421). If the field contains a unicode string, the string is UTF-8 encoded. When writing the written data will be converted to single-byte string. Note that if the field is an `ansistring` field, some characters may get lost when writing.

See also: `TField.AsString` (431), `TField.AsUnicodeString` (432), `TField.AsAnsi8String` (421), `TField.CodePage` (421)

### 11.30.30 TField.AsWideString

Synopsis: Access the field's contents as a WideString value.

Declaration: `Property AsWideString : WideString`

Visibility: `public`

Access: Read,Write

Description: `AsWideString` can be used to read or write the contents of the field as a WideString value. If the native type of the field is not a widestring value, then an attempt will be made to convert the field value from the native format to a widestring value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefor, when reading or writing a field value for a field whose native data type is not a widestring-compatible value, an exception may be raised.

See also: `TField.Value` ([439](#)), `TField.AString` ([421](#))

### 11.30.31 TField.AsVariant

Synopsis: Access the field's contents as a Variant value.

Declaration: `Property AsVariant : variant`

Visibility: `public`

Access: Read,Write

Description: `AsVariant` can be used to read or write the contents of the field as a Variant value. If the native type of the field is not a Variant value, then an attempt will be made to convert the field value from the native format to a variant value when reading the field's content. Likewise, when writing the property, the value will be converted to the native type of the field (if the value allows it). Therefor, when reading or writing a field value for a field whose native data type is not a variant-compatible value, an exception may be raised.

See also: `TField.Value` ([439](#)), `TField.AString` ([421](#))

### 11.30.32 TField.AttributeSet

Synopsis: Not used: dictionary information

Declaration: `Property AttributeSet : string`

Visibility: `public`

Access: Read,Write

Description: `AttributeSet` was used in older Delphi versions to store data dictionary information for use in data-aware controls at design time. Not used in FreePascal (or newer Delphi versions); kept for Delphi compatibility.

### 11.30.33 TField.Calculated

Synopsis: Is the field a calculated field ?

Declaration: `Property Calculated : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `Calculated` is `True` if the `FieldKind` ([442](#)) is `fkCalculated`. Setting the property will result in `FieldKind` being set to `fkCalculated` (for a value of `True`) or `fkData`. This property should be considered read-only.

See also: `TField.FieldKind` ([442](#))

### 11.30.34 TField.CanModify

Synopsis: Can the field's contents be modified.

Declaration: `Property CanModify : Boolean`

Visibility: `public`

Access: `Read`

Description: `CanModify` is `True` if the field is not read-only and the dataset allows modification.

See also: `TField.ReadOnly` ([445](#)), `TDataset.CanModify` ([393](#))

### 11.30.35 TField.CurValue

Synopsis: Current value of the field

Declaration: `Property CurValue : Variant`

Visibility: `public`

Access: `Read`

Description: `CurValue` returns the current value of the field as a variant.

See also: `TField.Value` ([439](#))

### 11.30.36 TField.DataSet

Synopsis: Dataset this field belongs to

Declaration: `Property DataSet : TDataset`

Visibility: `public`

Access: `Read,Write`

Description: `DataSet` contains the dataset this field belongs to. Writing this property will add the field to the list of fields of a dataset, after removing it from the list of fields of the dataset the field was previously assigned to. It should under normal circumstances never be necessary to set this property, the `TDataset` code will take care of this.

See also: `TDataset` ([371](#)), `TDataset.Fields` ([398](#))

### 11.30.37 TField.DataSize

Synopsis: Size of the field's data

Declaration: `Property DataSize : Integer`

Visibility: `public`

Access: `Read`

Description: `DataSource` is the memory size needed to store the field's contents. This is different from the `Size` (438) property which declares a logical size for datatypes that have a variable size (such as string fields). For BLOB fields, use the `TBlobField.BlobSize` (350) property to get the size of the field's contents for the current record..

See also: `TField.Size` (438), `TBlobField.BlobSize` (350)

### 11.30.38 TField.DataType

Synopsis: The data type of the field.

Declaration: `Property DataType : TFieldType`

Visibility: `public`

Access: `Read`

Description: `DataType` indicates the type of data the field has. This property is initialized when the dataset is opened or when persistent fields are created for the dataset. Instead of checking the class type of the field, it is better to check the `DataType`, since the actual class of the `TField` instance may differ depending on the dataset.

See also: `TField.FieldKind` (442)

### 11.30.39 TField.DisplayName

Synopsis: User-readable fieldname

Declaration: `Property DisplayName : string`

Visibility: `public`

Access: `Read`

Description: `DisplayName` is the name of the field as it will be displayed to the user e.g. in grid column headers. By default it equals the `FieldName` (442) property, unless assigned another value.

The use of this property is deprecated. Use `DisplayLabel` (441) instead.

See also: `TField.FieldName` (442)

### 11.30.40 TField.DisplayText

Synopsis: Formatted field value

Declaration: `Property DisplayText : string`

Visibility: `public`

Access: `Read`

**Description:** `DisplayText` returns the field's value as it should be displayed to the user, with all necessary formatting applied. Controls that should display the value of the field should use `DisplayText` instead of the `TField.AsString` (431) property, which does not take into account any formatting.

See also: `TField.AsString` (431)

#### 11.30.41 TField.EditMask

**Synopsis:** Specify an edit mask for an edit control

**Declaration:** `Property EditMask : TEditMask`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `EditMask` can be used to specify an edit mask for controls that allow to edit this field. It has no effect on the field value, and serves only to ensure that the user can enter only correct data for this field.

For more information on valid edit masks, see the documentation of the GUI controls.

See also: `TDateTimeField.EditMask` (415), `TStringField.EditMask` (506)

#### 11.30.42 TField.EditMaskPtr

**Synopsis:** Alias for `EditMask`

**Declaration:** `Property EditMaskPtr : TEditMask`

**Visibility:** `public`

**Access:** `Read`

**Description:** `EditMaskPtr` is a read-only alias for the `EditMask` (436) property. It is not used.

See also: `TField.EditMask` (436)

#### 11.30.43 TField.FieldNo

**Synopsis:** Number of the field in the record

**Declaration:** `Property FieldNo : LongInt`

**Visibility:** `public`

**Access:** `Read`

**Description:** `FieldNo` is the position of the field in the record. It is a 1-based index and is initialized when the dataset is opened or when persistent fields are created for the dataset.

See also: `TField.Index` (442)

#### 11.30.44 TField.IsIndexField

Synopsis: Is the field an indexed field ?

Declaration: `Property IsIndexField : Boolean`

Visibility: `public`

Access: `Read`

Description: `IsIndexField` is `true` if the field is an indexed field. By default this property is `False`, descendants of `TDataset` (371) can change this to `True`.

See also: `TField.Calculated` (434)

#### 11.30.45 TField.IsNull

Synopsis: Is the field empty

Declaration: `Property IsNull : Boolean`

Visibility: `public`

Access: `Read`

Description: `IsNull` is `True` if the field does not have a value. If the underlying data contained a value, or a value is written to it, `IsNull` will return `False`. After `TDataset.Insert` (386) is called or `Clear` (426) is called then `IsNull` will return `True`.

See also: `TField.Clear` (426), `TDataset.Insert` (386)

#### 11.30.46 TField.NewValue

Synopsis: The new value of the field

Declaration: `Property NewValue : Variant`

Visibility: `public`

Access: `Read,Write`

Description: `NewValue` returns the new value of the field. The FPC implementation of `TDataset` (371) does not yet support this.

See also: `TField.Value` (439), `TField.CurValue` (434)

#### 11.30.47 TField.Offset

Synopsis: Offset of the field's value in the dataset buffer

Declaration: `Property Offset : Word`

Visibility: `public`

Access: `Read`

Description: `Offset` is the location of the field's contents in the dataset memory buffer. It is read-only and initialized by the dataset when it is opened.

See also: `TField.FieldNo` (436), `TField.Index` (442), `TField.Datasize` (435)

**11.30.48 TField.Size**

Synopsis: Logical size of the field

Declaration: `Property Size : Integer`

Visibility: `public`

Access: Read,Write

Description: `Size` is the declared size of the field for datatypes that can have variable size, such as string types, BCD types or array types. To get the size of the storage needed to store the field's content, the `DataSize` (435) should be used. For blob fields, the current size of the data is not guaranteed to be present.

See also: `DataSize` (435)

**11.30.49 TField.Text**

Synopsis: Text representation of the field

Declaration: `Property Text : string`

Visibility: `public`

Access: Read,Write

Description: `Text` can be used to retrieve or set the value of the value as a string value for editing purposes. It will trigger the `TField.OnGetText` (446) event handler if a handler was specified. For display purposes, the `TField.DisplayText` (435) property should be used. Controls that should display the value in a textual format should use `Text` whenever they must display the text for editing purposes. Inversely, when a control should save the value entered by the user, it should write the contents to the `Text` property, not the `AsString` (431) property, this will invoke the `Tfield.OnSetText` (446) event handler, if one is set.

See also: `TField.AsString` (431), `TField.DisplayText` (435), `TField.Value` (439)

**11.30.50 TField.ValidChars**

Synopsis: Characters that are valid input for the field's content

Declaration: `Property ValidChars : TFieldChars`

Visibility: `public`

Access: Read,Write

Description: `ValidChars` is a property that is initialized by descendent classes to contain the set of characters that can be entered in an edit control which is used to edit the field. Numerical fields will set this to a set of numerical characters, string fields will set this to all possible characters. It is possible to restrict the possible input by setting this property to a subset of all possible characters (for example, set it to all uppercase letters to allow the user to enter only uppercase characters. `TField` itself does not enforce the validity of the data when the content of the field is set, an edit control should check the validity of the user input by means of the `IsValidChar` (427) function.

See also: `TField.IsValidChar` (427)

**11.30.51 TField.Value**

Synopsis: Value of the field as a variant value

Declaration: `Property Value : variant`

Visibility: public

Access: Read,Write

Description: `Value` can be used to read or write the value of the field as a Variant value. When setting the value, the value will be converted to the actual type of the field as defined in the underlying data. Likewise, when reading the value property, the actual field value will be converted to a variant value. If the field does not contain a value (when `IsNull` (437) returns `True`), then `Value` will contain `Null`.

It is not recommended to use the `Value` property: it should only be used when the type of the field is unknown. If the type of the field is known, it is better to use one of the `AsXXX` properties, which will not only result in faster code, but will also avoid strange type conversions.

See also: `TField.IsNull` (437), `TField.Text` (438), `TField.DisplayText` (435)

**11.30.52 TField.OldValue**

Synopsis: Old value of the field

Declaration: `Property OldValue : variant`

Visibility: public

Access: Read

Description: `OldValue` returns the value of the field prior to an edit operation. This feature is currently not supported in FPC.

See also: `TField.Value` (439), `TField.CurValue` (434), `TField.NewValue` (437)

**11.30.53 TField.LookupList**

Synopsis: List of lookup values

Declaration: `Property LookupList : TLookupList`

Visibility: public

Access: Read

Description: `LookupList` contains the list of key, value pairs used when caching the possible lookup values for a lookup field. The list is only valid when the `LookupCache` (443) property is set to `True`. It can be refreshed using the `RefreshLookupList` (427) method.

See also: `TField.RefreshLookupList` (427), `TField.LookupCache` (443)

**11.30.54 TField.FieldDef**

Synopsis: `FieldDef` associated with this field

Declaration: `Property FieldDef : TFieldDef`

Visibility: public



Access: Read

Description: `FieldDef` references the `TFieldDef` instance to which this field instance is bound. When a dataset is opened, the `TDataset.FieldDefs` (395) property is filled with field definitions as returned from the server. After this fields are created, or if they already exist, are bound to these fielddefs.

See also: `TDataset.FieldDefs` (395)

### 11.30.55 TField.Alignment

Synopsis: Alignment for this field

Declaration: Property `Alignment` : `TAlignment`

Visibility: published

Access: Read,Write

Description: `Alignment` contains the alignment that UI controls should observe when displaying the contents of the field. Setting the property at the field level will make sure that all DB-Aware controls will display the contents of the field with the same alignment.

See also: `TField.DisplayText` (435)

### 11.30.56 TField.CustomConstraint

Synopsis: Custom constraint for the field's value

Declaration: Property `CustomConstraint` : `string`

Visibility: published

Access: Read,Write

Description: `CustomConstraint` may contain a constraint that will be enforced when the dataset posts it's data. It should be a SQL-like expression that results in a `True` or `False` value. Examples of valid constraints are:

```
Salary < 10000
YearsEducation < Age
```

If the constraint is not satisfied when the record is posted, then an exception will be raised with the value of `ConstraintErrorMessage` (440) as a message.

This feature is not yet implemented in FPC.

See also: `TField.ConstraintErrorMessage` (440), `TField.ImportedConstraint` (443)

### 11.30.57 TField.ConstraintErrorMessage

Synopsis: Message to display if the `CustomConstraint` constraint is violated.

Declaration: Property `ConstraintErrorMessage` : `string`

Visibility: published

Access: Read,Write

**Description:** `ConstraintErrorMessage` is the message that should be displayed when the dataset checks the constraints and the constraint in `TField.CustomConstraint` (440) is violated.

This feature is not yet implemented in FPC.

See also: `TField.CustomConstraint` (440)

### 11.30.58 TField.DefaultExpression

**Synopsis:** Default value for the field

**Declaration:** `Property DefaultExpression : string`

**Visibility:** published

**Access:** Read,Write

**Description:** `DefaultValue` can be set to a value that should be entered in the field whenever the `TDataset.Append` (377) or `TDataset.Insert` (386) methods are executed. It should contain a valid SQL expression that results in the correct type for the field.

This feature is not yet implemented in FPC.

See also: `TDataset.Insert` (386), `TDataset.Append` (377), `TDataset.CustomConstraint` (371)

### 11.30.59 TField.DisplayLabel

**Synopsis:** Name of the field for display purposes

**Declaration:** `Property DisplayLabel : string`

**Visibility:** published

**Access:** Read,Write

**Description:** `DisplayLabel` is the name of the field as it will be displayed to the user e.g. in grid column headers. By default it equals the `FieldName` (442) property, unless assigned another value.

See also: `TField.FieldName` (442)

### 11.30.60 TField.DisplayWidth

**Synopsis:** Width of the field in characters

**Declaration:** `Property DisplayWidth : LongInt`

**Visibility:** published

**Access:** Read,Write

**Description:** `DisplayWidth` is the width (in characters) that should be used by controls that display the contents of the field (such as in grids or lookup lists). It is initialized to a default value for most fields (e.g. it equals `Size` (438) for string fields) but can be modified to obtain a more appropriate value for the field's expected content.

See also: `TField.Alignment` (440), `TField.DisplayText` (435)

**11.30.61 TField.FieldKind**

Synopsis: The kind of field.

Declaration: `Property FieldKind : TFieldKind`

Visibility: published

Access: Read,Write

Description: `FieldKind` indicates the type of the `TField` instance. Besides `TField` instances that represent fields present in the underlying data records, there can also be calculated or lookup fields. This property determines what kind of field the `TField` instance is.

**11.30.62 TField.FieldName**

Synopsis: Name of the field

Declaration: `Property FieldName : string`

Visibility: published

Access: Read,Write

Description: `FieldName` is the name of the field as it is defined in the underlying data structures (for instance the name of the field in a SQL table, DBase file, or the alias of the field if it was aliased in a SQL SELECT statement. It does not always equal the `Name` property, which is the name of the `TField` component instance. The `Name` property will generally equal the name of the dataset appended with the value of the `FieldName` property.

See also: `TFieldDef.Name` ([447](#)), `TField.Size` ([438](#)), `TField.DataType` ([435](#))

**11.30.63 TField.HasConstraints**

Synopsis: Does the field have any constraints defined

Declaration: `Property HasConstraints : Boolean`

Visibility: published

Access: Read

Description: `HasConstraints` will contain `True` if one of the `CustomConstraint` ([440](#)) or `ImportedConstraint` ([443](#)) properties is set to a non-empty value.

See also: `CustomConstraint` ([440](#)), `ImportedConstraint` ([443](#))

**11.30.64 TField.Index**

Synopsis: Index of the field in the list of fields

Declaration: `Property Index : LongInt`

Visibility: published

Access: Read,Write

Description: `Index` is the name of the field in the list of fields of a dataset. It is, in general, the (0-based) position of the field in the underlying data structures, but this need not always be so. The `TField.FieldNo` ([436](#)) property should be used for that.

See also: `TField.FieldNo` ([436](#))

**11.30.65 TField.ImportedConstraint**

Synopsis: Constraint for the field value on the level of the underlying database

Declaration: `Property ImportedConstraint : string`

Visibility: published

Access: Read,Write

Description: `ImportedConstraint` contains any constraints that the underlying data engine imposes on the values of a field (usually in an SQL CONSTRAINT) clause. Whether this field is filled with appropriate data depends on the implementation of the `TDataset` (371) descendent.

See also: `TField.CustomConstraint` (440), `TDataset` (371), `TField.ConstraintErrorMessage` (440)

**11.30.66 TField.KeyFields**

Synopsis: Key fields to use when looking up a field value.

Declaration: `Property KeyFields : string`

Visibility: published

Access: Read,Write

Description: `KeyFields` should contain a semi-colon separated list of field names from the lookupfield's dataset which will be matched to the fields enumerated in `LookupKeyFields` (444) in the dataset pointed to by the `LookupDataset` (443) property.

See also: `LookupKeyFields` (444), `LookupDataset` (443)

**11.30.67 TField.LookupCache**

Synopsis: Should lookup values be cached

Declaration: `Property LookupCache : Boolean`

Visibility: published

Access: Read,Write

Description: `LookupCache` is by default `False`. If it is set to `True` then a list of key, value pairs will be created from the `LookupKeyFields` (444) in the dataset pointed to by the `LookupDataset` (443) property. The list of key, value pairs is available through the `TField.LookupList` (439) property.

See also: `LookupKeyFields` (444), `LookupDataset` (443), `TField.LookupList` (439)

**11.30.68 TField.LookupDataSet**

Synopsis: Dataset with lookup values

Declaration: `Property LookupDataSet : TDataSet`

Visibility: published

Access: Read,Write

Description: `LookupDataset` is used by lookup fields to fetch the field's value. The `LookupKeyFields` (444) property is used as a list of fields to locate a record in this dataset, and the value of the `LookupResultField` (444) field is then used as the value of the lookup field.

See also: `KeyFields` (443), `LookupKeyFields` (444), `LookupResultField` (444), `LookupCache` (443)

**11.30.69 TField.LookupKeyFields**

Synopsis: Names of fields on which to perform a locate

Declaration: `Property LookupKeyFields : string`

Visibility: published

Access: Read,Write

Description: `LookupKeyFields` should contain a semi-colon separated list of field names from the dataset pointed to by the `LookupDataset` (443) property. These fields will be used when locating a record corresponding to the values in the `TField.KeyFields` (443) property.

See also: `KeyFields` (443), `LookupDataset` (443), `LookupResultField` (444), `LookupCache` (443)

**11.30.70 TField.LookupResultField**

Synopsis: Name of field to use as lookup value

Declaration: `Property LookupResultField : string`

Visibility: published

Access: Read,Write

Description: `LookupResultField` contains the field name from a field in the dataset pointed to by the `LookupDataset` (443) property. The value of this field will be used as the lookup's field value when a record is found in the lookup dataset as result for the lookup field value.

See also: `KeyFields` (443), `LookupDataset` (443), `LookupKeyFields` (444), `LookupCache` (443)

**11.30.71 TField.Lookup**

Synopsis: Is the field a lookup field

Declaration: `Property Lookup : Boolean; deprecated;`

Visibility: published

Access: Read,Write

Description: `Lookup` is `True` if the `FieldKind` (442) equals `fkLookup`, `False` otherwise. Setting the `Lookup` property will switch the `FieldKind` between the `fkLookup` and `fkData`.

See also: `TField.FieldKind` (442)

**11.30.72 TField.Origin**

Synopsis: Original fieldname of the field.

Declaration: `Property Origin : string`

Visibility: published

Access: Read,Write

Description: `Origin` contains the origin of the field in the form `TableName.fieldName`. This property is filled only if the `TDataset` (371) descendent or the database engine support retrieval of this property. It can be used to automatically create update statements, together with the `TField.ProviderFlags` (445) property.

See also: `TDataset` (371), `TField.ProviderFlags` (445)

**11.30.73 TField.ProviderFlags**

Synopsis: Flags for provider or update support

Declaration: `Property ProviderFlags : TProviderFlags`

Visibility: published

Access: Read,Write

Description: `ProviderFlags` contains a set of flags that can be used by engines that automatically generate update SQL statements or update data packets. The various items in the set tell the engine whether the key is a key field, should be used in the where clause of an update statement or whether - in fact - it should be updated at all.

These properties should be set by the programmer so engines such as SQLDB can create correct update SQL statements whenever they need to post changes to the database. Note that to be able to set these properties in a designer, persistent fields must be created.

See also: `TField.Origin` ([444](#))

**11.30.74 TField.ReadOnly**

Synopsis: Is the field read-only

Declaration: `Property ReadOnly : Boolean`

Visibility: published

Access: Read,Write

Description: `ReadOnly` can be set to `True` to prevent controls of writing data to the field, effectively making it a read-only field. Setting this property to `True` does not prevent the field from getting a value through code: it is just an indication for GUI controls that the field's value is considered read-only.

See also: `TFieldDef.Attributes` ([450](#))

**11.30.75 TField.Required**

Synopsis: Does the field require a value

Declaration: `Property Required : Boolean`

Visibility: published

Access: Read,Write

Description: `Required` determines whether the field needs a value when posting the data: when a dataset posts the changed made to a record (new or existing), it will check whether all fields with the `Required` property have a value assigned to them. If not, an exception will be raised. Descendents of `TDataset` ([371](#)) will set the property to `True` when opening the dataset, depending on whether the field is required in the underlying data engine. For fields that are not required by the database engine, the programmer can still set the property to `True` if the business logic requires a field.

See also: `TDataset.Open` ([389](#)), `ReadOnly` ([445](#)), `Visible` ([446](#))

### 11.30.76 TField.Visible

Synopsis: Should the field be shown in grids

Declaration: `Property Visible : Boolean`

Visibility: published

Access: Read,Write

Description: `Visible` can be used to hide fields from a grid when displaying data to the user. Invisible fields will by default not be shown in the grid.

See also: `TField.ReadOnly` (445), `TField.Required` (445)

### 11.30.77 TField.OnChange

Synopsis: Event triggered when the field's value has changed

Declaration: `Property OnChange : TFieldNotifyEvent`

Visibility: published

Access: Read,Write

Description: `OnChange` is triggered whenever the field's value has been changed. It is triggered only after the new contents have been written to the dataset buffer, so it can be used to react to changes in the field's content. To prevent the writing of changes to the buffer, use the `TField.OnValidate` (447) event. It is not allowed to change the state of the dataset or the contents of the field during the execution of this event handler: doing so may lead to infinite loops and other unexpected results.

See also: `TField.OnChange` (446)

### 11.30.78 TField.OnGetText

Synopsis: Event to format the field's content

Declaration: `Property OnGetText : TFieldGetTextEvent`

Visibility: published

Access: Read,Write

Description: `OnGetText` is triggered whenever the `TField.Text` (438) or `TField.DisplayText` (435) properties are read. It can be used to return a custom formatted string in the `AText` parameter which will then typically be used by a control to display the field's contents to the user. It is not allowed to change the state of the dataset or the contents of the field during the execution of this event handler.

See also: `TField.Text` (438), `TField.DisplayText` (435), `TField.OnSetText` (446), `TFieldGetTextEvent` (323)

### 11.30.79 TField.OnSetText

Synopsis: Event to set the field's content based on a user-formatted string

Declaration: `Property OnSetText : TFieldSetTextEvent`

Visibility: published

Access: Read,Write

**Description:** `OnSetText` is called whenever the `TField.Text` (438) property is written. It can be used to set the actual value of the field based on the passed `AText` parameter. Typically, this event handler will perform the inverse operation of the `TField.OnGetText` (446) handler, if it exists.

See also: `TField.Text` (438), `TField.OnGetText` (446), `TField.GetTextEvent` (323)

### 11.30.80 TField.OnValidate

**Synopsis:** Event to validate the value of a field before it is written to the data buffer

**Declaration:** `Property OnValidate : TFieldNotifyEvent`

**Visibility:** published

**Access:** Read,Write

**Description:** `OnValidate` is called prior to writing a new field value to the dataset's data buffer. It can be used to prevent writing the new value to the buffer by raising an exception in the event handler. Note that this event handler is always called, irrespective of the way the value of the field is set.

See also: `TField.Text` (438), `TField.OnGetText` (446), `TField.OnSetText` (446), `TField.OnChange` (446)

## 11.31 TFieldDef

### 11.31.1 Description

`TFieldDef` is used to describe the fields that are present in the data underlying the dataset. For each field in the underlying field, an `TFieldDef` instance is created when the dataset is opened. This class offers almost no methods, it is mainly a storage class, to store all relevant properties of fields in a record (name, data type, size, required or not, etc.)

See also: `TDataset.FieldDefs` (395), `TFieldDefs` (451)

### 11.31.2 Method overview

Page	Method	Description
448	Assign	Assign the contents of one <code>TFieldDef</code> instance to another.
448	Create	Constructor for <code>TFieldDef</code> .
448	CreateField	Create <code>TField</code> instance based on definitions in current <code>TFieldDef</code> instance.
448	Destroy	Free the <code>TFieldDef</code> instance

### 11.31.3 Property overview

Page	Properties	Access	Description
450	Attributes	rw	Additional attributes of the field.
449	CharSize	r	Character size
450	Codepage	r	
451	DataType	rw	Data type for the field
449	FieldClass	r	<code>TField</code> class used for this fielddef
449	FieldNo	r	Field number
450	InternalCalcField	rw	Is this a definition of an internally calculated field ?
451	Precision	rw	Precision used in BCD (Binary Coded Decimal) fields
450	Required	rw	Is the field required ?
451	Size	rw	Size of the buffer needed to store the data of the field



### 11.31.4 TFieldDef.Create

Synopsis: Constructor for TFieldDef.

Declaration: `constructor Create(ACollection: TCollection); Override`  
`constructor Create(AOwner: TFieldDefs; const AName: string;`  
`ADataType: TFieldType; ASize: Integer;`  
`ARequired: Boolean; AFieldNo: LongInt;`  
`ACodePage: TSystemCodePage); Overload`

Visibility: public

Description: `Create` is the constructor for the TFieldDef class.

If a simple call is used, with a single argument `ACollection`, the inherited `Create` is called and the Field number is set to the incremented current index.

If the more complicated call is used, with multiple arguments, then after the inherited `Create` call, the Name (447), datatype (451), size (451), precision (451), FieldNo (449), Required (450) and CodePage (450) property are all set according to the passed arguments.

Errors: If a duplicate name is passed, then an exception will occur.

See also: Name (447), datatype (451), size (451), precision (451), FieldNo (449), Required (450), CodePage (450)

### 11.31.5 TFieldDef.Destroy

Synopsis: Free the TFieldDef instance

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` destroys the TFieldDef instance. It simply calls the inherited destructor.

See also: TFieldDef.Create (448)

### 11.31.6 TFieldDef.Assign

Synopsis: Assign the contents of one TFieldDef instance to another.

Declaration: `procedure Assign(APersistent: TPersistent); Override`

Visibility: public

Description: `Assign` assigns all published properties of `APersistent` to the current instance, if `APersistent` is an instance of class TFieldDef.

Errors: If `APersistent` is not of class TFieldDef (447), then an exception will be raised.

### 11.31.7 TFieldDef.CreateField

Synopsis: Create TField instance based on definitions in current TFieldDef instance.

Declaration: `function CreateField(AOwner: TComponent) : TField`

Visibility: public

**Description:** `CreateField` determines, based on the `DataType` (451) what `TField` (421) descendent it should create, and then returns a newly created instance of this class. It sets the appropriate defaults for the `Size` (438), `FieldName` (442), `FieldNo` (436), `Precision` (421), `ReadOnly` (445) and `Required` (445) properties of the newly created instance. It should never be necessary to use this call in an end-user program, only `TDataset` descendent classes should use this call.

The newly created field is owned by the component instance passed in the `AOwner` parameter.

The `DefaultFieldClasses` (316) array is used to determine which `TField` Descendent class should be used when creating the `TField` instance, but descendents of `TDataset` may override the values in that array.

See also: `DefaultFieldClasses` (316), `TField` (421)

### 11.31.8 TFieldDef.FieldClass

**Synopsis:** `TField` class used for this fielddef

**Declaration:** `Property FieldClass : TFieldClass`

**Visibility:** public

**Access:** Read

**Description:** `FieldClass` is the class of the `TField` instance that is created by the `CreateField` (448) class. The return value is retrieved from the `TDataset` instance the `TFieldDef` instance is associated with. If there is no `TDataset` instance available, the return value is `Nil`.

See also: `TDataset` (371), `CreateField` (448), `TField` (421)

### 11.31.9 TFieldDef.FieldNo

**Synopsis:** Field number

**Declaration:** `Property FieldNo : LongInt`

**Visibility:** public

**Access:** Read

**Description:** `FieldNo` is the number of the field in the data structure where the dataset contents comes from, for instance in a DBase file. If the underlying data layer does not support the concept of field number, a sequential number is assigned.

### 11.31.10 TFieldDef.CharSize

**Synopsis:** Character size

**Declaration:** `Property CharSize : Word`

**Visibility:** public

**Access:** Read

**Description:** `CharSize` is only relevant for string fields: it indicates the number of bytes used to represent a single character. It is calculated from the `TFieldDef.CodePage` (450) property and can have the following values:

- 1 for single-byte string fields

- 2 for UnicodeString fields
- 4 for UTF8 strings

See also: `TFieldDef.CodePage` ([450](#)), `TFieldDef.Size` ([451](#))

### 11.31.11 TFieldDef.InternalCalcField

Synopsis: Is this a definition of an internally calculated field ?

Declaration: `Property InternalCalcField : Boolean`

Visibility: public

Access: Read,Write

Description: `InternalCalc` is `True` if the `fielddef` instance represents an internally calculated field: for internally calculated fields, storage must be provided by the underlying data mechanism.

### 11.31.12 TFieldDef.Required

Synopsis: Is the field required ?

Declaration: `Property Required : Boolean`

Visibility: public

Access: Read,Write

Description: `Required` is set to `True` if the field requires a value when posting data to the dataset. If no value was entered, the dataset will raise an exception when the record is posted. The `Required` property is usually initialized based on the definition of the field in the underlying database. For SQL-based databases, a field declared as `NOT NULL` will result in a `Required` property of `True`.

### 11.31.13 TFieldDef.Codepage

Declaration: `Property Codepage : TSystemCodePage`

Visibility: public

Access: Read

### 11.31.14 TFieldDef.Attributes

Synopsis: Additional attributes of the field.

Declaration: `Property Attributes : TFieldAttributes`

Visibility: published

Access: Read,Write

Description: `Attributes` contain additional attributes of the field. It shares the `faRequired` attribute with the `Required` property.

See also: `TFieldDef.Required` ([450](#))

### 11.31.15 TFieldDef.DataType

Synopsis: Data type for the field

Declaration: `Property DataType : TFieldType`

Visibility: published

Access: Read,Write

Description: `DataType` contains the data type of the field's contents. Based on this property, the `FieldClass` property determines what kind of field class must be used to represent this field.

See also: `TFieldDef.FieldClass` ([449](#)), `TFieldDef.CreateField` ([448](#))

### 11.31.16 TFieldDef.Precision

Synopsis: Precision used in BCD (Binary Coded Decimal) fields

Declaration: `Property Precision : LongInt`

Visibility: published

Access: Read,Write

Description: `Precision` is the number of digits used in a BCD (Binary Coded Decimal) field. It is not the number of digits after the decimal separator, but the total number of digits.

See also: `TFieldDef.Size` ([451](#))

### 11.31.17 TFieldDef.Size

Synopsis: Size of the buffer needed to store the data of the field

Declaration: `Property Size : Integer`

Visibility: published

Access: Read,Write

Description: `Size` indicates the size of the buffer needed to hold data for the field. For types with a fixed size (such as integer, word or data/time) the size can be zero: the buffer mechanism reserves automatically enough heap memory. For types which can have various sizes (blobs, string types), the `Size` property tells the buffer mechanism how many bytes are needed to hold the data for the field. For BCD fields, the size property indicates the number of decimals after the decimal separator.

See also: `TFieldDef.Precision` ([451](#)), `TFieldDef.DataType` ([451](#))

## 11.32 TFieldDefs

### 11.32.1 Description

`TFieldDefs` is used by each `TDataset` instance to keep a description of the data that it manages; for each field in a record that makes up the underlying data, the `TFieldDefs` instance keeps an instance of `TFieldDef` that describes the field's contents. For any internally calculated fields of the dataset, a `TFieldDef` instance is kept as well. This collection is filled by descendent classes of `TDataset` as soon as the dataset is opened; it is cleared when the dataset closes. After the

collection was populated, the dataset creates `TField` instances based on all the definitions in the collections. If persistent fields were used, the contents of the `fielddefs` collection is compared to the field components that are present in the dataset. If the collection contains more field definitions than Field components, these extra fields will not be available in the dataset.

See also: `TFieldDef` (447), `TDataset` (371)

### 11.32.2 Method overview

Page	Method	Description
<a href="#">452</a>	<code>Add</code>	Add a new field definition to the collection.
<a href="#">453</a>	<code>AddFieldDef</code>	Add new <code>TFieldDef</code>
<a href="#">453</a>	<code>Assign</code>	Copy all items from one dataset to another
<a href="#">452</a>	<code>Create</code>	Create a new instance of <code>TFieldDefs</code>
<a href="#">453</a>	<code>Find</code>	Find item by name
<a href="#">454</a>	<code>MakeNameUnique</code>	Create a unique field name starting from a base name
<a href="#">453</a>	<code>Update</code>	Force update of definitions

### 11.32.3 Property overview

Page	Properties	Access	Description
<a href="#">454</a>	<code>HiddenFields</code>	rw	Should field instances be created for hidden fields
<a href="#">454</a>	<code>Items</code>	rw	Indexed access to the <code>fielddef</code> instances

### 11.32.4 TFieldDefs.Create

Synopsis: Create a new instance of `TFieldDefs`

Declaration: `constructor Create(ADataset: TDataset)`

Visibility: `public`

Description: `Create` is used to create a new instance of `TFieldDefs`. The `ADataset` argument contains the dataset instance for which the collection contains the field definitions.

See also: `TFieldDef` (447), `TDataset` (371)

### 11.32.5 TFieldDefs.Add

Synopsis: Add a new field definition to the collection.

Declaration: `function Add(const AName: string; ADataType: TFieldType; ASize: Integer; APrecision: Integer; ARequired: Boolean; AReadOnly: Boolean; AFieldNo: Integer; ACodePage: TSystemCodePage) : TFieldDef; Overload`  
`function Add(const AName: string; ADataType: TFieldType; ASize: Word; ARequired: Boolean; AFieldNo: Integer) : TFieldDef; Overload`  
`procedure Add(const AName: string; ADataType: TFieldType; ASize: Word; ARequired: Boolean); Overload`  
`procedure Add(const AName: string; ADataType: TFieldType; ASize: Word); Overload`  
`procedure Add(const AName: string; ADataType: TFieldType); Overload`

Visibility: public

Description: Add adds a new item to the collection and fills in the Name, DataType, Size and Required properties of the newly added item with the provided parameters.

Errors: If an item with name AName already exists in the collection, then an exception will be raised.

See also: TFieldDefs.AddFieldDef ([453](#))

### 11.32.6 TFieldDefs.AddFieldDef

Synopsis: Add new TFieldDef

Declaration: `function AddFieldDef : TFieldDef`

Visibility: public

Description: AddFieldDef creates a new TFieldDef item and returns the instance.

See also: TFieldDefs.Add ([452](#))

### 11.32.7 TFieldDefs.Assign

Synopsis: Assign all items from one dataset to another

Declaration: `procedure Assign(FieldDefs: TFieldDefs); Overload`

Visibility: public

Description: Assign simply calls inherited Assign with the FieldDefs argument.

See also: TFieldDef.Assign ([448](#))

### 11.32.8 TFieldDefs.Find

Synopsis: Find item by name

Declaration: `function Find(const AName: string) : TFieldDef`

Visibility: public

Description: Find simply calls the inherited TDefCollection.Find ([419](#)) to find an item with name AName and typecasts the result to TFieldDef.

See also: TDefCollection.Find ([419](#)), TNamedItem.Name ([483](#))

### 11.32.9 TFieldDefs.Update

Synopsis: Force update of definitions

Declaration: `procedure Update; Overload`

Visibility: public

Description: Update notifies the dataset that the field definitions are updated, if it was not yet notified.

See also: TDefCollection.Updated ([420](#))

### 11.32.10 TFieldDefs.MakeNameUnique

Synopsis: Create a unique field name starting from a base name

Declaration: `function MakeNameUnique(const AName: string) : string; Virtual`

Visibility: public

Description: `MakeNameUnique` uses `AName` to construct a name of a field that is not yet in the collection. If `AName` is not yet in the collection, then `AName` is returned. If a field definition with field name equal to `AName` already exists, then a new name is constructed by appending a sequence number to `AName` till the resulting name does not appear in the list of field definitions.

See also: `TFieldDefs.Find` ([453](#)), `TFieldDef.Name` ([447](#))

### 11.32.11 TFieldDefs.HiddenFields

Synopsis: Should field instances be created for hidden fields

Declaration: `Property HiddenFields : Boolean`

Visibility: public

Access: Read,Write

Description: `HiddenFields` determines whether a field is created for fielddefs that have the `faHiddenCol` attribute set. If set to `False` (the default) then no `TField` instances will be created for hidden fields. If it is set to `True`, then a `TField` instance will be created for hidden fields.

See also: `TFieldDef.Attributes` ([450](#))

### 11.32.12 TFieldDefs.Items

Synopsis: Indexed access to the fielddef instances

Declaration: `Property Items[Index: LongInt]: TFieldDef; default`

Visibility: public

Access: Read,Write

Description: `Items` provides zero-based indexed access to all `TFieldDef` instances in the collection. The index must vary between 0 and `Count-1`, or an exception will be raised.

See also: `TFieldDef` ([447](#))

## 11.33 TFields

### 11.33.1 Description

`TFields` mimics a `TCollection` class for the `Fields` ([398](#)) property of `TDataset` ([371](#)) instance. Since `TField` ([421](#)) is a descendent of `TComponent`, it cannot be an item of a collection, and must be managed by another class.

See also: `TField` ([421](#)), `TDataset` ([371](#)), `TDataset.Fields` ([398](#))

### 11.33.2 Method overview

Page	Method	Description
<a href="#">455</a>	Add	Add a new field to the list
<a href="#">456</a>	CheckFieldName	Check field name for duplicate entries
<a href="#">456</a>	CheckFieldNames	Check a list of field names for duplicate entries
<a href="#">456</a>	Clear	Clear the list of fields
<a href="#">455</a>	Create	Create a new instance of TFields
<a href="#">455</a>	Destroy	Free the TFields instance
<a href="#">457</a>	FieldByName	Find a field based on its name
<a href="#">457</a>	FieldByNumber	Search field based on its fieldnumber
<a href="#">456</a>	FindField	Find a field based on its name
<a href="#">457</a>	GetEnumerator	Return an enumerator for the <code>for..in</code> construct
<a href="#">457</a>	GetFieldNames	Get the list of fieldnames
<a href="#">458</a>	IndexOf	Return the index of a field instance
<a href="#">458</a>	Remove	Remove an instance from the list

### 11.33.3 Property overview

Page	Properties	Access	Description
<a href="#">458</a>	Count	r	Number of fields in the list
<a href="#">458</a>	Dataset	r	Dataset the fields belong to
<a href="#">459</a>	Fields	rw	Indexed access to the fields in the list

### 11.33.4 TFields.Create

Synopsis: Create a new instance of TFields

Declaration: `constructor Create(ADataset: TDataSet)`

Visibility: `public`

Description: `Create` initializes a new instance of TFields. It stores the `ADataset` parameter, so it can be retrieved at any time in the `TFields.Dataset` ([458](#)) property, and initializes an internal list object to store the list of fields.

See also: `TDataSet` ([371](#)), `TFields.Dataset` ([458](#)), `TField` ([421](#))

### 11.33.5 TFields.Destroy

Synopsis: Free the TFields instance

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` frees the field instances that it manages on behalf of the `Dataset` ([458](#)). After that it cleans up the internal structures and then calls the inherited destructor.

See also: `TDataSet` ([371](#)), `TField` ([421](#)), `TFields.Clear` ([456](#))

### 11.33.6 TFields.Add

Synopsis: Add a new field to the list

Declaration: `procedure Add(Field: TField)`



Visibility: public

Description: Add must be used to add a new TField (421) instance to the list of fields. After a TField instance is added to the list, the TFields instance will free the field instance if it is cleared.

See also: TField (421), TFields.Clear (456)

### 11.33.7 TFields.CheckFieldName

Synopsis: Check field name for duplicate entries

Declaration: `procedure CheckFieldName(const Value: string)`

Visibility: public

Description: CheckFieldName checks whether a field with name equal to Value (case insensitive) already appears in the list of fields (using TFields.Find (454)). If it does, then an EDatabaseError (335) exception is raised.

See also: TField.FieldName (442), TFields.Find (454)

### 11.33.8 TFields.CheckFieldNames

Synopsis: Check a list of field names for duplicate entries

Declaration: `procedure CheckFieldNames(const Value: string)`

Visibility: public

Description: CheckFieldNames splits Value in a list of fieldnames, using semicolon as a separator. For each of the fieldnames obtained in this way, it calls CheckFieldName (456).

Errors: Spaces are not discarded, so leaving a space after or before a fieldname will not find the fieldname, and will yield a false negative result.

See also: TField.FieldName (442), TFields.CheckFieldName (456), TFields.Find (454)

### 11.33.9 TFields.Clear

Synopsis: Clear the list of fields

Declaration: `procedure Clear`

Visibility: public

Description: Clear removes all TField (421) var instances from the list. All field instances are freed after they have been removed from the list.

See also: TField (421)

### 11.33.10 TFields.FindField

Synopsis: Find a field based on its name

Declaration: `function FindField(const Value: string) : TField`

Visibility: public

**Description:** `FindField` searches the list of fields and returns the field instance whose `FieldName` (442) property matches `Value`. The search is performed case-insensitively. If no field instance is found, then `Nil` is returned.

See also: `TFields.FieldByName` (457)

### 11.33.11 TFields.FieldByName

**Synopsis:** Find a field based on its name

**Declaration:** `function FieldByName(const Value: string) : TField`

**Visibility:** public

**Description:** `Fieldbyname` searches the list of fields and returns the field instance whose `FieldName` (442) property matches `Value`. The search is performed case-insensitively.

**Errors:** If no field instance is found, then an exception is raised. If this behaviour is undesired, use `TField.FindField` (421), where `Nil` is returned if no match is found.

See also: `TFields.FindField` (456), `TFields.FieldName` (454), `TFields.FieldByNumber` (457), `TFields.IndexOf` (458)

### 11.33.12 TFields.FieldByNumber

**Synopsis:** Search field based on its fieldnumber

**Declaration:** `function FieldByNumber(FieldNo: Integer) : TField`

**Visibility:** public

**Description:** `FieldByNumber` searches for the field whose `TField.FieldNo` (436) property matches the `FieldNo` parameter. If no such field is found, `Nil` is returned.

See also: `TFields.FieldByName` (457), `TFields.FindField` (456), `TFields.IndexOf` (458)

### 11.33.13 TFields.GetEnumerator

**Synopsis:** Return an enumerator for the `for..in` construct

**Declaration:** `function GetEnumerator : TFieldsEnumerator`

**Visibility:** public

**Description:** `GetEnumerator` is the implementation of `IEnumerable` and returns an instance of `TFieldsEnumerator` (459)

See also: `TFieldsEnumerator` (459), `#rtl.system.IEnumerable` (??)

### 11.33.14 TFields.GetFieldNames

**Synopsis:** Get the list of fieldnames

**Declaration:** `procedure GetFieldNames(Values: TStrings)`

**Visibility:** public

**Description:** `GetFieldNames` fills `Values` with the fieldnames of all the fields in the list, each item in the list contains 1 fieldname. The list is cleared prior to filling it.

See also: `TField.FieldName` (442)

### 11.33.15 TFields.IndexOf

Synopsis: Return the index of a field instance

Declaration: `function IndexOf(Field: TField) : LongInt`

Visibility: public

Description: `IndexOf` scans the list of fields and returns the index of the field instance in the list (it compares actual field instances, not field names). If the field does not appear in the list, -1 is returned.

See also: `TFields.FieldName` ([457](#)), `TFields.FieldByNumber` ([457](#)), `TFields.FindField` ([456](#))

### 11.33.16 TFields.Remove

Synopsis: Remove an instance from the list

Declaration: `procedure Remove(Value: TField)`

Visibility: public

Description: `Remove` removes the field `Value` from the list. It does not free the field after it was removed. If the field is not in the list, then nothing happens.

See also: `TFields.Clear` ([456](#))

### 11.33.17 TFields.Count

Synopsis: Number of fields in the list

Declaration: `Property Count : Integer`

Visibility: public

Access: Read

Description: `Count` is the number of fields in the fieldlist. The items in the `Fields` ([459](#)) property are numbered from 0 to `Count-1`.

See also: `TFields.fields` ([459](#))

### 11.33.18 TFields.Dataset

Synopsis: Dataset the fields belong to

Declaration: `Property Dataset : TDataSet`

Visibility: public

Access: Read

Description: `Dataset` is the dataset instance that owns the fieldlist. It is set when the `TFields` ([454](#)) instance is created. This property is purely for informational purposes. When adding fields to the list, no check is performed whether the field's `Dataset` property matches this dataset.

See also: `TFields.Create` ([455](#)), `TField.Dataset` ([434](#)), `TDataSet` ([371](#))

### 11.33.19 TFields.Fields

Synopsis: Indexed access to the fields in the list

Declaration: `Property Fields[Index: Integer]: TField; default`

Visibility: `public`

Access: `Read, Write`

Description: `Fields` is the default property of the `TFields` class. It provides indexed access to the fields in the list: the index runs from 0 to `Count-1`.

Errors: Providing an index outside the allowed range will result in an `EListError` exception.

See also: `TFields.FieldName` (457)

## 11.34 TFieldsEnumerator

### 11.34.1 Description

`TFieldsEnumerator` implements all the methods of `IEnumerator` so a `TFields` (454) instance can be used in a `for..in` construct. `TFieldsEnumerator` returns all the fields in the `TFields` collection. Therefore the following construct is possible:

```
Var
  F : TField;

begin
  // ...
  For F in MyDataset.Fields do
    begin
      // F is of type TField.
    end;
  // ...
```

Do not create an instance of `TFieldsEnumerator` manually. The compiler will do all that is needed when it encounters the `for..in` construct.

See also: `TField` (421), `TFields` (454), `#rtl.system.IEnumerator` (??)

### 11.34.2 Method overview

Page	Method	Description
460	Create	Create a new instance of <code>TFieldsEnumerator</code> .
460	MoveNext	Move the current field to the next field in the collection.

### 11.34.3 Property overview

Page	Properties	Access	Description
460	Current	r	Return the current field

### 11.34.4 TFieldsEnumerator.Create

Synopsis: Create a new instance of `TFieldsEnumerator`.

Declaration: `constructor Create(AFields: TFields)`

Visibility: `public`

Description: `Create` instantiates a new instance of `TFieldsEnumerator`. It stores the `AFields` reference, pointing to the `TFields` (454) instance that created the enumerator. It initializes the enumerator position.

### 11.34.5 TFieldsEnumerator.MoveNext

Synopsis: Move the current field to the next field in the collection.

Declaration: `function MoveNext : Boolean`

Visibility: `public`

Description: `MoveNext` moves the internal pointer to the next field in the fields collection, and returns `True` if the operation was a success. If no more fields are available, then `False` is returned.

See also: `TFieldsEnumerator.Current` (460)

### 11.34.6 TFieldsEnumerator.Current

Synopsis: Return the current field

Declaration: `Property Current : TField`

Visibility: `public`

Access: `Read`

Description: `Current` returns the current field. It will return a non-nil value only after `MoveNext` returned `True`.

See also: `TFieldsEnumerator.MoveNext` (460)

## 11.35 TFloatField

### 11.35.1 Description

`TFloatField` is the class created when a dataset must manage floating point values of double precision. It exposes a few new properties such as `Currency` (461), `MaxValue` (462), `MinValue` (462) and overrides some `TField` (421) methods to work with floating point data.

It should never be necessary to create an instance of `TFloatField` manually, a field of this class will be instantiated automatically for each floating-point field when a dataset is opened.

See also: `Currency` (461), `MaxValue` (462), `MinValue` (462)

### 11.35.2 Method overview

Page	Method	Description
461	<code>CheckRange</code>	Check whether a value is in the allowed range of values for the field
461	<code>Create</code>	Create a new instance of the <code>TFloatField</code>

### 11.35.3 Property overview

Page	Properties	Access	Description
<a href="#">461</a>	Currency	rw	Is the field a currency field.
<a href="#">462</a>	MaxValue	rw	Maximum value for the field
<a href="#">462</a>	MinValue	rw	Minimum value for the field
<a href="#">462</a>	Precision	rw	Precision (number of digits) of the field in text representations
<a href="#">461</a>	Value	rw	Value of the field as a double type

### 11.35.4 TFloatField.Create

Synopsis: Create a new instance of the `TFloatField`

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initializes a new instance of `TFloatField`. It calls the inherited constructor and then initializes some properties.

### 11.35.5 TFloatField.CheckRange

Synopsis: Check whether a value is in the allowed range of values for the field

Declaration: `function CheckRange(AValue: Double) : Boolean`

Visibility: `public`

Description: `CheckRange` returns `True` if `AValue` lies within the range defined by the `MinValue` ([462](#)) and `MaxValue` ([462](#)) properties. If the value lies outside of the allowed range, then `False` is returned.

See also: `MaxValue` ([462](#)), `MinValue` ([462](#))

### 11.35.6 TFloatField.Value

Synopsis: Value of the field as a double type

Declaration: `Property Value : Double`

Visibility: `public`

Access: `Read,Write`

Description: `Value` is redefined by `TFloatField` to return a value of type `Double`. It returns the same value as `TField.AsFloat` ([430](#))

See also: `TField.AsFloat` ([430](#)), `TField.Value` ([439](#))

### 11.35.7 TFloatField.Currency

Synopsis: Is the field a currency field.

Declaration: `Property Currency : Boolean`

Visibility: `published`

Access: `Read,Write`

**Description:** `Currency` can be set to `True` to indicate that the field contains data representing an amount of currency. This affects the way the `TField.DisplayText` (435) and `TField.Text` (438) properties format the value of the field: if the `Currency` property is `True`, then these properties will format the value as a currency value (generally appending the currency sign) and if the `Currency` property is `False`, then they will format it as a normal floating-point value.

See also: `TField.DisplayText` (435), `TField.Text` (438), `TNumericField.DisplayFormat` (484), `TNumericField.EditFormat` (484)

### 11.35.8 TFloatField.MaxValue

**Synopsis:** Maximum value for the field

**Declaration:** `Property MaxValue : Double`

**Visibility:** published

**Access:** Read,Write

**Description:** `MaxValue` can be set to a value different from zero, it is then the maximum value for the field if set to any value different from zero. When setting the field's value, the value may not be larger than `MaxValue`. Any attempt to write a larger value as the field's content will result in an exception. By default `MaxValue` equals 0, i.e. any floating-point value is allowed.

If `MaxValue` is set, `MinValue` (462) should also be set, because it will also be checked.

See also: `TFloatField.MinValue` (462)

### 11.35.9 TFloatField.MinValue

**Synopsis:** Minimum value for the field

**Declaration:** `Property MinValue : Double`

**Visibility:** published

**Access:** Read,Write

**Description:** `MinValue` can be set to a value different from zero, then it is the minimum value for the field. When setting the field's value, the value may not be less than `MinValue`. Any attempt to write a smaller value as the field's content will result in an exception. By default `MinValue` equals 0, i.e. any floating-point value is allowed.

If `MinValue` is set, `MaxValue` (462) should also be set, because it will also be checked.

See also: `TFloatField.MaxValue` (462), `TFloatField.CheckRange` (461)

### 11.35.10 TFloatField.Precision

**Synopsis:** Precision (number of digits) of the field in text representations

**Declaration:** `Property Precision : LongInt`

**Visibility:** published

**Access:** Read,Write

**Description:** `Precision` is the maximum number of digits that should be used when the field is converted to a textual representation in `TField.Displaytext` (435) or `TField.Text` (438), it is used in the arguments to `FormatFloat` (??).

See also: `TField.Displaytext` (435), `TField.Text` (438), `FormatFloat` (??)

## 11.36 TFMTBCDField

### 11.36.1 Description

TFMTBCDField is the field created when a data type of `ftFMTBCD` is encountered. It represents usually a fixed-precision floating point data type (BCD : Binary Coded Decimal data) such as the `DECIMAL` or `NUMERIC` field types in an SQL database.

See also: TFloatField ([460](#))

### 11.36.2 Method overview

Page	Method	Description
<a href="#">463</a>	CheckRange	Check value if it is in the range defined by MinValue and MaxValue
<a href="#">463</a>	Create	Create a new instance of the TFMTBCDField class.

### 11.36.3 Property overview

Page	Properties	Access	Description
<a href="#">464</a>	Currency	rw	Does the field contain currency data ?
<a href="#">464</a>	MaxValue	rw	Maximum value for the field
<a href="#">465</a>	MinValue	rw	Minimum value for the field
<a href="#">464</a>	Precision	rw	Total number of digits in the BCD data
<a href="#">465</a>	Size		Number of digits after the decimal point
<a href="#">464</a>	Value	rw	The value of the field as a BCD value

### 11.36.4 TFMTBCDField.Create

Synopsis: Create a new instance of the TFMTBCDField class.

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initializes a new instance of the TFMTBCDField class: it sets the `MinValue` ([316](#)), `MaxValue` ([316](#)), `Size` ([438](#)) (15) and `Precision` ([316](#)) (2) fields to their default values.

See also: `MinValue` ([316](#)), `MaxValue` ([316](#)), `Size` ([438](#)), `Precision` ([316](#))

### 11.36.5 TFMTBCDField.CheckRange

Synopsis: Check value if it is in the range defined by MinValue and MaxValue

Declaration: `function CheckRange(AValue: TBCD) : Boolean`

Visibility: `public`

Description: `CheckRange` checks whether `AValue` is between `MinValue` ([316](#)) and `MaxValue` ([316](#)) if they are both nonzero. If either of them is zero, then `True` is returned. The `MinValue` and `MaxValue` values themselves are also valid values.

See also: `MinValue` ([316](#)), `MaxValue` ([316](#))



### 11.36.6 TFMTBCDField.Value

Synopsis: The value of the field as a BCD value

Declaration: `Property Value : TBCD`

Visibility: public

Access: Read,Write

Description: `Value` is the value of the field as a BCD (Binary Coded Decimal) value.

See also: `TField.AsFloat` (430), `TField.AsCurrency` (429)

### 11.36.7 TFMTBCDField.Precision

Synopsis: Total number of digits in the BCD data

Declaration: `Property Precision : LongInt`

Visibility: published

Access: Read,Write

Description: `Precision` is the total number of digits in the BCD data. The maximum precision is 32.

See also: `TField.AsFloat` (430), `TField.AsCurrency` (429), `Size` (316)

### 11.36.8 TFMTBCDField.Currency

Synopsis: Does the field contain currency data ?

Declaration: `Property Currency : Boolean`

Visibility: published

Access: Read,Write

Description: `Currency` determines how the textual representation of the data is formatted. It has no influence on the actual data itself. If `True` it is represented as a currency (monetary value). If `DisplayFormat` (421) or `EditFormat` (421) are set, these values are used instead to format the value.

See also: `TField.DisplayFormat` (421), `TField.EditFormat` (421)

### 11.36.9 TFMTBCDField.MaxValue

Synopsis: Maximum value for the field

Declaration: `Property MaxValue : string`

Visibility: published

Access: Read,Write

Description: `MaxValue` can be set to a nonzero value to indicate the maximum value the field may contain. It must be set together with `MinValue` (316) or it will not have any effect.

See also: `TFMTBCDField.CheckRange` (463), `MinValue` (316)

### 11.36.10 TFMTBCDField.MinValue

Synopsis: Minimum value for the field

Declaration: `Property MinValue : string`

Visibility: published

Access: Read,Write

Description: `MinValue` can be set to a nonzero value to indicate the maximum value the field may contain. It must be set together with `MaxValue` (316) or it will not have any effect.

See also: `TFMTBCDField.CheckRange` (463), `MaxValue` (316)

### 11.36.11 TFMTBCDField.Size

Synopsis: Number of digits after the decimal point

Declaration: `Property Size :`

Visibility: published

Access:

Description: `Size` is the maximum number of digits allowed after the decimal point. Together with the `Precision` (316) property it determines the maximum allowed range of values for the field. This range can be restricted using the `MinValue` (316) and `MaxValue` (316) properties.

See also: `MinValue` (316), `MaxValue` (316), `Precision` (316)

## 11.37 TGraphicField

### 11.37.1 Description

`TGraphicField` is the class used when a dataset must manage graphical BLOB data. (`TField.DataType` (435) equals `ftGraphic`). It initializes some of the properties of the `TField` (421) class. All methods to be able to work with graphical BLOB data have been implemented in the `TBlobField` (348) parent class.

It should never be necessary to create an instance of `TGraphicsField` manually, a field of this class will be instantiated automatically for each graphical BLOB field when a dataset is opened.

See also: `TDataset` (371), `TField` (421), `TBlobField` (348), `TMemoField` (481), `TWideMemoField` (509)

### 11.37.2 Method overview

Page	Method	Description
465	Create	Create a new instance of the <code>TGraphicField</code> class

### 11.37.3 TGraphicField.Create

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

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

**Description:** `Create` initializes a new instance of the `TGraphicField` class. It calls the inherited destructor, and then sets some `TField` (421) properties to configure the instance for working with graphical BLOB values.

See also: `TField` (421)

## 11.38 TGUIDField

### 11.38.1 Description

`TGUIDField` is the class used when a dataset must manage native variant-typed data. (`TField.DataType` (435) equals `ftGUID`). It initializes some of the properties of the `TField` (421) class and overrides some of its methods to be able to work with variant data. It also adds a method to retrieve the field value as a native `TGUID` type.

It should never be necessary to create an instance of `TGUIDField` manually, a field of this class will be instantiated automatically for each GUID field when a dataset is opened.

See also: `TDataset` (371), `TField` (421), `TGUIDField.AsGuid` (466)

### 11.38.2 Method overview

Page	Method	Description
466	<code>Create</code>	Create a new instance of the <code>TGUIDField</code> class

### 11.38.3 Property overview

Page	Properties	Access	Description
466	<code>AsGuid</code>	rw	Field content as a GUID value

### 11.38.4 TGUIDField.Create

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

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** `public`

**Description:** `Create` initializes a new instance of the `TGUIDField` class. It calls the inherited destructor, and then sets some `TField` (421) properties to configure the instance for working with GUID values.

See also: `TField` (421)

### 11.38.5 TGUIDField.AsGuid

**Synopsis:** Field content as a GUID value

**Declaration:** `Property AsGuid : TGUID`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `AsGUID` can be used to get or set the field's content as a value of type `TGUID`.

See also: `TField.AsString` (431)

## 11.39 TIndexDef

### 11.39.1 Description

`TIndexDef` describes one index in a set of indexes of a `TDataset` (371) instance. The collection of indexes is described by the `TIndexDefs` (469) class. It just has the necessary properties to describe an index, but does not implement any functionality to maintain an index.

See also: `TIndexDefs` (469)

### 11.39.2 Method overview

Page	Method	Description
<a href="#">467</a>	Create	Create a new index definition

### 11.39.3 Property overview

Page	Properties	Access	Description
<a href="#">468</a>	CaseInsFields	rw	Fields in field list that are ordered case-insensitively
<a href="#">468</a>	DescFields	rw	Fields in field list that are ordered descending
<a href="#">467</a>	Expression	rw	Expression that makes up the index values
<a href="#">468</a>	Fields	rw	Fields making up the index
<a href="#">469</a>	Options	rw	Index options
<a href="#">469</a>	Source	rw	Source of the index

### 11.39.4 TIndexDef.Create

Synopsis: Create a new index definition

Declaration: constructor `Create`(Owner: `TIndexDefs`; const AName: string;  
const TheFields: string; TheOptions: `TIndexOptions`)  
; Overload

Visibility: public

Description: `Create` initializes a new `TIndexDef` (467) instance with the `AName` value as the index name, `AField` as the fields making up the index, and `TheOptions` as the options. `Owner` should be the `TIndexDefs` (469) instance to which the new `TIndexDef` can be added.

Errors: If an index with name `AName` already exists in the collection, an exception will be raised.

See also: `TIndexDefs` (469), `TIndexDef.Options` (469), `TIndexDef.Fields` (468)

### 11.39.5 TIndexDef.Expression

Synopsis: Expression that makes up the index values

Declaration: Property `Expression` : string

Visibility: published

Access: Read,Write

Description: `Expression` is an SQL expression based on which the index values are computed. It is only used when `ixExpression` is in `TIndexDef.Options` (469)

See also: `TIndexDef.Options` (469), `TIndexDef.Fields` (468)

### 11.39.6 TIndexDef.Fields

Synopsis: Fields making up the index

Declaration: `Property Fields : string`

Visibility: published

Access: Read,Write

Description: `Fields` is a list of fieldnames, separated by semicolons: the fields that make up the index, in case the index is not based on an expression. The list contains the names of all fields, regardless of whether the sort order for a particular field is ascending or descending. The fields should be in the right order, i.e. the first field is sorted on first, and so on.

The `TIndexDef.DescFields` (468) property can be used to determine the fields in the list that have a descending sort order. The `TIndexDef.CaseInsFields` (468) property determines which fields are sorted in a case-insensitive manner.

See also: `TIndexDef.DescFields` (468), `TIndexDef.CaseInsFields` (468), `TIndexDef.Expression` (467)

### 11.39.7 TIndexDef.CaseInsFields

Synopsis: Fields in field list that are ordered case-insensitively

Declaration: `Property CaseInsFields : string`

Visibility: published

Access: Read,Write

Description: `CaseInsFields` is a list of fieldnames, separated by semicolons. It contains the names of the fields in the `Fields` (468) property which are ordered in a case-insensitive manner. `CaseInsFields` may not contain fieldnames that do not appear in `Fields`.

See also: `TIndexDef.Fields` (468), `TIndexDef.Expression` (467), `TIndexDef.DescFields` (468)

### 11.39.8 TIndexDef.DescFields

Synopsis: Fields in field list that are ordered descending

Declaration: `Property DescFields : string`

Visibility: published

Access: Read,Write

Description: `DescFields` is a list of fieldnames, separated by semicolons. It contains the names of the fields in the `Fields` (468) property which are ordered in a descending manner. `DescFields` may not contain fieldnames that do not appear in `Fields`.

See also: `TIndexDef.Fields` (468), `TIndexDef.Expression` (467), `TIndexDef.DescFields` (468)

### 11.39.9 TIndexDef.Options

Synopsis: Index options

Declaration: `Property Options : TIndexOptions`

Visibility: published

Access: Read,Write

Description: `Options` describes the various properties of the index. This is usually filled by the dataset that provides the index definitions. For datasets that provide In-memory indexes, this should be set prior to creating the index: it cannot be changed once the index is created.

See the description of `TIndexOption` ([327](#)) for more information on the various available options.

See also: `TIndexOptions` ([327](#))

### 11.39.10 TIndexDef.Source

Synopsis: Source of the index

Declaration: `Property Source : string`

Visibility: published

Access: Read,Write

Description: `Source` describes where the index comes from. This is a property for the convenience of the various datasets that provide indexes: they can use it to describe the source of the index.

## 11.40 TIndexDefs

### 11.40.1 Description

`TIndexDefs` is used to keep a collection of index (sort order) definitions. It can be used by classes that provide in-memory or on-disk indexes to provide a list of available indexes.

See also: `TIndexDef` ([467](#)), `TIndexDefs.Items` ([471](#))

### 11.40.2 Method overview

Page	Method	Description
<a href="#">470</a>	Add	Add a new index definition with given name and options
<a href="#">470</a>	AddIndexDef	Add a new, empty, index definition
<a href="#">470</a>	Create	Create a new <code>TIndexDefs</code> instance
<a href="#">470</a>	Find	Find an index by name
<a href="#">471</a>	FindIndexForFields	Find index definition based on field names
<a href="#">471</a>	GetIndexForFields	Get index definition based on field names
<a href="#">471</a>	Update	Called whenever one of the items changes

### 11.40.3 Property overview

Page	Properties	Access	Description
<a href="#">471</a>	Items	rw	Indexed access to the index definitions

#### 11.40.4 TIndexDefs.Create

Synopsis: Create a new TIndexDefs instance

Declaration: `constructor Create(ADataset: TDataSet); Virtual; Overload`

Visibility: public

Description: `Create` initializes a new instance of the TIndexDefs class. It simply calls the inherited destructor with the appropriate item class, TIndexDef (467).

See also: TIndexDef (467), TIndexDefs.Destroy (469)

#### 11.40.5 TIndexDefs.Add

Synopsis: Add a new index definition with given name and options

Declaration: `procedure Add(const Name: string; const Fields: string;  
Options: TIndexOptions); Overload`

Visibility: public

Description: `Add` adds a new TIndexDef (467) instance to the list of indexes. It initializes the index definition properties `Name`, `Fields` and `Options` with the values given in the parameters with the same names.

Errors: If an index with the same `Name` already exists in the list of indexes, an exception will be raised.

See also: TIndexDef (467), TNamedItem.Name (483), TIndexDef.Fields (468), TIndexDef.Options (469), TIndexDefs.AddIndexDef (470)

#### 11.40.6 TIndexDefs.AddIndexDef

Synopsis: Add a new, empty, index definition

Declaration: `function AddIndexDef : TIndexDef`

Visibility: public

Description: `AddIndexDef` adds a new TIndexDef (467) instance to the list of indexes, and returns the newly created instance. It does not initialize any of the properties of the new index definition.

See also: TIndexDefs.Add (470)

#### 11.40.7 TIndexDefs.Find

Synopsis: Find an index by name

Declaration: `function Find(const IndexName: string) : TIndexDef`

Visibility: public

Description: `Find` overloads the TDefCollection.Find (419) method to search and return a TIndexDef (467) instance based on the name. The search is case-insensitive and raises an exception if no matching index definition was found. Note: `TIndexDefs.IndexOf` can be used instead if an exception is not desired.

See also: TIndexDef (467), TDefCollection.Find (419), TIndexDefs.FindIndexForFields (471)

### 11.40.8 TIndexDefs.FindIndexForFields

Synopsis: Find index definition based on field names

Declaration: `function FindIndexForFields(const Fields: string) : TIndexDef`

Visibility: public

Description: `FindIndexForFields` searches in the list of indexes for an index whose `TIndexDef.Fields` (468) property matches the list of fields in `Fields`. If it finds an index definition, then it returns the found instance.

Errors: If no matching definition is found, an exception is raised. This is different from other `Find` functionality, where `Find` usually returns `Nil` if nothing is found.

See also: `TIndexDef` (467), `TIndexDefs.Find` (470), `TIndexDefs.GetindexForFields` (471)

### 11.40.9 TIndexDefs.GetIndexForFields

Synopsis: Get index definition based on field names

Declaration: `function GetIndexForFields(const Fields: string;  
CaseInsensitive: Boolean) : TIndexDef`

Visibility: public

Description: `GetIndexForFields` searches in the list of indexes for an index whose `TIndexDef.Fields` (468) property matches the list of fields in `Fields`. If `CaseInsensitive` is `True` it only searches for case-sensitive indexes. If it finds an index definition, then it returns the found instance. If it does not find a matching definition, `Nil` is returned.

See also: `TIndexDef` (467), `TIndexDefs.Find` (470), `TIndexDefs.FindIndexForFields` (471)

### 11.40.10 TIndexDefs.Update

Synopsis: Called whenever one of the items changes

Declaration: `procedure Update; Virtual; Overload`

Visibility: public

Description: `Update` can be called to have the dataset update its index definitions.

### 11.40.11 TIndexDefs.Items

Synopsis: Indexed access to the index definitions

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

Visibility: public

Access: Read,Write

Description: `Items` is redefined by `TIndexDefs` using `TIndexDef` as the type for the elements. It is the default property of the `TIndexDefs` class.

See also: `TIndexDef` (467)



## 11.41 TIntegerField

### 11.41.1 Description

TIntegerField is an alias for TLongintField ([474](#)).

See also: TLongintField ([474](#)), TField ([421](#))

## 11.42 TLargeintField

### 11.42.1 Description

TLargeIntField is instantiated when a dataset must manage a field with 64-bit signed data: the data type `ftLargeInt`. It overrides some methods of TField ([421](#)) to handle int64 data, and sets some of the properties to values for int64 data. It also introduces some methods and properties specific to 64-bit integer data such as MinValue ([473](#)) and MaxValue ([473](#)).

It should never be necessary to create an instance of TLargeIntField manually, a field of this class will be instantiated automatically for each int64 field when a dataset is opened.

See also: TField ([421](#)), MinValue ([473](#)), MaxValue ([473](#))

### 11.42.2 Method overview

Page	Method	Description
<a href="#">472</a>	CheckRange	Check whether a values falls within the allowed range
<a href="#">472</a>	Create	Create a new instance of the TLargeintField class

### 11.42.3 Property overview

Page	Properties	Access	Description
<a href="#">473</a>	MaxValue	rw	Maximum value for the field
<a href="#">473</a>	MinValue	rw	Minimum value for the field
<a href="#">473</a>	Value	rw	Field contents as a 64-bit integer value

### 11.42.4 TLargeintField.Create

Synopsis: Create a new instance of the TLargeintField class

Declaration: `constructor Create(AOwner: TComponent);` Override

Visibility: public

Description: Create initializes a new instance of the TLargeIntField class: it calls the inherited constructor and then initializes the various properties of Tfield ([421](#)) and MinValue ([473](#)) and MaxValue ([473](#)).

See also: TField ([421](#)), MinValue ([473](#)), MaxValue ([473](#))

### 11.42.5 TLargeintField.CheckRange

Synopsis: Check whether a values falls within the allowed range

Declaration: `function CheckRange(AValue: LargeInt) : Boolean`

Visibility: public

Description: `CheckRange` returns `True` if `AValue` lies within the range defined by the `MinValue` (473) and `MaxValue` (473) properties. If the value lies outside of the allowed range, then `False` is returned.

See also: `MaxValue` (473), `MinValue` (473)

### 11.42.6 `TLargeIntField.Value`

Synopsis: Field contents as a 64-bit integer value

Declaration: `Property Value : LargeInt`

Visibility: public

Access: Read,Write

Description: `Value` is redefined by `TLargeIntField` as a 64-bit integer value. It returns the same value as `TField.AsLargeInt` (431).

See also: `TField.Value` (439), `TField.AsLargeInt` (431)

### 11.42.7 `TLargeIntField.MaxValue`

Synopsis: Maximum value for the field

Declaration: `Property MaxValue : LargeInt`

Visibility: published

Access: Read,Write

Description: `MaxValue` is the maximum value for the field if set to any value different from zero. When setting the field's value, the value may not be larger than `MaxValue`. Any attempt to write a larger value as the field's content will result in an exception. By default `MaxValue` equals 0, i.e. any integer value is allowed.

If `MaxValue` is set, `MinValue` (473) should also be set, because it will also be checked.

See also: `TLargeIntField.MinValue` (473)

### 11.42.8 `TLargeIntField.MinValue`

Synopsis: Minimum value for the field

Declaration: `Property MinValue : LargeInt`

Visibility: published

Access: Read,Write

Description: `MinValue` is the minimum value for the field. When setting the field's value, the value may not be less than `MinValue`. Any attempt to write a smaller value as the field's content will result in an exception. By default `MinValue` equals 0, i.e. any integer value is allowed.

If `MinValue` is set, `MaxValue` (473) should also be set, because it will also be checked.

See also: `TLargeIntField.MaxValue` (473)

## 11.43 TLongintField

### 11.43.1 Description

TLongintField is instantiated when a dataset must manage a field with 32-bit signed data: the data type `ftInteger`. It overrides some methods of TField (421) to handle integer data, and sets some of the properties to values for integer data. It also introduces some methods and properties specific to integer data such as MinValue (475) and MaxValue (475).

It should never be necessary to create an instance of TLongintField manually, a field of this class will be instantiated automatically for each integer field when a dataset is opened.

See also: TField (421), MaxValue (475), MinValue (475)

### 11.43.2 Method overview

Page	Method	Description
<a href="#">474</a>	CheckRange	Check whether a valid is in the allowed range of values for the field
<a href="#">474</a>	Create	Create a new instance of TLongintField

### 11.43.3 Property overview

Page	Properties	Access	Description
<a href="#">475</a>	MaxValue	rw	Maximum value for the field
<a href="#">475</a>	MinValue	rw	Minimum value for the field
<a href="#">475</a>	Value	rw	Value of the field as longint

### 11.43.4 TLongintField.Create

Synopsis: Create a new instance of TLongintField

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initializes a new instance of TLongintField. After calling the inherited constructor, it initializes the MinValue (475) and MaxValue (475) properties.

See also: TField (421), MaxValue (475), MinValue (475)

### 11.43.5 TLongintField.CheckRange

Synopsis: Check whether a valid is in the allowed range of values for the field

Declaration: `function CheckRange(AValue: LongInt) : Boolean`

Visibility: `public`

Description: `CheckRange` returns `True` if `AValue` lies within the range defined by the MinValue (475) and MaxValue (475) properties. If the value lies outside of the allowed range, then `False` is returned.

See also: MaxValue (475), MinValue (475)

### 11.43.6 TLongintField.Value

Synopsis: Value of the field as longint

Declaration: `Property Value : LongInt`

Visibility: public

Access: Read,Write

Description: `Value` is redefined by `TLongintField` as a 32-bit signed integer value. It returns the same value as the `TField.AsInteger` (431) property.

See also: `TField.Value` (439)

### 11.43.7 TLongintField.MaxValue

Synopsis: Maximum value for the field

Declaration: `Property MaxValue : LongInt`

Visibility: published

Access: Read,Write

Description: `MaxValue` is the maximum value for the field. When setting the field's value, the value may not be larger than `MaxValue`. Any attempt to write a larger value as the field's content will result in an exception. By default `MaxValue` equals `MaxInt`, i.e. any integer value is allowed.

See also: `MinValue` (475)

### 11.43.8 TLongintField.MinValue

Synopsis: Minimum value for the field

Declaration: `Property MinValue : LongInt`

Visibility: published

Access: Read,Write

Description: `MinValue` is the minimum value for the field. When setting the field's value, the value may not be less than `MinValue`. Any attempt to write a smaller value as the field's content will result in an exception. By default `MinValue` equals `-MaxInt`, i.e. any integer value is allowed.

See also: `MaxValue` (475)

## 11.44 TLookupList

### 11.44.1 Description

`TLookupList` is a list object used for storing values of lookup operations by lookup fields. There should be no need to create an instance of `TLookupList` manually, the `TField` instance will create an instance of `TLookupList` on demand.

See also: `TField.LookupCache` (443)

**11.44.2 Method overview**

Page	Method	Description
<a href="#">476</a>	Add	Add a key, value pair to the list
<a href="#">476</a>	Clear	Remove all key, value pairs from the list
<a href="#">476</a>	Create	Create a new instance of TLookupList.
<a href="#">476</a>	Destroy	Free a TLookupList instance from memory
<a href="#">477</a>	FirstKeyByValue	Find the first key that matches a value
<a href="#">477</a>	ValueOfKey	Look up value based on a key
<a href="#">477</a>	ValuesToStrings	Convert values to stringlist

**11.44.3 TLookupList.Create**

Synopsis: Create a new instance of TLookupList.

Declaration: `constructor Create`

Visibility: `public`

Description: `Create` sets up the necessary structures to manage a list of lookup values for a lookup field.

See also: `TLookupList.Destroy` ([476](#))

**11.44.4 TLookupList.Destroy**

Synopsis: Free a TLookupList instance from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` frees all resources (mostly memory) allocated by the lookup list, and calls then the inherited destructor.

See also: `TLookupList.Create` ([476](#))

**11.44.5 TLookupList.Add**

Synopsis: Add a key, value pair to the list

Declaration: `procedure Add(const AKey: Variant; const AValue: Variant)`

Visibility: `public`

Description: `Add` will add the value `AValue` to the list and associate it with key `AKey`. The same key cannot be added twice.

See also: `TLookupList.Clear` ([476](#))

**11.44.6 TLookupList.Clear**

Synopsis: Remove all key, value pairs from the list

Declaration: `procedure Clear`

Visibility: `public`

Description: `Clear` removes all keys and associated values from the list.

See also: `TLookupList.Add` ([476](#))

### 11.44.7 TLookupList.FirstKeyByValue

Synopsis: Find the first key that matches a value

Declaration: `function FirstKeyByValue(const AValue: Variant) : Variant`

Visibility: public

Description: `FirstKeyByValue` does a reverse lookup: it returns the first key value in the list that matches the `AValue` value. If none is found, `Null` is returned. This mechanism is quite slow, as a linear search is performed.

Errors: If no key is found, `Null` is returned.

See also: `TLookupList.ValueOfKey` ([477](#))

### 11.44.8 TLookupList.ValueOfKey

Synopsis: Look up value based on a key

Declaration: `function ValueOfKey(const AKey: Variant) : Variant`

Visibility: public

Description: `ValueOfKey` does a value lookup based on a key: it returns the value in the list that matches the `AKey` key. If none is found, `Null` is returned. This mechanism is quite slow, as a linear search is performed.

See also: `TLookupList.FirstKeyByValue` ([477](#)), `TLookupList.Add` ([476](#))

### 11.44.9 TLookupList.ValuesToStrings

Synopsis: Convert values to stringlist

Declaration: `procedure ValuesToStrings(AStrings: TStrings)`

Visibility: public

Description: `ValuesToStrings` converts the list of values to a stringlist, so they can be used e.g. in a drop-down list.

See also: `TLookupList.ValueOfKey` ([477](#))

## 11.45 TMasterDataLink

### 11.45.1 Description

`TMasterDataLink` is a `TDatalink` descendent which handles master-detail relations. It can be used in `TDataset` ([371](#)) descendents that must have master-detail functionality: the detail dataset creates an instance of `TMasterDataLink` to point to the master dataset, which is subsequently available through the `TDatalink.Dataset` ([370](#)) property.

The class also provides functionality for keeping a list of fields that make up the master-detail functionality, in the `TMasterDataLink.FieldName` ([478](#)) and `TMasterDataLink.Fields` ([479](#)) properties.

This class should never be used in application code.

See also: `TDataset` ([371](#)), `TDatalink.DataSource` ([370](#)), `TDatalink.DataSet` ([370](#)), `TMasterDataLink.FieldName` ([478](#)), `TMasterDataLink.Fields` ([479](#))

### 11.45.2 Method overview

Page	Method	Description
<a href="#">478</a>	Create	Create a new instance of TMasterDataLink
<a href="#">478</a>	Destroy	Free the datalink instance from memory

### 11.45.3 Property overview

Page	Properties	Access	Description
<a href="#">478</a>	FieldNames	rw	List of fieldnames that make up the master-detail relationship
<a href="#">479</a>	Fields	r	List of fields as specified in FieldNames
<a href="#">479</a>	OnMasterChange	rw	Called whenever the master dataset data changes
<a href="#">479</a>	OnMasterDisable	rw	Called whenever the master dataset is disabled

### 11.45.4 TMasterDataLink.Create

Synopsis: Create a new instance of TMasterDataLink

Declaration: `constructor Create(ADataset: TDataSet); Virtual`

Visibility: public

Description: Create initializes a new instance of TMasterDataLink. The ADataset parameter is the detail dataset in the master-detail relation: it is saved in the DetailDataset ([421](#)) property. The master dataset must be set through the DataSource ([370](#)) property, and is usually set by the application programmer.

See also: TDetailDataLink.DetailDataset ([421](#)), TDatalink.Datasource ([370](#))

### 11.45.5 TMasterDataLink.Destroy

Synopsis: Free the datalink instance from memory

Declaration: `destructor Destroy; Override`

Visibility: public

Description: Destroy cleans up the resources used by TMasterDataLink and then calls the inherited destructor.

See also: TMasterDataLink.Create ([478](#))

### 11.45.6 TMasterDataLink.FieldNames

Synopsis: List of fieldnames that make up the master-detail relationship

Declaration: `Property FieldNames : string`

Visibility: public

Access: Read,Write

Description: FieldNames is a semicolon-separated list of fieldnames in the master dataset (TDatalink.Dataset ([370](#))) on which the master-detail relationship is based. Setting this property will fill the TMasterDataLink.Fields ([479](#)) property with the field instances of the master dataset.

See also: TMasterDataLink.Fields ([479](#)), TDatalink.Dataset ([370](#)), TDataSet.GetFieldList ([385](#))

### 11.45.7 TMasterDataLink.Fields

Synopsis: List of fields as specified in `FieldNames`

Declaration: `Property Fields : TList`

Visibility: `public`

Access: `Read`

Description: `Fields` is filled with the `TField` (421) instances from the master dataset (`TDatalink.Dataset` (370)) when the `FieldNames` (478) property is set, and when the master dataset opens.

See also: `TField` (421), `TMasterDataLink.FieldNames` (478)

### 11.45.8 TMasterDataLink.OnMasterChange

Synopsis: Called whenever the master dataset data changes

Declaration: `Property OnMasterChange : TNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `OnMasterChange` is called whenever the field values in the master dataset changes, i.e. when it becomes active, or when the current record changes. If the `TMasterDataLink.Fields` (479) list is empty, `TMasterDataLink.OnMasterDisable` (479) is called instead.

See also: `TMasterDataLink.OnMasterDisable` (479)

### 11.45.9 TMasterDataLink.OnMasterDisable

Synopsis: Called whenever the master dataset is disabled

Declaration: `Property OnMasterDisable : TNotifyEvent`

Visibility: `public`

Access: `Read,Write`

Description: `OnMasterDisable` is called whenever the master dataset is disabled, or when it is active and the field list is empty.

See also: `TMasterDataLink.OnMasterChange` (479)

## 11.46 TMasterParamsDataLink

### 11.46.1 Description

`TMasterParamsDataLink` is a `TDataLink` (366) descendent that can be used to establish a master-detail relationship between 2 `TDataset` instances where the detail dataset is parameterized using a `TParams` instance. It takes care of closing and opening the detail dataset and copying the parameter values from the master dataset whenever the data in the master dataset changes.

See also: `TDatalink` (366), `TDataset` (371), `TParams` (497), `TParam` (484)



**11.46.2 Method overview**

Page	Method	Description
<a href="#">480</a>	CopyParamsFromMaster	Copy parameter values from master dataset.
<a href="#">480</a>	Create	Initialize a new TMasterParamsDataLink instance
<a href="#">480</a>	RefreshParamNames	Refresh the list of parameter names

**11.46.3 Property overview**

Page	Properties	Access	Description
<a href="#">481</a>	Params	rw	Parameters of detail dataset.

**11.46.4 TMasterParamsDataLink.Create**

Synopsis: Initialize a new TMasterParamsDataLink instance

Declaration: `constructor Create(ADataset: TDataSet); Override`

Visibility: `public`

Description: `Create` first calls the inherited constructor using `ADataset`, and then looks for a property named `Params` of type `TParams` ([497](#)) in the published properties of `ADataset` and assigns it to the `Params` ([481](#)) property.

See also: `TDataSet` ([371](#)), `TParams` ([497](#)), `TMasterParamsDataLink.Params` ([481](#))

**11.46.5 TMasterParamsDataLink.RefreshParamNames**

Synopsis: Refresh the list of parameter names

Declaration: `procedure RefreshParamNames; Virtual`

Visibility: `public`

Description: `RefreshParamNames` scans the `Params` ([481](#)) property and sets the `FieldNames` ([478](#)) property to the list of parameter names.

See also: `TMasterParamsDataLink.Params` ([481](#)), `TMasterDataLink.FieldNames` ([478](#))

**11.46.6 TMasterParamsDataLink.CopyParamsFromMaster**

Synopsis: Copy parameter values from master dataset.

Declaration: `procedure CopyParamsFromMaster(CopyBound: Boolean); Virtual`

Visibility: `public`

Description: `CopyParamsFromMaster` calls `TParams.CopyParamValuesFromDataset` ([502](#)), passing it the master dataset: it provides the parameters of the detail dataset with their new values. If `CopyBound` is `false`, then only parameters with their `Bound` ([494](#)) property set to `False` are copied. If it is `True` then the value is set for all parameters.

Errors: If the master dataset does not have a corresponding field for each parameter, then an exception will be raised.

See also: `TParams.CopyParamValuesFromDataset` ([502](#)), `TParam.Bound` ([494](#))

### 11.46.7 TMasterParamsDataLink.Params

Synopsis: Parameters of detail dataset.

Declaration: `Property Params : TParams`

Visibility: `public`

Access: `Read,Write`

Description: `Params` is the `TParams` instance of the detail dataset. If the detail dataset contains a property named `Params` of type `TParams`, then it will be set when the `TMasterParamsDataLink` instance was created. If the property is not published, or has another name, then the `Params` property must be set in code.

See also: [Tparams \(497\)](#), [TMasterParamsDataLink.Create \(480\)](#)

## 11.47 TMemoField

### 11.47.1 Description

`TMemoField` is the class used when a dataset must manage memo (Text BLOB) data. (`TField.DataType (435)` equals `ftMemo`). It initializes some of the properties of the `TField (421)` class. All methods to be able to work with memo fields have been implemented in the `TBlobField (348)` parent class.

It should never be necessary to create an instance of `TMemoField` manually, a field of this class will be instantiated automatically for each memo field when a dataset is opened.

See also: [TDataSet \(371\)](#), [TField \(421\)](#), [TBLOBField \(348\)](#), [TWideMemoField \(509\)](#), [TGraphicField \(465\)](#)

### 11.47.2 Method overview

Page	Method	Description
<a href="#">481</a>	<code>Create</code>	Create a new instance of the <code>TMemoField</code> class

### 11.47.3 Property overview

Page	Properties	Access	Description
<a href="#">482</a>	<code>CodePage</code>	<code>r</code>	Codepage of the memo field string data
<a href="#">482</a>	<code>Transliterate</code>		Should the contents of the field be transliterated

### 11.47.4 TMemoField.Create

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

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initializes a new instance of the `TMemoField` class. It calls the inherited destructor, and then sets some `TField (421)` properties to configure the instance for working with memo values.

See also: [TField \(421\)](#)

### 11.47.5 TMemoField.CodePage

Synopsis: Codepage of the memo field string data

Declaration: `Property CodePage : TSystemCodePage`

Visibility: public

Access: Read

Description: `CodePage` is the code page of the string data in the field. It is determined when the field is initially created from the dataset's data, and cannot be changed while the dataset is active.

See also: `TField.AsString` ([431](#)), `TFieldDef.CodePage` ([450](#))

### 11.47.6 TMemoField.Transliterate

Synopsis: Should the contents of the field be transliterated

Declaration: `Property Transliterate :`

Visibility: published

Access:

Description: `Transliterate` is redefined from `TBlobField.Transliterate` ([351](#)) with a default value of `true`.

See also: `TBlobField.Transliterate` ([351](#)), `TStringField.Transliterate` ([506](#)), `TDataset.Translate` ([391](#))

## 11.48 TNamedItem

### 11.48.1 Description

`NamedItem` is a `TCollectionItem` ([??](#)) descendent which introduces a `Name` ([483](#)) property. It automatically returns the value of the `Name` property as the value of the `DisplayName` ([482](#)) property.

See also: `DisplayName` ([482](#)), `Name` ([483](#))

### 11.48.2 Property overview

Page	Properties	Access	Description
<a href="#">482</a>	<code>DisplayName</code>	rw	Display name
<a href="#">483</a>	<code>Name</code>	rw	Name of the item

### 11.48.3 TNamedItem.DisplayName

Synopsis: Display name

Declaration: `Property DisplayName : string`

Visibility: public

Access: Read,Write

Description: `DisplayName` is declared in `TCollectionItem` ([??](#)), and is made public in `TNamedItem`. The value equals the value of the `Name` ([483](#)) property.

See also: `Name` ([483](#))

### 11.48.4 TNamedItem.Name

Synopsis: Name of the item

Declaration: `Property Name : string`

Visibility: `published`

Access: `Read,Write`

Description: `Name` is the name of the item in the collection. This property is also used as the value for the `DisplayName` (482) property. If the `TNamedItem` item is owned by a `TDefCollection` (419) collection, then the name must be unique, i.e. each `Name` value may appear only once in the collection.

See also: `DisplayName` (482), `TDefCollection` (419)

## 11.49 TNumericField

### 11.49.1 Description

`TNumericField` is an abstract class which overrides some of the methods of `TField` (421) to handle numerical data. It also introduces or publishes a couple of properties that are only relevant in the case of numerical data, such as `TNumericField.DisplayFormat` (484) and `TNumericField.EditFormat` (484).

Since `TNumericField` is an abstract class, it must never be instantiated directly. Instead one of the descendent classes should be created.

See also: `TField` (421), `TNumericField.DisplayFormat` (484), `TNumericField.EditFormat` (484), `TField.Alignment` (440), `TIntegerField` (472), `TLargeIntField` (472), `TFloatField` (460), `TBCDField` (344)

### 11.49.2 Method overview

Page	Method	Description
483	<code>Create</code>	Create a new instance of <code>TNumericField</code>

### 11.49.3 Property overview

Page	Properties	Access	Description
484	<code>Alignment</code>		Alignment of the field
484	<code>DisplayFormat</code>	<code>rw</code>	Format string for display of numerical data
484	<code>EditFormat</code>	<code>rw</code>	Format string for editing of numerical data

### 11.49.4 TNumericField.Create

Synopsis: Create a new instance of `TNumericField`

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` calls the inherited constructor and then initializes the `TField.Alignment` (440) property with

See also: `TField.Alignment` (440)

### 11.49.5 TNumericField.Alignment

Synopsis: Alignment of the field

Declaration: `Property Alignment :`

Visibility: published

Access:

Description: `Alignment` is published by `TNumericField` with `taRightJustify` as a default value.

See also: `TField.Alignment` ([440](#))

### 11.49.6 TNumericField.DisplayFormat

Synopsis: Format string for display of numerical data

Declaration: `Property DisplayFormat : string`

Visibility: published

Access: Read,Write

Description: `DisplayFormat` specifies a format string (such as used by the `Format` (??) and `FormatFloat` (??) functions) for display purposes: the `TField.DisplayText` ([435](#)) property will use this property to format the field's value. Which formatting function (and, consequently, which format can be entered) is used depends on the descendent of the `TNumericField` class.

See also: `Format` (??), `FormatFloat` (??), `TField.DisplayText` ([435](#)), `TNumericField.EditFormat` ([484](#))

### 11.49.7 TNumericField.EditFormat

Synopsis: Format string for editing of numerical data

Declaration: `Property EditFormat : string`

Visibility: published

Access: Read,Write

Description: `EditFormat` specifies a format string (such as used by the `Format` (??) and `FormatFloat` (??) functions) for editing purposes: the `TField.Text` ([438](#)) property will use this property to format the field's value. Which formatting function (and, consequently, which format can be entered) is used depends on the descendent of the `TNumericField` class.

See also: `Format` (??), `FormatFloat` (??), `TField.Text` ([438](#)), `TNumericField.DisplayFormat` ([484](#))

## 11.50 TParam

### 11.50.1 Description

`TParam` is one item in a `TParams` ([497](#)) collection. It describes the name (`TParam.Name` ([496](#))), type (`ParamType` ([496](#))) and value (`TParam.Value` ([497](#))) of a parameter in a parameterized query or stored procedure. Under normal circumstances, it should never be necessary to create a `TParam` instance manually; the `TDataset` ([371](#)) descendent that owns the parameters should have created all necessary `TParam` instances.

See also: `TParams` ([497](#))

**11.50.2 Method overview**

Page	Method	Description
486	Assign	Assign one parameter instance to another
486	AssignField	Copy value from field instance
487	AssignFieldValue	Assign field value to the parameter.
487	AssignFromField	Copy field type and value
486	AssignToField	Assign parameter value to field
487	Clear	Clear the parameter value
486	Create	Create a new parameter value
487	GetData	Get the parameter value from a memory buffer
488	GetDataSize	Return the size of the data.
488	LoadFromFile	Load a parameter value from file
488	LoadFromStream	Load a parameter value from stream
489	SetBlobData	Set BLOB data
489	SetData	Set the parameter value from a buffer

**11.50.3 Property overview**

Page	Properties	Access	Description
492	AsAnsiString	rw	Parameter contents as an ANSI string
489	AsBCD	rw	Get or set parameter value as BCD value
489	AsBlob	rw	Return parameter value as a blob
490	AsBoolean	rw	Get/Set parameter value as a boolean value
490	AsBytes	rw	Get or set parameter value as TBytes
490	AsCurrency	rw	Get/Set parameter value as a currency value
490	AsDate	rw	Get/Set parameter value as a date (TDateTime) value
491	AsDateTime	rw	Get/Set parameter value as a date/time (TDateTime) value
491	AsFloat	rw	Get/Set parameter value as a floating-point value
494	AsFMTBCD	rw	Parameter value as a BCD value
491	AsInteger	rw	Get/Set parameter value as an integer (32-bit) value
491	AsLargeInt	rw	Get/Set parameter value as a 64-bit integer value
492	AsMemo	rw	Get/Set parameter value as a memo (string) value
492	AsSmallInt	rw	Get/Set parameter value as a smallint value
492	AsString	rw	Get/Set parameter value as a string value
493	AsTime	rw	Get/Set parameter value as a time (TDateTime) value
493	AsUnicodeString	rw	Parameter contents as a Unicode string
493	AsUTF8String	rw	Parameter contents as an UTF8 string
495	AsWideString	rw	Get/Set the value as a widestring
493	AsWord	rw	Get/Set parameter value as a word value
494	Bound	rw	Is the parameter value bound (set to fixed value)
494	Dataset	r	Dataset to which this parameter belongs
495	DataType	rw	Data type of the parameter
494	IsNull	r	Is the parameter empty
496	Name	rw	Name of the parameter
495	NativeStr	rw	No description available
496	NumericScale	rw	Numeric scale
496	ParamType	rw	Type of parameter
497	Precision	rw	Precision of the BCD value
497	Size	rw	Size of the parameter
495	Text	rw	Read or write the value of the parameter as a string
497	Value	rws	Value as a variant

### 11.50.4 TParam.Create

Synopsis: Create a new parameter value

Declaration: `constructor Create(ACollection: TCollection); Override; Overload`  
`constructor Create(AParams: TParams; AParamType: TParamType); Overload`  
`; Reintroduce`

Visibility: public

Description: `Create` first calls the inherited `create`, and then initializes the parameter properties. The first form creates a default parameter, the second form is a convenience function and initializes a parameter of a certain kind (`AParamType`), in which case the owning `TParams` collection must be specified in `AParams`

See also: `TParams` ([497](#))

### 11.50.5 TParam.Assign

Synopsis: Assign one parameter instance to another

Declaration: `procedure Assign(Source: TPersistent); Override`

Visibility: public

Description: `Assign` copies the `Name`, `ParamType`, `Bound`, `Value`, `SizePrecision` and `NumericScale` properties from `ASource` if it is of type `TParam`. If `Source` is of type `TField` ([421](#)), then it is passed to `TParam.AssignField` ([486](#)). If `Source` is of type `TStrings`, then it is assigned to `TParams.AsMemo` ([497](#)).

Errors: If `Source` is not of type `TParam`, `TField` or `TStrings`, an exception will be raised.

See also: `TField` ([421](#)), `TParam.Name` ([496](#)), `TParam.Bound` ([494](#)), `TParam.NumericScale` ([496](#)), `TParam.ParamType` ([496](#)), `TParam.value` ([497](#)), `TParam.Size` ([497](#)), `TParam.AssignField` ([486](#)), `TParam.AsMemo` ([492](#))

### 11.50.6 TParam.AssignField

Synopsis: Copy value from field instance

Declaration: `procedure AssignField(Field: TField)`

Visibility: public

Description: `AssignField` copies the `Field`, `FieldName` ([442](#)) and `Value` ([439](#)) to the parameter instance. The parameter is bound after this operation. If `Field` is `Nil` then the parameter name and value are cleared.

See also: `TParam.assign` ([486](#)), `TParam.AssignToField` ([486](#)), `TParam.AssignFieldValue` ([487](#))

### 11.50.7 TParam.AssignToField

Synopsis: Assign parameter value to field

Declaration: `procedure AssignToField(Field: TField)`

Visibility: public

Description: `AssignToField` copies the parameter value ([497](#)) to the field instance. If `Field` is `Nil`, nothing happens.

**Errors:** An `EDatabaseError` (335) exception is raised if the field has an unsupported field type (for types `ftCursor`, `ftArray`, `ftDataset`, `ftReference`).

See also: `TParam.Assign` (486), `TParam.AssignField` (486), `TParam.AssignFromField` (487)

### 11.50.8 TParam.AssignFieldValue

**Synopsis:** Assign field value to the parameter.

**Declaration:** `procedure AssignFieldValue(Field: TField; const AValue: Variant)`

**Visibility:** public

**Description:** `AssignFieldValue` copies only the field type from `Field` and the value from the `AValue` parameter. It sets the `TParam.Bound` (494) bound parameter to `True`. This method is called from `TParam.AssignField` (486).

See also: `TField` (421), `TParam.AssignField` (486), `TParam.Bound` (494)

### 11.50.9 TParam.AssignFromField

**Synopsis:** Copy field type and value

**Declaration:** `procedure AssignFromField(Field: TField)`

**Visibility:** public

**Description:** `AssignFromField` copies the field value (439) and data type (`TField.DataType` (435)) to the parameter instance. If `Field` is `Nil`, nothing happens. This is the reverse operation of `TParam.AssignToField` (486).

**Errors:** An `EDatabaseError` (335) exception is raised if the field has an unsupported field type (for types `ftCursor`, `ftArray`, `ftDataset`, `ftReference`).

See also: `TParam.Assign` (486), `TParam.AssignField` (486), `TParam.AssignToField` (486)

### 11.50.10 TParam.Clear

**Synopsis:** Clear the parameter value

**Declaration:** `procedure Clear`

**Visibility:** public

**Description:** `Clear` clears the parameter value, it is set to `UnAssigned`. The `Datatype`, parameter type or name are not touched.

See also: `TParam.Value` (497), `TParam.Name` (496), `TParam.ParamType` (496), `TParam.DataType` (495)

### 11.50.11 TParam.GetData

**Synopsis:** Get the parameter value from a memory buffer

**Declaration:** `procedure GetData(Buffer: Pointer)`

**Visibility:** public



**Description:** `GetData` retrieves the parameter value and stores it in `buffer`. It uses the same data layout as `TField` (421), and can be used to copy the parameter value to a record buffer.

**Errors:** Only basic field types are supported. Using an unsupported field type will result in an `EDatabaseError` (335) exception.

See also: `TParam.SetData` (489), `TField` (421)

### 11.50.12 TParam.GetDataSize

**Synopsis:** Return the size of the data.

**Declaration:** `function GetDataSize : Integer`

**Visibility:** `public`

**Description:** `GetDataSize` returns the size (in bytes) needed to store the current value of the parameter.

**Errors:** For an unsupported data type, an `EDatabaseError` (335) exception is raised when this function is called.

See also: `TParam.GetData` (487), `TParam.SetData` (489)

### 11.50.13 TParam.LoadFromFile

**Synopsis:** Load a parameter value from file

**Declaration:** `procedure LoadFromFile(const FileName: string; BlobType: TBlobType)`

**Visibility:** `public`

**Description:** `LoadFromFile` can be used to load a BLOB-type parameter from a file named `FileName`. The `BlobType` parameter can be used to set the exact data type of the parameter: it must be one of the BLOB data types. This function simply creates a `TFileStream` instance and passes it to `TParam.LoadFromStream` (488).

**Errors:** If the specified `FileName` is not a valid file, or the file is not readable, an exception will occur.

See also: `TParam.LoadFromStream` (488), `TBlobType` (319), `TParam.SaveToFile` (484)

### 11.50.14 TParam.LoadFromStream

**Synopsis:** Load a parameter value from stream

**Declaration:** `procedure LoadFromStream(Stream: TStream; BlobType: TBlobType)`

**Visibility:** `public`

**Description:** `LoadFromStream` can be used to load a BLOB-type parameter from a stream. The `BlobType` parameter can be used to set the exact data type of the parameter: it must be one of the BLOB data types.

**Errors:** If the stream does not support taking the `Size` of the stream, an exception will be raised.

See also: `TParam.LoadFromFile` (488), `TParam.SaveToStream` (484)

**11.50.15 TParam.SetBlobData**

Synopsis: Set BLOB data

Declaration: `procedure SetBlobData(Buffer: Pointer; ASize: Integer)`

Visibility: public

Description: `SetBlobData` reads the value of a BLOB type parameter from a memory buffer: the data is read from the memory buffer `Buffer` and is assumed to be `Size` bytes long.

Errors: No checking is performed on the validity of the data buffer. If the data buffer is invalid or the size is wrong, an exception may occur.

See also: `TParam.LoadFromStream` (488)

**11.50.16 TParam.SetData**

Synopsis: Set the parameter value from a buffer

Declaration: `procedure SetData(Buffer: Pointer)`

Visibility: public

Description: `SetData` performs the reverse operation of `TParam.GetData` (487): it reads the parameter value from the memory area pointed to by `Buffer`. The size of the data read is determined by `TParam.GetDataSize` (488) and the type of data by `TParam.DataType` (495): it is the same storage mechanism used by `TField` (421), and so can be used to copy the value from a `TDataset` (371) record buffer.

Errors: Not all field types are supported. If an unsupported field type is encountered, an `EDatabaseError` (335) exception is raised.

See also: `TDataset` (371), `TParam.GetData` (487), `TParam.DataType` (495), `TParam.GetDataSize` (488)

**11.50.17 TParam.AsBCD**

Synopsis: Get or set parameter value as BCD value

Declaration: `Property AsBCD : Currency`

Visibility: public

Access: Read,Write

Description: `AsBCD` can be used to get or set a parameter value as a BCD encoded floating point value.

See also: `TParam.AsFloat` (491)

**11.50.18 TParam.AsBlob**

Synopsis: Return parameter value as a blob

Declaration: `Property AsBlob : TBlobData`

Visibility: public

Access: Read,Write

Description: `AsBlob` returns the parameter value as a blob: currently this is a string. It can be set to set the parameter value.

See also: `TParam.AsString` (492)

### 11.50.19 TParam.AsBoolean

Synopsis: Get/Set parameter value as a boolean value

Declaration: `Property AsBoolean : Boolean`

Visibility: public

Access: Read,Write

Description: `AsBoolean` will return the parameter value as a boolean value. If it is written, the value is set to the specified value and the data type is set to `ftBoolean`.

See also: `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

### 11.50.20 TParam.AsBytes

Synopsis: Get or set parameter value as `TBytes`

Declaration: `Property AsBytes : TBytes`

Visibility: public

Access: Read,Write

Description: `AsBytes` can be used to get or set a parameter value as a `TBytes` value. This should normally only be used for blob type parameters.

See also: `TParam.AsString` ([492](#))

### 11.50.21 TParam.AsCurrency

Synopsis: Get/Set parameter value as a currency value

Declaration: `Property AsCurrency : Currency`

Visibility: public

Access: Read,Write

Description: `AsCurrency` will return the parameter value as a currency value. If it is written, the value is set to the specified value and the data type is set to `ftCurrency`.

See also: `TParam.AsFloat` ([491](#)), `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

### 11.50.22 TParam.AsDate

Synopsis: Get/Set parameter value as a date (`TDateTime`) value

Declaration: `Property AsDate : TDateTime`

Visibility: public

Access: Read,Write

Description: `AsDate` will return the parameter value as a date value. If it is written, the value is set to the specified value and the data type is set to `ftDate`.

See also: `TParam.AsDateTime` ([491](#)), `TParam.AsTime` ([493](#)), `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

**11.50.23 TParam.AsDateTime**

Synopsis: Get/Set parameter value as a date/time (TDateTime) value

Declaration: `Property AsDateTime : TDateTime`

Visibility: `public`

Access: Read,Write

Description: `AsDateTime` will return the parameter value as a `TDateTime` value. If it is written, the value is set to the specified value and the data type is set to `ftDateTime`.

See also: `TParam.AsDate` (490), `TParam.asTime` (493), `TParam.DataType` (495), `TParam.Value` (497)

**11.50.24 TParam.AsFloat**

Synopsis: Get/Set parameter value as a floating-point value

Declaration: `Property AsFloat : Double`

Visibility: `public`

Access: Read,Write

Description: `AsFloat` will return the parameter value as a double floating-point value. If it is written, the value is set to the specified value and the data type is set to `ftFloat`.

See also: `TParam.AsCurrency` (490), `TParam.DataType` (495), `TParam.Value` (497)

**11.50.25 TParam.AsInteger**

Synopsis: Get/Set parameter value as an integer (32-bit) value

Declaration: `Property AsInteger : LongInt`

Visibility: `public`

Access: Read,Write

Description: `AsInteger` will return the parameter value as a 32-bit signed integer value. If it is written, the value is set to the specified value and the data type is set to `ftInteger`.

See also: `TParam.AsLargeInt` (491), `TParam.AsSmallInt` (492), `TParam.AsWord` (493), `TParam.DataType` (495), `TParam.Value` (497)

**11.50.26 TParam.AsLargeInt**

Synopsis: Get/Set parameter value as a 64-bit integer value

Declaration: `Property AsLargeInt : LargeInt`

Visibility: `public`

Access: Read,Write

Description: `AsLargeInt` will return the parameter value as a 64-bit signed integer value. If it is written, the value is set to the specified value and the data type is set to `ftLargeInt`.

See also: `TParam.asInteger` (491), `TParam.asSmallint` (492), `TParam.AsWord` (493), `TParam.DataType` (495), `TParam.Value` (497)

**11.50.27 TParam.AsMemo**

Synopsis: Get/Set parameter value as a memo (string) value

Declaration: `Property AsMemo : string`

Visibility: public

Access: Read,Write

Description: `AsMemo` will return the parameter value as a memo (string) value. If it is written, the value is set to the specified value and the data type is set to `ftMemo`.

See also: `TParam.asString` ([492](#)), `TParam.LoadFromStream` ([488](#)), `TParam.SaveToStream` ([484](#)), `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

**11.50.28 TParam.AsSmallInt**

Synopsis: Get/Set parameter value as a smallint value

Declaration: `Property AsSmallInt : LongInt`

Visibility: public

Access: Read,Write

Description: `AsSmallInt` will return the parameter value as a 16-bit signed integer value. If it is written, the value is set to the specified value and the data type is set to `ftSmallInt`.

See also: `TParam.AsInteger` ([491](#)), `TParam.AsLargeInt` ([491](#)), `TParam.AsWord` ([493](#)), `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

**11.50.29 TParam.AsString**

Synopsis: Get/Set parameter value as a string value

Declaration: `Property AsString : string`

Visibility: public

Access: Read,Write

Description: `AsString` will return the parameter value as a string value. If it is written, the value is set to the specified value and the data type is set to `ftString`.

See also: `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

**11.50.30 TParam.AsAnsiString**

Synopsis: Parameter contents as an ANSI string

Declaration: `Property AsAnsiString : AnsiString`

Visibility: public

Access: Read,Write

Description: `AsAnsiString` returns the parameter data as an ANSI string (single byte character string). Note that if the parameter contains unicode data, some characters may get lost when reading.

See also: `TParam.AsString` ([492](#)), `TParam.AsUnicodeString` ([493](#)), `TParam.AsUTF8String` ([493](#))

**11.50.31 TParam.AsUTF8String**

Synopsis: Parameter contents as an UTF8 string

Declaration: `Property AsUTF8String : UTF8String`

Visibility: `public`

Access: `Read,Write`

Description: `AsUTF8String` returns the parameter data as an UTF8 string (single byte-encoded unicode string).

See also: `TParam.AsString` ([492](#)), `TParam.AsUnicodeString` ([493](#)), `TParam.AsAnsiString` ([492](#))

**11.50.32 TParam.AsUnicodeString**

Synopsis: Parameter contents as a Unicode string

Declaration: `Property AsUnicodeString : UnicodeString`

Visibility: `public`

Access: `Read,Write`

Description: `AsUnicodeString` returns the parameter data as a `UnicodeString` (double byte unicode string).

See also: `TParam.AsString` ([492](#)), `TParam.AsUTF8String` ([493](#)), `TParam.AsAnsiString` ([492](#))

**11.50.33 TParam.AsTime**

Synopsis: Get/Set parameter value as a time (`TDateTime`) value

Declaration: `Property AsTime : TDateTime`

Visibility: `public`

Access: `Read,Write`

Description: `AsTime` will return the parameter value as a time (`TDateTime`) value. If it is written, the value is set to the specified value and the data type is set to `ftTime`.

See also: `TParam.AsDate` ([490](#)), `TParam.AsDateTime` ([491](#)), `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

**11.50.34 TParam.AsWord**

Synopsis: Get/Set parameter value as a word value

Declaration: `Property AsWord : LongInt`

Visibility: `public`

Access: `Read,Write`

Description: `AsWord` will return the parameter value as an integer. If it is written, the value is set to the specified value and the data type is set to `ftWord`.

See also: `TParam.AsInteger` ([491](#)), `TParam.AsLargeInt` ([491](#)), `TParam.AsSmallint` ([492](#)), `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

### 11.50.35 TParam.AsFMTBCD

Synopsis: Parameter value as a BCD value

Declaration: `Property AsFMTBCD : TBCD`

Visibility: `public`

Access: `Read,Write`

Description: `AsFMTBCD` can be used to get or set the parameter's value as a BCD typed value.

See also: `AsFloat` ([316](#)), `AsCurrency` ([316](#))

### 11.50.36 TParam.Bound

Synopsis: Is the parameter value bound (set to fixed value)

Declaration: `Property Bound : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `Bound` indicates whether a parameter has received a fixed value: setting the parameter value will set `Bound` to `True`. When creating master-detail relationships, parameters with their `Bound` property set to `True` will not receive a value from the master dataset: their value will be kept. Only parameters where `Bound` is `False` will receive a new value from the master dataset.

See also: `TParam.DataType` ([495](#)), `TParam.Value` ([497](#))

### 11.50.37 TParam.Dataset

Synopsis: Dataset to which this parameter belongs

Declaration: `Property Dataset : TDataSet`

Visibility: `public`

Access: `Read`

Description: `Dataset` is the dataset that owns the `TParams` ([497](#)) instance of which this `TParam` instance is a part. It is `Nil` if the collection is not set, or is not a `TParams` instance.

See also: `TDataSet` ([371](#)), `TParams` ([497](#))

### 11.50.38 TParam.IsNull

Synopsis: Is the parameter empty

Declaration: `Property IsNull : Boolean`

Visibility: `public`

Access: `Read`

Description: `IsNull` is `True` if the value is empty or not set (`Null` or `UnAssigned`).

See also: `TParam.Clear` ([487](#)), `TParam.Value` ([497](#))

### 11.50.39 TParam.NativeStr

Synopsis: No description available

Declaration: `Property NativeStr : string`

Visibility: public

Access: Read,Write

Description: No description available

### 11.50.40 TParam.Text

Synopsis: Read or write the value of the parameter as a string

Declaration: `Property Text : string`

Visibility: public

Access: Read,Write

Description: `AsText` returns the same value as `TParam.AsString` ([492](#)), but, when written, does not set the data type: instead, it attempts to convert the value to the type specified in `TParam.DataType` ([495](#)).

See also: `TParam.AsString` ([492](#)), `TParam.DataType` ([495](#))

### 11.50.41 TParam.AsWideString

Synopsis: Get/Set the value as a widestring

Declaration: `Property AsWideString : WideString`

Visibility: public

Access: Read,Write

Description: `AsWideString` returns the parameter value as a widestring value. Setting the property will set the value of the parameter and will also set the `DataType` ([495](#)) to `ftWideString`.

See also: `TParam.AsString` ([492](#)), `TParam.Value` ([497](#)), `TParam.DataType` ([495](#))

### 11.50.42 TParam.DataType

Synopsis: Data type of the parameter

Declaration: `Property DataType : TFieldType`

Visibility: published

Access: Read,Write

Description: `DataType` is the current data type of the parameter value. It is set automatically when one of the various `AsXYZ` properties is written, or when the value is copied from a field value.

See also: `TParam.IsNull` ([494](#)), `TParam.Value` ([497](#)), `TParam.AssignField` ([486](#))



**11.50.43 TParam.Name**

Synopsis: Name of the parameter

Declaration: `Property Name : string`

Visibility: published

Access: Read,Write

Description: `Name` is the name of the parameter. The name is usually determined automatically from the SQL statement the parameter is part of. Each parameter name should appear only once in the collection.

See also: `TParam.DataType` (495), `TParam.Value` (497), `TParams.ParamByName` (500)

**11.50.44 TParam.NumericScale**

Synopsis: Numeric scale

Declaration: `Property NumericScale : Integer`

Visibility: published

Access: Read,Write

Description: `NumericScale` can be used to store the numerical scale for BCD values. It is currently unused.

See also: `TParam.Precision` (497), `TParam.Size` (497)

**11.50.45 TParam.ParamType**

Synopsis: Type of parameter

Declaration: `Property ParamType : TParamType`

Visibility: published

Access: Read,Write

Description: `ParamType` specifies the type of parameter: is the parameter value written to the database engine, or is it received from the database engine, or both ? It can have the following value:

**ptUnknown**Unknown type

**ptInput**Input parameter

**ptOutput**Output parameter, filled on result

**ptInputOutput**Input/output parameter

**ptResult**Result parameter

The `ParamType` property is usually set by the database engine that creates the parameter instances.

See also: `TParam.DataType` (495), `TParam.DataSize` (484), `TParam.Name` (496)

**11.50.46 TParam.Precision**

Synopsis: Precision of the BCD value

Declaration: `Property Precision : Integer`

Visibility: published

Access: Read,Write

Description: `Precision` can be used to store the numerical precision for BCD values. It is currently unused.

See also: `TParam.NumericScale` ([496](#)), `TParam.Size` ([497](#))

**11.50.47 TParam.Size**

Synopsis: Size of the parameter

Declaration: `Property Size : Integer`

Visibility: published

Access: Read,Write

Description: `Size` is the declared size of the parameter. In the current implementation, this parameter is ignored other than copying it from `TField.DataSize` ([435](#)) in the `TParam.AssignFieldValue` ([487](#)) method. The actual size can be retrieved through the `TParam.Datasize` ([484](#)) property.

See also: `TParam.Datasize` ([484](#)), `TField.DataSize` ([435](#)), `TParam.AssignFieldValue` ([487](#))

**11.50.48 TParam.Value**

Synopsis: Value as a variant

Declaration: `Property Value : Variant`

Visibility: published

Access: Read,Write

Description: `Value` returns (or sets) the value as a variant value.

See also: `TParam.DataType` ([495](#))

**11.51 TParams****11.51.1 Description**

`TParams` is a collection of `TParam` ([484](#)) values. It is used to specify parameter values for parameterized SQL statements, but is also used to specify parameter values for stored procedures. Its default property is an array of `TParam` ([484](#)) values. The class also offers a method to scan a SQL statement for parameter names and replace them with placeholders understood by the SQL engine: `TParams.ParseSQL` ([500](#)).

`TDataset` ([371](#)) itself does not use `TParams`. The class is provided in the `DB` unit, so all `TDataset` descendents that need some kind of parameterization make use of the same interface. The `TMasterParamsDataLink` ([479](#)) class can be used to establish a master-detail relationship between a parameter-aware `TDataset` instance and another dataset; it will automatically refresh parameter values when the fields in the master dataset change. To this end, the `TParams.CopyParamValuesFromDataset` ([502](#)) method exists.

See also: TDataSet ([371](#)), TMasterParamsDataLink ([479](#)), TParam ([484](#)), TParams.ParseSQL ([500](#)), TParams.CopyParamValuesFromDataset ([502](#))

### 11.51.2 Method overview

Page	Method	Description
<a href="#">498</a>	AddParam	Add a parameter to the collection
<a href="#">499</a>	AssignValues	Copy values from another collection
<a href="#">502</a>	CopyParamValuesFromDataset	Copy parameter values from the fields in a dataset.
<a href="#">498</a>	Create	Create a new instance of TParams
<a href="#">499</a>	CreateParam	Create and add a new parameter to the collection
<a href="#">499</a>	FindParam	Find a parameter with given name
<a href="#">500</a>	GetEnumerator	Return an enumerator for the parameters
<a href="#">499</a>	GetParamList	Fetch a list of TParam instances
<a href="#">500</a>	IsEqual	Is the list of parameters equal
<a href="#">500</a>	ParamByName	Return a parameter by name
<a href="#">500</a>	ParseSQL	Parse SQL statement, replacing parameter names with SQL parameter placeholders
<a href="#">501</a>	RemoveParam	Remove a parameter from the collection

### 11.51.3 Property overview

Page	Properties	Access	Description
<a href="#">502</a>	Dataset	r	Dataset that owns the TParams instance
<a href="#">502</a>	Items	rw	Indexed access to TParams instances in the collection
<a href="#">502</a>	ParamValues	rw	Named access to the parameter values.

### 11.51.4 TParams.Create

Synopsis: Create a new instance of TParams

Declaration: `constructor Create(AOwner: TPersistent;  
                                  AItemClass: TCollectionItemClass); Overload  
constructor Create(AOwner: TPersistent); Overload  
constructor Create; Overload`

Visibility: public

Description: Create initializes a new instance of TParams. It calls the inherited constructor with TParam ([484](#)) as the collection's item class, and sets AOwner as the owner of the collection. Usually, AOwner will be the dataset that needs parameters.

See also: #rtl.classes.TCollection.create (??), TParam ([484](#))

### 11.51.5 TParams.AddParam

Synopsis: Add a parameter to the collection

Declaration: `procedure AddParam(Value: TParam)`

Visibility: public

Description: AddParam adds Value to the collection.

Errors: No checks are done on the TParam instance. If it is Nil, an exception will be raised.

See also: [TParam \(484\)](#), [#rtl.classes.tcollection.add \(??\)](#)

### 11.51.6 TParams.AssignValues

Synopsis: Copy values from another collection

Declaration: `procedure AssignValues(Value: TParams)`

Visibility: public

Description: `AssignValues` examines all [TParam \(484\)](#) instances in `Value`, and looks in its own items for a `TParam` instance with the same name. If it is found, then the value and type of the parameter are copied (using [TParam.Assign \(486\)](#)). If it is not found, nothing is done.

See also: [TParam \(484\)](#), [TParam.Assign \(486\)](#)

### 11.51.7 TParams.CreateParam

Synopsis: Create and add a new parameter to the collection

Declaration: `function CreateParam(FldType: TFieldType; const ParamName: string; ParamType: TParamType) : TParam`

Visibility: public

Description: `CreateParam` creates a new [TParam \(484\)](#) instance with datatype equal to `fldType`, Name equal to `ParamName` and sets its `ParamType` property to `ParamType`. The parameter is then added to the collection.

See also: [TParam \(484\)](#), [TParam.Name \(496\)](#), [TParam.Datatype \(495\)](#), [TParam.Paramtype \(496\)](#)

### 11.51.8 TParams.FindParam

Synopsis: Find a parameter with given name

Declaration: `function FindParam(const Value: string) : TParam`

Visibility: public

Description: `FindParam` searches the collection for the [TParam \(484\)](#) instance with property `Name` equal to `Value`. It will return the last instance with the given name, and will only return one instance. If no match is found, `Nil` is returned.

**Remark** A `TParams` collection can have 2 `TParam` instances with the same name: no checking for duplicates is done.

See also: [TParam.Name \(496\)](#), [TParams.ParamByName \(500\)](#), [TParams.GetParamList \(499\)](#)

### 11.51.9 TParams.GetParamList

Synopsis: Fetch a list of `TParam` instances

Declaration: `procedure GetParamList(List: TList; const ParamNames: string)`

Visibility: public

Description: `GetParamList` examines the parameter names in the semicolon-separated list `ParamNames`. It searches each `TParam` instance from the names in the list and adds it to `List`.

**Errors:** If the `ParamNames` list contains an unknown parameter name, then an exception is raised. Whitespace is not discarded.

See also: `TParam` ([484](#)), `TParam.Name` ([496](#)), `TParams.ParamByName` ([500](#))

### 11.51.10 TParams.IsEqual

**Synopsis:** Is the list of parameters equal

**Declaration:** `function IsEqual(Value: TParams) : Boolean`

**Visibility:** public

**Description:** `IsEqual` compares the parameter count of `Value` and if it matches, it compares all `TParam` items of `Value` with the items it owns. If all items are equal (all properties match), then `True` is returned. The items are compared on index, so the order is important.

See also: `TParam` ([484](#))

### 11.51.11 TParams.GetEnumerator

**Synopsis:** Return an enumerator for the parameters

**Declaration:** `function GetEnumerator : TParamsEnumerator`

**Visibility:** public

**Description:** `GetEnumerator` returns an enumerator that loops over all parameters (as implemented by `TParamsEnumerator` ([503](#)))

See also: `TParamsEnumerator` ([503](#))

### 11.51.12 TParams.ParamByName

**Synopsis:** Return a parameter by name

**Declaration:** `function ParamByName(const Value: string) : TParam`

**Visibility:** public

**Description:** `ParamByName` searches the collection for the `TParam` ([484](#)) instance with property `Name` equal to `Value`. It will return the last instance with the given name, and will only return one instance. If no match is found, an exception is raised.

**Remark** A `TParams` collection can have 2 `TParam` instances with the same name: no checking for duplicates is done.

See also: `TParam.Name` ([496](#)), `TParams.FindParam` ([499](#)), `TParams.GetParamList` ([499](#))

### 11.51.13 TParams.ParseSQL

**Synopsis:** Parse SQL statement, replacing parameter names with SQL parameter placeholders

**Declaration:** `function ParseSQL(SQL: string; DoCreate: Boolean) : string; Overload  
function ParseSQL(SQL: string; DoCreate: Boolean; EscapeSlash: Boolean;  
EscapeRepeat: Boolean; ParameterStyle: TParamStyle)  
: string; Overload`

```
function ParseSQL(SQL: string; DoCreate: Boolean; EscapeSlash: Boolean;
    EscapeRepeat: Boolean; ParameterStyle: TParamStyle;
    out ParamBinding: TParamBinding) : string; Overload
function ParseSQL(SQL: string; DoCreate: Boolean; EscapeSlash: Boolean;
    EscapeRepeat: Boolean; ParameterStyle: TParamStyle;
    out ParamBinding: TParamBinding;
    out ReplaceString: string) : string; Overload
```

Visibility: public

**Description:** ParseSQL parses the SQL statement for parameter names in the form :ParamName. It replaces them with a SQL parameter placeholder. If DoCreate is True then a TParam instance is added to the collection with the found parameter name.

The parameter placeholder is determined by the ParameterStyle property, which can have the following values:

**psInterbaseParameters** are specified by a ? character

**psPostgreSQLParameters** are specified by a \$N character.

**psSimulatedParameters** are specified by a \$N character.

psInterbase is the default.

If the EscapeSlash parameter is True, then backslash characters are used to quote the next character in the SQL statement. If it is False, the backslash character is regarded as a normal character.

If the EscapeRepeat parameter is True (the default) then embedded quotes in string literals are escaped by repeating themselves. If it is false then they should be quoted with backslashes.

ParamBinding, if specified, is filled with the indexes of the parameter instances in the parameter collection: for each SQL parameter placeholder, the index of the corresponding TParam instance is returned in the array.

ReplaceString, if specified, contains the placeholder used for the parameter names (by default, \$). It has effect only when ParameterStyle equals psSimulated.

The function returns the SQL statement with the parameter names replaced by placeholders.

See also: TParam ([484](#)), TParam.Name ([496](#)), TParamStyle ([328](#))

### 11.51.14 TParams.RemoveParam

**Synopsis:** Remove a parameter from the collection

**Declaration:** procedure RemoveParam(Value: TParam)

Visibility: public

**Description:** RemoveParam removes the parameter Value from the collection, but does not free the instance.

**Errors:** Value must be a valid instance, or an exception will be raised.

See also: TParam ([484](#))

**11.51.15 TParams.CopyParamValuesFromDataset**

Synopsis: Copy parameter values from the fields in a dataset.

Declaration: `procedure CopyParamValuesFromDataset (ADataset: TDataSet;  
CopyBound: Boolean)`

Visibility: public

Description: `CopyParamValuesFromDataset` assigns values to all parameters in the collection by searching in `ADataset` for fields with the same name, and assigning the value of the field to the `TParam` instances using `TParam.AssignField` (486). By default, this operation is only performed on `TParam` instances with their `Bound` (494) property set to `False`. If `CopyBound` is true, then the operation is performed on all `TParam` instances in the collection.

Errors: If, for some `TParam` instance, `ADataset` misses a field with the same name, an `EDatabaseError` exception will be raised.

See also: `TParam` (484), `TParam.Bound` (494), `TParam.AssignField` (486), `TDataSet` (371), `TDataSet.FieldName` (382)

**11.51.16 TParams.Dataset**

Synopsis: Dataset that owns the `TParams` instance

Declaration: `Property Dataset : TDataSet`

Visibility: public

Access: Read

Description: `Dataset` is the `TDataSet` (371) instance that was specified when the `TParams` instance was created.

See also: `TParams.Create` (498), `TDataSet` (371)

**11.51.17 TParams.Items**

Synopsis: Indexed access to `TParams` instances in the collection

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

Visibility: public

Access: Read,Write

Description: `Items` is overridden by `TParams` so it has the proper type (`TParam`). The `Index` runs from 0 to `Count-1`.

See also: `TParams` (497)

**11.51.18 TParams.ParamValues**

Synopsis: Named access to the parameter values.

Declaration: `Property ParamValues[ParamName: string]: Variant`

Visibility: public

Access: Read,Write

Description: `ParamValues` provides access to the parameter values (`TParam.Value` (497)) by name. It is equivalent to reading and writing

```
ParamByName (ParamName) .Value
```

See also: `TParam.Value` (497), `TParams.ParamByName` (500)

## 11.52 TParamsEnumerator

### 11.52.1 Description

`TParamsEnumerator` is a helper class to implement enumeration (`for..in`) of parameters. It implements the `IEnumerator` interface.

See also: `TParams.GetEnumerator` (500)

### 11.52.2 Method overview

Page	Method	Description
503	<code>Create</code>	Create a new <code>TParamsEnumerator</code> instance
503	<code>MoveNext</code>	Go to next <code>TParam</code>

### 11.52.3 Property overview

Page	Properties	Access	Description
504	<code>Current</code>	<code>r</code>	Current <code>TParam</code> instance

### 11.52.4 TParamsEnumerator.Create

Synopsis: Create a new `TParamsEnumerator` instance

Declaration: `constructor Create (AParams: TParams)`

Visibility: `public`

Description: `Create` instantiates a new enumerator for `AParams`.

See also: `TParams.GetEnumerator` (500)

### 11.52.5 TParamsEnumerator.MoveNext

Synopsis: Go to next `TParam`

Declaration: `function MoveNext : Boolean`

Visibility: `public`

Description: `MoveNext` will move to the next `TParam` instance if possible. If it returns `True` then `TParamsEnumerator.Current` (504) will return the new current `TParam`

See also: `TParamsEnumerator.Current` (504)



### 11.52.6 TParamsEnumerator.Current

Synopsis: Current TParam instance

Declaration: `Property Current : TParam`

Visibility: public

Access: Read

Description: `Current` is the current TParam instance. It is only valid if `TParamsEnumerator.MoveNext` ([503](#)) returned true.

See also: `TParamsEnumerator.MoveNext` ([503](#))

## 11.53 TSmallIntField

### 11.53.1 Description

`TSmallIntField` is the class created when a dataset must manage 16-bit signed integer data, of datatype `ftSmallInt`. It exposes no new properties, but simply overrides some methods to manage 16-bit signed integer data.

It should never be necessary to create an instance of `TSmallIntField` manually, a field of this class will be instantiated automatically for each smallint field when a dataset is opened.

See also: `TField` ([421](#)), `TNumericField` ([483](#)), `TLongIntField` ([474](#)), `TWordField` ([511](#))

### 11.53.2 Method overview

Page	Method	Description
<a href="#">504</a>	Create	Create a new instance of the <code>TSmallIntField</code> class.

### 11.53.3 TSmallIntField.Create

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

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` initializes a new instance of the `TSmallIntField` ([504](#)) class. It calls the inherited constructor and then simply sets some of the `TField` ([421](#)) properties to work with 16-bit signed integer data.

See also: `TField` ([421](#))

## 11.54 TStringField

### 11.54.1 Description

`TStringField` is the class used whenever a dataset has to handle a string field type (data type `ftString`). This class overrides some of the standard `TField` ([421](#)) methods to handle string data, and introduces some properties that are only pertinent for data fields of string type. It should never be necessary to create an instance of `TStringField` manually, a field of this class will be instantiated automatically for each string field when a dataset is opened.

See also: TField ([421](#)), TWideStringField ([510](#)), TDataset ([371](#))

### 11.54.2 Method overview

Page	Method	Description
<a href="#">505</a>	Create	Create a new instance of the TStringField class
<a href="#">505</a>	SetFieldType	Set the field type

### 11.54.3 Property overview

Page	Properties	Access	Description
<a href="#">505</a>	CodePage	r	Codepage of the field string data
<a href="#">506</a>	EditMask		Specify an edit mask for an edit control
<a href="#">506</a>	FixedChar	rw	Is the string declared with a fixed length ?
<a href="#">507</a>	Size		Maximum size of the string
<a href="#">506</a>	Transliterate	rw	Should the field value be transliterated when reading or writing
<a href="#">506</a>	Value	rw	Value of the field as a string

### 11.54.4 TStringField.Create

Synopsis: Create a new instance of the TStringField class

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: Create is used to create a new instance of the TStringField class. It initializes some TField ([421](#)) properties after having called the inherited constructor.

### 11.54.5 TStringField.SetFieldType

Synopsis: Set the field type

Declaration: `procedure SetFieldType(AValue: TFieldType); Override`

Visibility: public

Description: SetFieldType is overridden in TStringField ([504](#)) to check the data type more accurately (ftString and ftFixedChar). No extra functionality is added.

See also: TField.DataType ([435](#))

### 11.54.6 TStringField.CodePage

Synopsis: Codepage of the field string data

Declaration: `Property CodePage : TSystemCodePage`

Visibility: public

Access: Read

Description: CodePage is the code page of the string data in the field. It is determined when the field is initially created from the dataset's data, and cannot be changed while the dataset is active.

See also: TField.AsString ([431](#)), TField.AsUnicodeString ([432](#)), TField.AsAnsi8String ([421](#)), TFieldDef.CodePage ([450](#))

### 11.54.7 TStringField.FixedChar

Synopsis: Is the string declared with a fixed length ?

Declaration: `Property FixedChar : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `FixedChar` is `True` if the underlying data engine has declared the field with a fixed length, as in a `SQL CHAR()` declaration: the field's value will then always be padded with as many spaces as needed to obtain the declared length of the field. If it is `False` then the declared length is simply the maximum length for the field, and no padding with spaces is performed.

### 11.54.8 TStringField.Transliterate

Synopsis: Should the field value be transliterated when reading or writing

Declaration: `Property Transliterate : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `Transliterate` can be set to `True` if the field's contents should be transliterated prior to copying it from or to the field's buffer. Transliteration is done by a method of `TDataset`: `TDataset.Translate` (391).

See also: `TDataset.Translate` (391)

### 11.54.9 TStringField.Value

Synopsis: Value of the field as a string

Declaration: `Property Value : string`

Visibility: `public`

Access: `Read,Write`

Description: `Value` is overridden in `TField` to return the value of the field as a string. It returns the contents of `TField.AsString` (431) when read, or sets the `AsString` property when written to.

See also: `TField.AsString` (431), `TField.Value` (439)

### 11.54.10 TStringField.EditMask

Synopsis: Specify an edit mask for an edit control

Declaration: `Property EditMask :`

Visibility: `published`

Access:

**Description:** `EditMask` can be used to specify an edit mask for controls that allow to edit this field. It has no effect on the field value, and serves only to ensure that the user can enter only correct data for this field.

`TStringField` just changes the visibility of the `EditMask` property, it is introduced in `TField`.

For more information on valid edit masks, see the documentation of the GUI controls.

See also: `TField.EditMask` ([436](#))

### 11.54.11 TStringField.Size

**Synopsis:** Maximum size of the string

**Declaration:** `Property Size :`

**Visibility:** published

**Access:**

**Description:** `Size` is made published by the `TStringField` class so it can be set in the IDE: it is the declared maximum size of the string (in characters) and is used to calculate the size of the dataset buffer.

See also: `TField.Size` ([438](#))

## 11.55 TTimeField

### 11.55.1 Description

`TimeField` is the class used when a dataset must manage data of type time. (`TField.DataType` ([435](#)) equals `ftTime`). It initializes some of the properties of the `TField` ([421](#)) class to be able to work with time fields.

It should never be necessary to create an instance of `TTimeField` manually, a field of this class will be instantiated automatically for each time field when a dataset is opened.

See also: `TDataset` ([371](#)), `TField` ([421](#)), `TDateTimeField` ([414](#)), `TDateField` ([413](#))

### 11.55.2 Method overview

Page	Method	Description
<a href="#">507</a>	<code>Create</code>	Create a new instance of a <code>TTimeField</code> class.

### 11.55.3 TTimeField.Create

**Synopsis:** Create a new instance of a `TTimeField` class.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** public

**Description:** `Create` initializes a new instance of the `TTimeField` class. It calls the inherited destructor, and then sets some `TField` ([421](#)) properties to configure the instance for working with time values.

See also: `TField` ([421](#))

## 11.56 TVarBytesField

### 11.56.1 Description

`TVarBytesField` is the class used when a dataset must manage data of variable-size binary type. (`TField.DataType` (435) equals `ftVarBytes`). It initializes some of the properties of the `TField` (421) class to be able to work with variable-size byte fields.

It should never be necessary to create an instance of `TVarBytesField` manually, a field of this class will be instantiated automatically for each variable-sized binary data field when a dataset is opened.

See also: `TDataset` (371), `TField` (421), `TBytesField` (353)

### 11.56.2 Method overview

Page	Method	Description
<a href="#">508</a>	Create	Create a new instance of a <code>TVarBytesField</code> class.

### 11.56.3 TVarBytesField.Create

Synopsis: Create a new instance of a `TVarBytesField` class.

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` initializes a new instance of the `TVarBytesField` class. It calls the inherited destructor, and then sets some `TField` (421) properties to configure the instance for working with variable-size binary data values.

See also: `TField` (421)

## 11.57 TVariantField

### 11.57.1 Description

`TVariantField` is the class used when a dataset must manage native variant-typed data. (`TField.DataType` (435) equals `ftVariant`). It initializes some of the properties of the `TField` (421) class and overrides some of its methods to be able to work with variant data.

It should never be necessary to create an instance of `TVariantField` manually, a field of this class will be instantiated automatically for each variant field when a dataset is opened.

See also: `TDataset` (371), `TField` (421)

### 11.57.2 Method overview

Page	Method	Description
<a href="#">509</a>	Create	Create a new instance of the <code>TVariantField</code> class

### 11.57.3 TVariantField.Create

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

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** `public`

**Description:** `Create` initializes a new instance of the `TVariantField` class. It calls the inherited destructor, and then sets some `TField` (421) properties to configure the instance for working with variant values.

See also: `TField` (421)

## 11.58 TWideMemoField

### 11.58.1 Description

`TWideMemoField` is the class used when a dataset must manage memo (Text BLOB) data. (`TField.DataType` (435) equals `ftWideMemo`). It initializes some of the properties of the `TField` (421) class. All methods to be able to work with widestring memo fields have been implemented in the `TBlobField` (348) parent class.

It should never be necessary to create an instance of `TWideMemoField` manually, a field of this class will be instantiated automatically for each widestring memo field when a dataset is opened.

See also: `TDataset` (371), `TField` (421), `TBLOBField` (348), `TMemoField` (481), `TGraphicField` (465)

### 11.58.2 Method overview

Page	Method	Description
509	<code>Create</code>	Create a new instance of the <code>TWideMemoField</code> class

### 11.58.3 Property overview

Page	Properties	Access	Description
510	<code>Value</code>	<code>rw</code>	Value of the field's contents as a widestring

### 11.58.4 TWideMemoField.Create

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

**Declaration:** `constructor Create(aOwner: TComponent); Override`

**Visibility:** `public`

**Description:** `Create` initializes a new instance of the `TWideMemoField` class. It calls the inherited destructor, and then sets some `TField` (421) properties to configure the instance for working with widestring memo values.

See also: `TField` (421)

### 11.58.5 TWideMemoField.Value

Synopsis: Value of the field's contents as a widestring

Declaration: `Property Value : WideString`

Visibility: `public`

Access: `Read, Write`

Description: `Value` is redefined by `TWideMemoField` as a `WideString` value. Reading and writing this property is equivalent to reading and writing the `TField.AsWideString` (433) property.

See also: `TField.Value` (439), `TField.AsWideString` (433)

## 11.59 TWideStringField

### 11.59.1 Description

`TWideStringField` is the string field class instantiated for fields of data type `ftWideString`. This class overrides some of the standard `TField` (421) methods to handle widestring data, and introduces some properties that are only pertinent for data fields of widestring type. It should never be necessary to create an instance of `TWideStringField` manually, a field of this class will be instantiated automatically for each widestring field when a dataset is opened.

See also: `TField` (421), `TStringField` (504), `TDataset` (371)

### 11.59.2 Method overview

Page	Method	Description
<a href="#">510</a>	<code>Create</code>	Create a new instance of the <code>TWideStringField</code> class.
<a href="#">510</a>	<code>SetFieldType</code>	Set the field type

### 11.59.3 Property overview

Page	Properties	Access	Description
<a href="#">511</a>	<code>Value</code>	<code>rw</code>	Value of the field as a widestring

### 11.59.4 TWideStringField.Create

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

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` is used to create a new instance of the `TWideStringField` class. It initializes some `TField` (421) properties after having called the inherited constructor.

### 11.59.5 TWideStringField.SetFieldType

Synopsis: Set the field type

Declaration: `procedure SetFieldType(AValue: TFieldType); Override`

Visibility: public

Description: `SetFieldType` is overridden in `TWideStringField` (510) to check the data type more accurately (`ftWideString` and `ftFixedWideChar`). No extra functionality is added.

See also: `TField.DataType` (435)

### 11.59.6 TWideStringField.Value

Synopsis: Value of the field as a widestring

Declaration: `Property Value : WideString`

Visibility: public

Access: Read,Write

Description: `Value` is overridden by the `TWideStringField` to return a `WideString` value. It is the same value as the `TField.AsWideString` (433) property.

See also: `TField.AsWideString` (433), `TField.Value` (439)

## 11.60 TWordField

### 11.60.1 Description

`TWordField` is the class created when a dataset must manage 16-bit unsigned integer data, of datatype `ftWord`. It exposes no new properties, but simply overrides some methods to manage 16-bit unsigned integer data.

It should never be necessary to create an instance of `TWordField` manually, a field of this class will be instantiated automatically for each word field when a dataset is opened.

See also: `TField` (421), `TNumericField` (483), `TLongintField` (474), `TSmallIntField` (504)

### 11.60.2 Method overview

Page	Method	Description
511	Create	Create a new instance of the <code>TWordField</code> class.

### 11.60.3 TWordField.Create

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

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` initializes a new instance of the `TWordField` (511) class. It calls the inherited constructor and then simply sets some of the `TField` (421) properties to work with 16-bit unsigned integer data.

See also: `TField` (421)



## Chapter 12

# Reference for unit 'dbugintf'

### 12.1 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 transparent in its use: it does not need initialization, it will start the debug server by itself if it can find it: the program should be called `debugserver` and should be in the `PATH`. When the first debug message is sent, the unit will initialize itself.

The FCL contains a sample debug server (`dbugsvr`) which can be started in advance, and which writes debug message to the console (both on Windows and Linux). The Lazarus project contains a visual application which displays the messages in a GUI.

The `dbugintf` unit relies on the SimpleIPC (512) 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 (512) unit should also be functional.

### 12.2 Writing a debug server

Writing a debug server is relatively easy. It should instantiate a `TSimpleIPCServer` class from the SimpleIPC (512) 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 (512) 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 (512) unit should also be functional.

### 12.3 Constants, types and variables

#### 12.3.1 Resource strings

```
SEntering = '> Entering '
```

String used when sending method enter message.

```
SExiting = '< Exiting '
```

String used when sending method exit message.

`SProcessID = 'Process %s'`

String used when sending identification message to the server.

`SSeparator = '>-----<'`

String used when sending a separator line.

`SServerStartFailed = 'Failed to start debugserver. (%s)'`

String used to display an error message when the start of the debug server failed

### 12.3.2 Constants

`DefaultDebugServer = 'debugserver'`

`DefaultDebugServer` is the name at which the default debug server can be reached.

`SendError : string = ''`

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

### 12.3.3 Types

`TDebugLevel = (dlInformation, dlWarning, dlError)`

Table 12.1: 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.

### 12.3.4 Variables

`DebugServerExe : string = DefaultDebugServer`

`DefaultDebugServerExe` is the filename for default debug server executable.

## 12.4 Procedures and functions

### 12.4.1 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` (518), `SendDebug` (515)

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

**Errors:** None.

**See also:** `SendDebug` (515), `StartDebugServer` (518)

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

**Errors:** None.

**See also:** `SendDebug` (515), `SendDateTime` (514), `SendInteger` (516), `SendPointer` (517)

### 12.4.4 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` (515) 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` (515), `SendBoolean` (514), `SendInteger` (516), `SendPointer` (517)

### 12.4.5 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` (513).

See also: `SendDebugEx` (515), `SendDebugFmt` (515), `SendDebugFmtEx` (516)

### 12.4.6 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` (515) uses informational messages.

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

Errors: None.

See also: `SendDebug` (515), `SendDebugFmt` (515), `SendDebugFmtEx` (516)

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

Errors: None.

See also: `SendDebug` (515), `SendDebugEx` (515), `SendDebugFmtEx` (516), `#rtl.sysutils.format` (??)

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

Errors: None.

See also: `SendDebug` (515), `SendDebugEx` (515), `SendDebugFmt` (515), `#rtl.sysutils.format` (??)

### 12.4.9 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` (515) 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` (515), `SendBoolean` (514), `SendDateTime` (514), `SendPointer` (517)

### 12.4.10 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` (517), 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` (515), `SendMethodExit` (517), `SendSeparator` (517)

### 12.4.11 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` (516), 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` (515), `SendMethodEnter` (516), `SendSeparator` (517)

### 12.4.12 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` (515) 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` (515), `SendBoolean` (514), `SendDateTime` (514), `SendInteger` (516)

### 12.4.13 SendSeparator

Synopsis: Send a separator message

Declaration: `procedure SendSeparator`

Visibility: default

Description: `SendSeparator` is a simple wrapper around `SendDebug` (515) 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` (515), `SendMethodEnter` (516), `SendMethodExit` (517)

### 12.4.14 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` (515) 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` (513), `SendDebug` (515)

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

Errors: On error, `False` is returned.

See also: `SendDebug` (515), `InitDebugClient` (514)

## Chapter 13

# Reference for unit 'dbugmsg'

### 13.1 Used units

Table 13.1: Used units by unit 'dbugmsg'

Name	Page
Classes	??

### 13.2 Overview

`dbugmsg` is an auxiliary unit used in the `dbugintf` ([512](#)) unit. It defines the message protocol used between the debug unit and the debug server.

### 13.3 Constants, types and variables

#### 13.3.1 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.

### 13.3.2 Types

## 13.4 Procedures and functions

### 13.4.1 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` (521) 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.

### 13.4.2 ReadDebugMessageFromStream

Synopsis: Read a message from stream

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

Visibility: default

Description: `ReadDebugMessageFromStream` reads a `TDebugMessage` (521) 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` (521), `WriteDebugMessageToStream` (520)

### 13.4.3 WriteDebugMessageToStream

Synopsis: Write a message to stream

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

Visibility: default

**Description:** `WriteDebugMessageFromStream` writes a `TDebugMessage` (521) 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` (521), `ReadDebugMessageFromStream` (520)

## 13.5 TDebugMessage

```
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` (520) and `ReadDebugMessageFromStream` (520) can be used to read or write the message from/to a stream.

# Chapter 14

## Reference for unit 'eventlog'

### 14.1 Used units

Table 14.1: Used units by unit 'eventlog'

Name	Page
Classes	??
sysutils	??

### 14.2 Overview

The EventLog unit implements the TEventLog ([524](#)) 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.

### 14.3 Constants, types and variables

#### 14.3.1 Resource strings

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

Message used to format an error when an error exception is raised.

`SLogCustom = 'Custom (%d)'`

Custom message formatting string

`SLogDebug = 'Debug'`

Debug message name

`SLogError = 'Error'`

Error message name

`SLogInfo = 'Info'`

Informational message name

SLogWarning = 'Warning'

Warning message name

### 14.3.2 Types

TLogCategoryEvent = procedure(Sender: TObject; var Code: Word) of object

TLogCategoryEvent is the event type for the TEventLog.OnGetCustomCategory (531) 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 (531) and OnGetCustomEventID (531) 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 14.2: 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 (528) 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.

## 14.4 ELogError

### 14.4.1 Description

ELogError is the exception used in the TEventLog (524) component to indicate errors.

See also: TEventLog (524)

## 14.5 TEventLog

### 14.5.1 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 (526) call, or the specialized Warning (527), Error (527), Info (528) or Debug (527) calls, which have the event type predefined.

See also: Log (526), Warning (527), Error (527), Info (528), Debug (527)

### 14.5.2 Method overview

Page	Method	Description
527	Debug	Log a debug message
524	Destroy	Clean up TEventLog instance
527	Error	Log an error message to
525	EventTypeToString	Create a string representation of an event type
528	Info	Log an informational message
526	Log	Log a message to the system log.
526	Pause	Pause the sending of log messages.
525	RegisterMessageFile	Register message file
526	Resume	Resume sending of log messages if sending was paused
526	UnRegisterMessageFile	Unregister the message file (needed on windows only)
527	Warning	Log a warning message.

### 14.5.3 Property overview

Page	Properties	Access	Description
529	Active	rw	Activate the log mechanism
528	AppendContent	rw	Control whether output is appended to an existing file
530	CustomLogType	rw	Custom log type ID
529	DefaultEventType	rw	Default event type for the Log (526) call.
530	EventIDOffset	rw	Offset for event ID messages identifiers
529	FileName	rw	File name for log file
528	Identification	rw	Identification string for messages
528	LogType	rw	Log type
531	OnGetCustomCategory	rw	Event to retrieve custom message category
531	OnGetCustomEvent	rw	Event to retrieve custom event Code
531	OnGetCustomEventID	rw	Event to retrieve custom event ID
532	Paused	rw	Is the message sending paused ?
529	RaiseExceptionOnError	rw	Determines whether logging errors are reported or ignored
530	TimeStampFormat	rw	Format for the timestamp string

### 14.5.4 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` (529) property to `False`.

See also: `Active` (529)

### 14.5.5 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: `#rtl.sysutils.TEventType` (??)

### 14.5.6 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

**1** Should contain the description of the `etInfo` event type.

**2** Should contain the description of the `etWarning` event type.

**4** Should contain the description of the `etError` event type.

**4** Should 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` (530) (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 viewer in 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.

### 14.5.7 TEventLog.UnRegisterMessageFile

Synopsis: Unregister the message file (needed on windows only)

Declaration: `function UnRegisterMessageFile : Boolean; Virtual`

Visibility: public

Description: `UnRegisterMessageFile` can be used to unregister a message file previously registered with `TEventLog.RegisterMessageFile` (525). This function is usable only on windows, it has no effect on other platforms. Note that windows locks the registered message file while viewing messages, so unregistering helps to avoid file locks while event viewer is open.

See also: `TEventLog.RegisterMessageFile` (525)

### 14.5.8 TEventLog.Pause

Synopsis: Pause the sending of log messages.

Declaration: `procedure Pause`

Visibility: public

Description: `Pause` temporarily suspends the sending of log messages. the various log calls will simply eat the log message and return as if the message was sent.

The sending can be resumed by calling `Resume` (522).

See also: `TEventLog.Resume` (526), `TEventLog.Paused` (532)

### 14.5.9 TEventLog.Resume

Synopsis: Resume sending of log messages if sending was paused

Declaration: `procedure Resume`

Visibility: public

Description: `Resume` resumes the sending of log messages if sending was paused through `Pause` (522).

See also: `TEventLog.Pause` (526), `TEventLog.Paused` (532)

### 14.5.10 TEventLog.Log

Synopsis: Log a message to the system log.

Declaration: `procedure Log(EventType: TEventType; const Msg: string); Overload`  
`procedure Log(EventType: TEventType; const Fmt: string;`  
`Args: Array of const); Overload`  
`procedure Log(const Msg: string); Overload`  
`procedure Log(const 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` (529).

If `EventType` is `etCustom`, then the `OnGetCustomEvent` (531), `OnGetCustomEventID` (531) and `OnGetCustomCategory` (531).

The other logging calls: `Info` (528), `Warning` (527), `Error` (527) and `Debug` (527) use the `Log` call to do the actual work.

See also: `Info` (528), `Warning` (527), `Error` (527), `Debug` (527), `OnGetCustomEvent` (531), `OnGetCustomEventID` (531), `OnGetCustomCategory` (531)

### 14.5.11 TEventLog.Warning

**Synopsis:** Log a warning message.

**Declaration:** `procedure Warning(const Msg: string); Overload`  
`procedure Warning(const 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: `Log` (526), `Info` (528), `Error` (527), `Debug` (527)

### 14.5.12 TEventLog.Error

**Synopsis:** Log an error message to

**Declaration:** `procedure Error(const Msg: string); Overload`  
`procedure Error(const 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: `Log` (526), `Info` (528), `Warning` (527), `Debug` (527)

### 14.5.13 TEventLog.Debug

**Synopsis:** Log a debug message

**Declaration:** `procedure Debug(const Msg: string); Overload`  
`procedure Debug(const 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: `Log` (526), `Info` (528), `Warning` (527), `Error` (527)



### 14.5.14 TEventLog.Info

Synopsis: Log an informational message

Declaration: `procedure Info(const Msg: string); Overload`  
`procedure Info(const 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: [Log \(526\)](#), [Warning \(527\)](#), [Error \(527\)](#), [Debug \(527\)](#)

### 14.5.15 TEventLog.AppendContent

Synopsis: Control whether output is appended to an existing file

Declaration: `Property AppendContent : Boolean`

Visibility: published

Access: Read,Write

Description: `AppendContent` determines what is done when the log type is `ltFile` and a log file already exists. If the log file already exists, then the default behaviour (`AppendContent=False`) is to re-create the log file when the log is activated. If `AppendContent` is `True` then output will be appended to the existing file.

See also: [LogType \(528\)](#), [FileName \(529\)](#)

### 14.5.16 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: [Active \(529\)](#), [TimeStampFormat \(530\)](#)

### 14.5.17 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 \(529\)](#) property.

See also: [FileName \(529\)](#)

### 14.5.18 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: [Log \(526\)](#)

### 14.5.19 TEventLog.RaiseExceptionOnError

Synopsis: Determines whether logging errors are reported or ignored

Declaration: `Property RaiseExceptionOnError : Boolean`

Visibility: published

Access: Read,Write

Description: `RaiseExceptionOnError` determines whether an error during a logging operation will be signaled with an exception or not. If set to `False`, errors will be silently ignored, thus not disturbing normal operation of the program.

### 14.5.20 TEventLog.DefaultEventType

Synopsis: Default event type for the [Log \(526\)](#) call.

Declaration: `Property DefaultEventType : TEventType`

Visibility: published

Access: Read,Write

Description: `DefaultEventType` is the event type used by the [Log \(526\)](#) call if it's `EventType` parameter is omitted.

See also: [Log \(526\)](#)

### 14.5.21 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` (528) 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: `LogType` (528)

### 14.5.22 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` (528)

### 14.5.23 TEventLog.CustomLogType

**Synopsis:** Custom log type ID

**Declaration:** `Property CustomLogType : Word`

**Visibility:** published

**Access:** Read,Write

**Description:** `CustomLogType` is used in the `EventTypeToString` (525) to format the custom log event type string.

See also: `EventTypeToString` (525)

### 14.5.24 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` (525) 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: RegisterMessageFile ([525](#))

### 14.5.25 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: OnGetCustomEventID ([531](#)), OnGetCustomEvent ([531](#))

### 14.5.26 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: OnGetCustomCategory ([531](#)), OnGetCustomEvent ([531](#))

### 14.5.27 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: OnGetCustomCategory ([531](#)), OnGetCustomEventID ([531](#))

### 14.5.28 TEventLog.Paused

Synopsis: Is the message sending paused ?

Declaration: `Property Paused : Boolean`

Visibility: `published`

Access: `Read, Write`

Description: `Paused` indicates whether the sending of messages is temporarily suspended or not. Setting it to `True` has the same effect as calling `Pause` ([522](#)), setting it to `False` has the same effect as calling `Resume` ([522](#)).

See also: `TEventLog.Pause` ([526](#)), `TEventLog.Resume` ([526](#))

# Chapter 15

## Reference for unit 'ezcgi'

### 15.1 Used units

Table 15.1: Used units by unit 'ezcgi'

Name	Page
Classes	??
sysutils	??

### 15.2 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` ([536](#)) or `DoGet` ([536](#)) methods should be overridden.

### 15.3 Constants, types and variables

#### 15.3.1 Constants

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

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

### 15.4 ECGIException

#### 15.4.1 Description

Exception raised by `TEZcgi` ([534](#))

See also: `TEZcgi` ([534](#))

## 15.5 TEZcgi

### 15.5.1 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 (536) or DoGet (536) methods. To run the program, an instance of this class must be created, and it's Run (535) 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.

### 15.5.2 Method overview

Page	Method	Description
534	Create	Creates a new instance of the TEZCGI component
534	Destroy	Removes the TEZCGI component from memory
536	DoGet	Method to handle GET requests
536	DoPost	Method to handle POST requests
536	GetValue	Return the value of a request variable.
535	PutLine	Send a line of output to the web-client
535	Run	Run the CGI application.
535	WriteContent	Writes the content type to standard output

### 15.5.3 Property overview

Page	Properties	Access	Description
538	Email	rw	Email of the server administrator
538	Name	rw	Name of the server administrator
537	Names	r	Indexed array with available variable names.
536	Values	r	Variables passed to the CGI script
538	VariableCount	r	Number of available variables.
537	Variables	r	Indexed array with variables as name=value pairs.

### 15.5.4 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 (537) property.

See also: Variables (537), Names (537), Values (536)

### 15.5.5 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: `Create` ([534](#))

### 15.5.6 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` ([537](#)) property) and calls the `DoPost` ([536](#)) or `DoGet` ([536](#)) 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: `Variables` ([537](#)), `DoPost` ([536](#)), `DoGet` ([536](#))

### 15.5.7 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` ([536](#)) or `DoGet` ([536](#)) methods.

See also: `DoPost` ([536](#)), `DoGet` ([536](#)), `PutLine` ([535](#))

### 15.5.8 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` ([535](#)) 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: `WriteContent` ([535](#))



### 15.5.9 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: [Values \(536\)](#)

### 15.5.10 TEZcgi.DoPost

Synopsis: Method to handle POST requests

Declaration: `procedure DoPost; Virtual`

Visibility: public

Description: `DoPost` is called by the `Run (535)` 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: [Run \(535\)](#), [DoGet \(536\)](#)

### 15.5.11 TEZcgi.DoGet

Synopsis: Method to handle GET requests

Declaration: `procedure DoGet; Virtual`

Visibility: public

Description: `DoGet` is called by the `Run (535)` 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: [Run \(535\)](#), [DoPost \(536\)](#)

### 15.5.12 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 \(537\)](#), [TEZCGI.Names \(537\)](#), [TEZCGI.GetValue \(536\)](#), [TEZcgi.VariableCount \(538\)](#)

### 15.5.13 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` ([538](#)). Any other value will result in an exception being raised.

See also: [TEZcgi.Variables \(537\)](#), [TEZcgi.Values \(536\)](#), [TEZcgi.GetValue \(536\)](#), [TEZcgi.VariableCount \(538\)](#)

### 15.5.14 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` (538). Any other value will result in an exception being raised.

See also: `TEZcgi.Names` (537), `TEZcgi.Values` (536), `TEZcgi.GetValue` (536), `TEZcgi.VariableCount` (538)

### 15.5.15 `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` (537) and `Variables` (537) properties.

See also: `Names` (537), `Variables` (537), `TEZcgi.Values` (536), `TEZcgi.GetValue` (536)

### 15.5.16 `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` (535) method.

See also: `TEZcgi.Run` (535), `TEZcgi.Email` (538)

### 15.5.17 `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` (535) method.

See also: `TEZcgi.Run` (535), `TEZcgi.Name` (538)

## Chapter 16

# Reference for unit 'fpjson'

### 16.1 Used units

Table 16.1: Used units by unit 'fpjson'

Name	Page
Classes	??
Contrns	<a href="#">200</a>
sysutils	??

### 16.2 Overview

The JSON unit implements JSON support for Free Pascal. It contains the data structures (`TJSONData` ([559](#)) and descendent objects) to treat JSON data and output JSON as a string `TJSONData.AsJSON` ([567](#)). The generated JSON can be formatted in several ways `TJSONData.FormatJSON` ([563](#)).

Using the JSON data structures is simple. Instantiate an appropriate descendent of `TJSONData`, set the data and call `AsJSON`. The following JSON data types are supported:

**Numbers** in one of `TJSONIntegerNumber` ([568](#)), `TJSONFloatNumber` ([567](#)) or `TJSONInt64Number` ([539](#)), depending on the type of the number.

**Strings** in `TJSONString` ([580](#)).

**Boolean** in `TJSONBoolean` ([557](#)).

**null** is supported using `TJSONNull` ([569](#))

**Array** is supported using `TJSONArray` ([549](#))

**Object** is supported using `TJSONObject` ([571](#))

The constructors of these objects allow to set the value, making them very easy to use. The memory management is automatic in the sense that arrays and objects own their values, and when the array or object is freed, all data in it is freed as well.

Typical use would be:

```

Var
  O : TJSONObject;

begin
  O:=TJSONObject.Create(['Age', 44,
                        'Firstname', 'Michael',
                        'Lastname', 'Van Canneyt']);

  Writeln(O.AsJSON);
  Write('Welcome ', O.Strings['Firstname'], ', ');
  Writeln(O.Get('Lastname', '')); // empty default.
  Writeln(', your current age is ', O.Integers('Age'));
  O.Free;
end;

```

The `TJSONArray` and `TJSONObject` classes offer methods to examine, get and set the various members and search through the various members.

Currently the JSON support only allows the use of UTF-8 data.

Parsing incoming JSON and constructing the JSON data structures is not implemented in the `fpJSON` unit. For this, the `jsonscanner` (539) unit must be included in the program unit clause. This sets several callback hooks (using `SetJSONParserHandler` (539) and then the `GetJSON` (539) function can then be used to transform a string or stream to JSON data structures:

```

uses fpjson, jsonparser;

Var
  D,E : TJSONData;

begin
  D:=GetJSON('{ "Children" : ['+
    '  { "Age" : 23, '+
    '    "Names" : { "LastName" : "Rodriquez", '+
    '                "FirstName" : "Roberto" } }, '+
    '  { "Age" : 20, '+
    '    "Names" : { "LastName" : "Rodriquez", '+
    '                "FirstName" : "Maria" } } '+
    ' ] }');
  E:=D.FindPath('Children[1].Names.FirstName');
  Writeln(E.AsJSON);
  D.Free;
end.

```

will print "Maria".

The `FPJSON` code does not use hardcoded class names when creating the JSON: it uses the various `CreateJSON` (545) functions to create the data. These functions use a registry of classes, so it is possible to create descendents of the classes in the `fpjson` unit and have these used for construction of JSON Data structures. The `GetJSONInstanceType` (546) and `SetJSONInstanceType` (546) functions can be used to get or set the classes that must be used. the default parser used by `GetJSON` (539) will also use these functions.

## 16.3 Constants, types and variables

### 16.3.1 Constants

`ActualValueJSONTypes = ValueJSONTypes - [jtNull]`

`ActualValueJSONTypes` is a set constant designating the JSON types that have a non-null single value, i.e., all types except array or object or null.

`AsCompactJSON = [foSingleLineArray, foSingleLineObject, foskipWhiteSpace, foDoNotQuoteMembers]`

`AsCompressedJSON` can be used to let `FormatJSON` (563) behave as `TJSONData.AsJSON` (567) with `TJSONData.CompressedJSON` (564) equal to `True` and `TJSONData.UnquotedMemberNames` (559) equal to `True`.

`AsCompressedJSON = [foSingleLineArray, foSingleLineObject, foskipWhiteSpace]`

`AsCompressedJSON` can be used to let `FormatJSON` (563) behave as `TJSONData.AsJSON` (567) with `TJSONData.CompressedJSON` (564) equal to `True`

`AsJSONFormat = [foSingleLineArray, foSingleLineObject]`

`AsJSONFormat` contains the options that make `TJSONData.FormatJSON` (563) behave like `TJSONData.AsJSON` (567)

`DefaultFormat = []`

`DefaultFormat` contains the default formatting options used in formatted JSON.

`DefaultIndentSize = 2`

`DefaultIndentSize` is the default indent size used in formatted JSON.

`StructuredJSONTypes = [jtArray, jtObject]`

`StructuredJSONTypes` is a set constant designating the JSON types that contain multiple values: array or object.

`ValueJSONTypes = [jtNumber, jtString, jtBoolean, jtNull]`

`ValueJSONTypes` is a set constant designating the JSON types that have a single value, i.e., all types except array or object.

### 16.3.2 Types

`TFormatOption = (foSingleLineArray, foSingleLineObject, foDoNotQuoteMembers, foUseTabchar, foSkipWhiteSpace, foSkipWhiteSpaceOnlyLeading)`

Table 16.2: Enumeration values for type TFormatOption

Value	Explanation
foDoNotQuoteMembers	Do not use quote characters around object member names.
foSingleLineArray	Keep all array elements on a single line.
foSingleLineObject	Keep all object elements on a single line.
foSkipWhiteSpace	Skip whitespace
foSkipWhiteSpaceOnlyLeading	Only skip leading whitespace when formatting JSON
foUseTabchar	Use the tabulator character for indents

TFormatOption enumerates the various formatting options that can be used in the TJSONData.FormatJSON (563) function.

TFormatOptions = Set of TFormatOption

TFormatOptions is the set definition used to specify options in TJSONData.FormatJSON (563).

TFPJSSStream = TJSArray

TFPJSSStream resolves to a stream on native platforms, TJSArray in javascript runtimes.

TJSONArrayClass = Class of TJSONArray

TJSONArray is the class type for the TJSONArray (549) class. It is used in CreateJSONArray (545).

TJSONArrayIterator = procedure(Item: TJSONData; Data: TObject;  
var Continue: Boolean) of object

TJSONArrayIterator is the procedural callback used by TJSONArray.Iterate (551) to iterate over the values. Item is the current item in the iteration. Data is the data passed on when calling Iterate. The Continue parameter can be set to false to stop the iteration loop.

TJSONBooleanClass = Class of TJSONBoolean

TJSONBooleanClass is the class type of TJSONBoolean (557). It is used in the factory methods.

TJSONCharType = Char

TJSONCharType is the type of a single character in a TJSONStringType (544) string. It is used by the parser.

TJSONDataClass = Class of TJSONData

TJSONDataClass is used in the CreateJSON (545), SetJSONInstanceType (546) and GetJSONInstanceType (546) functions to set the actual classes used when creating JSON data.

TJSONFloat = Double

TJSONFloat is the floating point type used in the JSON support. It is currently a double, but this can be changed easily.

```
TJSONFloatNumberClass = Class of TJSONFloatNumber
```

TJSONFloatNumberClass is the class type of TJSONFloatNumber (567). It is used in the factory methods.

```
TJSONInstanceType = (jitUnknown, jitNumberInteger, jitNumberFloat,
    jitString, jitBoolean, jitNull, jitArray, jitObject
)
```

Table 16.3: Enumeration values for type TJSONInstanceType

Value	Explanation
jitArray	Array value
jitBoolean	Boolean value
jitNull	Null value
jitNumberFloat	Floating point real number value
jitNumberInteger	32-bit signed integer number value
jitObject	Object value
jitString	String value
jitUnknown	Unknown

TJSONInstanceType is used by the parser to determine what kind of TJJSONData (559) descendent to create for a particular data item. It is a more fine-grained division than TJSONType (544)

```
TJSONIntegerNumberClass = Class of TJSONIntegerNumber
```

TJSONIntegerNumberClass is the class type of TJSONIntegerNumber (568). It is used in the factory methods.

```
TJSONNullClass = Class of TJSONNull
```

TJSONNullClass is the class type of TJSONNull (569). It is used in the factory methods.

```
TJSONNumberType = (ntFloat, ntInteger)
```

Table 16.4: Enumeration values for type TJSONNumberType

Value	Explanation
ntFloat	Floating point value
ntInteger	32-bit Integer value

TJSONNumberType is used to enumerate the different kind of numerical types: JSON only has a single 'number' format. Depending on how the value was parsed, FPC tries to create a value that is as close to the original value as possible: this can be one of integer, int64 or TJSONFloatType (normally a double). The number types have a common ancestor, and they are distinguished by their TJSONNumber.NumberType (571) value.

```
TJSONObjectClass = Class of TJSONObject
```



TJSONObject is the class type for the TJSONObject (571) class. It is used in CreateJSONObject (545).

```
TJSONObjectIterator = procedure(const AName: TJSONStringType;
    Item: TJSONData; Data: TObject;
    var Continue: Boolean) of object
```

TJSONObjectIterator is the procedural callback used by TJSONObject.Iterate (574) to iterate over the values. Item is the current item in the iteration, and AName it's name. Data is the data passed on when calling Iterate. The Continue parameter can be set to false to stop the iteration loop.

```
TJSONStringClass = Class of TJSONString
```

TJSONStringClass is the class type of TJSONString (580). It is used in the factory methods.

```
TJSONStringType = String
```

TJSONFloat is the string point type used in the JSON support. It is currently an ansistring, but this can be changed easily. Unicode characters can be encoded with UTF-8.

```
TJSONtype = (jtUnknown, jtNumber, jtString, jtBoolean, jtNull, jtArray
    ,
    jtObject)
```

Table 16.5: Enumeration values for type TJSONtype

Value	Explanation
jtArray	Array data (integer index, elements can be any type)
jtBoolean	Boolean data
jtNull	Null data
jtNumber	Numerical type. This can be integer (32/64 bit) or float.
jtObject	Object data (named index, elements can be any type)
jtString	String data type.
jtUnknown	Unknown JSON data type

TJSONtype determines the type of JSON data a particular object contains. The class function TJSONData.JSONType (559) returns this type, and indicates what kind of data that particular descendent contains. The values correspond to the original data types in the JSON specification. The TJSONData object itself returns the unknown value.

```
TJSONUnicodeStringType = Unicodestring
```

TJSONUnicodeStringType is an alias used wherever a Unicode (double byte) string is used in the fpJSON code, in particular the TJSONData.AsUnicodeString (559) property.

```
TJSONVariant = jsvalue
```

TJSONVariant resolves to Variant on native platforms, JSValue in javascript runtimes.

## 16.4 Procedures and functions

### 16.4.1 CreateJSON

Synopsis: Create a JSON data item

**Declaration:** `function CreateJSON : TJSONNull`  
`function CreateJSON(Data: Boolean) : TJSONBoolean`  
`function CreateJSON(Data: Integer) : TJSONIntegerNumber`  
`function CreateJSON(Data: TJSONFloat) : TJSONFloatNumber`  
`function CreateJSON(const Data: TJSONStringType) : TJSONString`

Visibility: default

**Description:** `CreateJSON` will create a JSON Data item depending on the type of data passed to it, and will use the classes returned by `GetJSONInstanceType` (546) to do so. The classes to be used can be set using the `SetJSONInstanceType` (546).

The JSON parser uses these functions to create instances of `TJSONData` (559).

Errors: None.

See also: `GetJSONInstanceType` (546), `SetJSONInstanceType` (546), `GetJSON` (539), `CreateJSONArray` (545), `CreateJSONObject` (545)

### 16.4.2 CreateJSONArray

Synopsis: Create a JSON array

**Declaration:** `function CreateJSONArray(const Data: Array of const) : TJSONArray`

Visibility: default

**Description:** `CreateJSONArray` retrieves the class registered to represent JSON array data, and creates an instance of this class, passing `Data` to the constructor. For the `Data` array the same type conversion rules as for the constructor apply.

Errors: if one of the elements in `Data` cannot be converted to a JSON structure, an exception will be raised.

See also: `GetJSONInstanceType` (546), `SetJSONInstanceType` (546), `GetJSON` (539), `CreateJSON` (545), `TJSONArray` (549)

### 16.4.3 CreateJSONObject

Synopsis: Create a JSON object

**Declaration:** `function CreateJSONObject(const Data: Array of const) : TJSONObject`

Visibility: default

**Description:** `CreateJSONObject` retrieves the class registered to represent JSON object data, and creates an instance of this class, passing `Data` to the constructor. For the `Data` array the same type conversion rules as for the `TJSONObject.Create` (572) constructor apply.

Errors: if one of the elements in `Data` cannot be converted to a JSON structure, an exception will be raised.

See also: `GetJSONInstanceType` (546), `SetJSONInstanceType` (546), `GetJSON` (539), `CreateJSON` (545), `TJSONObject` (571)

### 16.4.4 GetJSONInstanceType

Synopsis: JSON factory: Get the JSONData class types to use

Declaration: `function GetJSONInstanceType(AType: TJSONInstanceType) : TJSONDataClass`

Visibility: default

Description: `GetJSONInstanceType` can be used to retrieve the registered descendents of the `TJSONData` (559) class, one for each possible kind of data. The result is the class type used to instantiate data of type `AType`.

The JSON parser and the `CreateJSON` (545) function will use the registered types to instantiate JSON Data. When the parser encounters a value of type `AType`, it will instantiate a class of the type returned by this function. By default, the classes in the `fpJSON` unit are returned.

See also: `CreateJSON` (545), `TJSONData` (559), `GetJSON` (539)

### 16.4.5 JSONStringToString

Synopsis: Convert a JSON-escaped string to a string

Declaration: `function JSONStringToString(const S: TJSONStringType) : TJSONStringType`

Visibility: default

Description: `JSONStringToString` examines the string `S` and replaces any special characters by an escaped string, as in the JSON specification. The following escaped characters are recognized:

```
\\ \" \/ \b \t \n \f \r \u000X
```

See also: `StringToJSONString` (547), `JSONTypeName` (546)

### 16.4.6 JSONTypeName

Synopsis: Convert a JSON type to a string

Declaration: `function JSONTypeName(JSONType: TJSONType) : string`

Visibility: default

Description: `JSONTypeName` converts the `JSONType` to a string that describes the type of JSON value.

See also: `StringToJSONString` (547), `JSONStringToString` (546)

### 16.4.7 SetJSONInstanceType

Synopsis: JSON factory: Set the JSONData class types to use

Declaration: `function SetJSONInstanceType(AType: TJSONInstanceType;  
AClass: TJSONDataClass) : TJSONDataClass`

Visibility: default

**Description:** `SetJSONInstanceType` can be used to register descendents of the `TJSONData` (559) class, one for each possible kind of data. The class type used to instantiate data of type `AType` is passed in `AClass`.

The JSON parser will use the registered types to instantiate JSON Data instanced: when the parser encounters a value of type `AType`, it will instantiate a class of type `AClass`. By default, the classes in the `fpJSON` unit are used.

The `CreateJSON` (545) functions also use the types registered here to instantiate their data.

The return value is the previously registered instance type for the `AType`.

**Errors:** If `AClass` is not suitable to contain data of type `AType`, an exception is raised.

See also: `GetJSONInstanceType` (546), `CreateJSON` (545)

### 16.4.8 StringToJSONString

**Synopsis:** Convert a string to a JSON-escaped string

**Declaration:** `function StringToJSONString(const S: TJJSONStringType; Strict: Boolean) : TJJSONStringType`

**Visibility:** default

**Description:** `StringToJSONString` examines the string `S` and replaces any special characters by an escaped string, as in the JSON specification. The following characters are escaped:

`\ " #8 #9 #10 #12 #13.`

`Strict` indicates that only the absolutely necessary characters will be escaped (when set to `True`) when converging string values to JSON. If set to `False` / will also be escaped, although this is strictly speaking not necessary.

See also: `JSONStringToString` (546), `JSONTypeName` (546)

## 16.5 TJJSONEnum

```
TJJSONEnum = record
  Key : TJJSONStringType;
  KeyNum : Integer;
  Value : TJJSONData;
end
```

`TJJSONEnum` is the loop variable type to use when implementing a JSON enumerator (`for in`). It contains 3 elements which are available in the loop: `key`, `keynum` (numerical key) and the actual value (`TJJSONData`).

## 16.6 EJJSON

### 16.6.1 Description

`EJJSON` is the exception raised by the JSON implementation to report JSON error.

## 16.7 TBaseJSONEnumerator

### 16.7.1 Description

`TBaseJSONEnumerator` is the base type for the JSON enumerators. It should not be used directly, instead use the enumerator support of Object pascal to loop over values in JSON data.

The value of the `TBaseJSONEnumerator` enumerator is a record that describes the key and value of a JSON value. The key can be string-based (for records) or numerical (for arrays).

See also: `TJSONEnum` ([547](#))

### 16.7.2 Method overview

Page	Method	Description
<a href="#">548</a>	<code>GetCurrent</code>	Return the current value of the enumerator
<a href="#">548</a>	<code>MoveNext</code>	Move to next value in array/object

### 16.7.3 Property overview

Page	Properties	Access	Description
<a href="#">548</a>	<code>Current</code>	<code>r</code>	Return the current value of the enumerator

### 16.7.4 TBaseJSONEnumerator.GetCurrent

Synopsis: Return the current value of the enumerator

Declaration: `function GetCurrent : TJSONEnum; Virtual; Abstract`

Visibility: `public`

Description: `GetCurrent` returns the current value of the enumerator. This is a `TJSONEnum` ([547](#)) value.

See also: `TJSONEnum` ([547](#))

### 16.7.5 TBaseJSONEnumerator.MoveNext

Synopsis: Move to next value in array/object

Declaration: `function MoveNext : Boolean; Virtual; Abstract`

Visibility: `public`

Description: `MoveNext` attempts to move to the next value. This will return `True` if the move was successful, or `False` if not. When `True` is returned, then

See also: `TJSONEnum` ([547](#)), `TJSONData` ([559](#))

### 16.7.6 TBaseJSONEnumerator.Current

Synopsis: Return the current value of the enumerator

Declaration: `Property Current : TJSONEnum`

Visibility: `public`

Access: Read

Description: `Current` returns the current enumerator value of type `TJSONEnum` (547). It is only valid after `MoveNext` (539) returned `True`.

See also: `TJSONEnum` (547), `TJSONData` (559)

## 16.8 TJSONArray

### 16.8.1 Description

`TJSONArrayClass` is the class type of `TJSONArray` (549). It is used in the factory methods.

See also: `TJSONArray` (549), `SetJSONInstanceType` (546), `GetJSONInstanceType` (546)

### 16.8.2 Method overview

Page	Method	Description
552	Add	Add a JSON value to the array
551	Clear	Clear the array
550	Clone	Clone the JSON array
549	Create	Create a new instance of JSON array data.
552	Delete	Delete an element from the list by index
550	Destroy	Free the JSON array
552	Exchange	Exchange 2 elements in the list
552	Extract	Extract an element from the array
551	GetEnumerator	Get an array enumerator
551	IndexOf	Return index of <code>JSONData</code> instance in array
553	Insert	Insert an element in the array.
551	Iterate	Iterate over all elements in the array
550	JSONType	native JSON data type
553	Move	Move a value from one location to another
553	Remove	Remove an element from the list
554	Sort	Sort the items in the array

### 16.8.3 Property overview

Page	Properties	Access	Description
556	Arrays	rw	Get or set elements as JSON array values
556	Booleans	rw	Get or set elements as boolean values
556	Floats	rw	Get or set elements as floating-point numerical values
555	Integers	rw	Get or set elements as integer values
554	Items		Indexed access to the values in the array
554	Nulls	r	Check which elements are null
557	Objects	rw	Get or set elements as JSON object values
555	Strings	rw	Get or set elements as string values
554	Types	r	JSON types of elements in the array

### 16.8.4 TJSONArray.Create

Synopsis: Create a new instance of JSON array data.

**Declaration:** `constructor Create; Overload; Reintroduce  
constructor Create(const Elements: Array of const); Overload`

**Visibility:** `public`

**Description:** `Create` creates a new JSON array instance, and initializes the data with `Elements`. The elements are converted to various `TJSONData` (559) instances, instances of `TJSONData` are inserted in the array as-is.

The data type of the inserted objects is determined from the type of data passed to it, with a natural mapping. A `Nil` pointer will be inserted as a `TJSONNull` value.

**Errors:** If an invalid class or not recognized data type (pointer) is inserted in the elements array, an `EConvertError` exception will be raised.

See also: `GetJSONInstanceType` (546)

### 16.8.5 TJSONArray.Destroy

**Synopsis:** Free the JSON array

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

**Visibility:** `public`

**Description:** `Destroy` will delete all elements in the array and clean up the `TJSONArray` (549) instance.

See also: `TJSONArray.Clear` (551), `TJSONArray.Create` (549)

### 16.8.6 TJSONArray.JSONType

**Synopsis:** native JSON data type

**Declaration:** `class function JSONType : TJSONType; Override`

**Visibility:** `public`

**Description:** `JSONType` is overridden by `TJSONArray` to return `jtArray`.

See also: `TJSONData.JSONType` (559)

### 16.8.7 TJSONArray.Clone

**Synopsis:** Clone the JSON array

**Declaration:** `function Clone : TJSONData; Override`

**Visibility:** `public`

**Description:** `Clone` creates a new `TJSONArray`, clones all elements in the array and adds them to the newly created array in the same order as they are in the array.

See also: `TJSONData.Clone` (563)

### 16.8.8 TJSONArray.Iterate

Synopsis: Iterate over all elements in the array

Declaration: `procedure Iterate(Iterator: TJSONArrayIterator; Data: TObject)`

Visibility: public

Description: `Iterate` iterates over all elements in the array, passing them one by one to the `Iterator` callback, together with the `Data` parameter. The iteration stops when all elements have been passed or when the iterator callback returned `False` in the `Continue` parameter.

See also: `TJSONArrayIterator` ([542](#))

### 16.8.9 TJSONArray.IndexOf

Synopsis: Return index of `JSONData` instance in array

Declaration: `function IndexOf(obj: TJSONData) : Integer`

Visibility: public

Description: `IndexOf` compares all elements in the array with `Obj` and returns the index of the element instance that equals `Obj`. The actual instances are compared, not the JSON value. If none of the elements match, the function returns -1.

See also: `Clear` ([539](#))

### 16.8.10 TJSONArray.GetEnumerator

Synopsis: Get an array enumerator

Declaration: `function GetEnumerator : TBaseJSONEnumerator; Override`

Visibility: public

Description: `GetEnumerator` is overridden in `TJSONArray` so it returns an array enumerator. The array enumerator will return all the elements in the array, and stores their index in the `KeyNum` member of `TJSONEnum` ([547](#)).

See also: `TJSONEnum` ([547](#)), `TJSONData.GetEnumerator` ([561](#))

### 16.8.11 TJSONArray.Clear

Synopsis: Clear the array

Declaration: `procedure Clear; Override`

Visibility: public

Description: `Clear` clears the array and frees all elements in it. After the call to clear, `Count` ([564](#)) returns 0.

See also: `Delete` ([539](#)), `Extract` ([539](#))



### 16.8.12 TJSONArray.Add

Synopsis: Add a JSON value to the array

Declaration:

```
function Add(Item: TJSONData) : Integer
function Add(I: Integer) : Integer
function Add(const S: string) : Integer
function Add : Integer
function Add(F: TJSONFloat) : Integer
function Add(B: Boolean) : Integer
function Add(AnArray: TJSONArray) : Integer
function Add(AnObject: TJSONObject) : Integer
```

Visibility: public

Description: Add adds the value passed on to the array. If it is a plain pascal value, it is converted to an appropriate TJSONData (559) instance. If a TJSONData instance is passed, it is simply added to the array. Note that the instance will be owned by the array, and destroyed when the array is cleared (this is in particular true is an JSON array or object).

The function returns the TJSONData instance that was added to the array.

See also: Delete (539), Extract (539)

### 16.8.13 TJSONArray.Delete

Synopsis: Delete an element from the list by index

Declaration:

```
procedure Delete(Index: Integer)
```

Visibility: public

Description: Delete deletes the element with given Index from the list. The TJSONData (559) element is freed.

Errors: If an invalid index is passed, an exception is raised.

See also: Clear (539), Add (539), Extract (539), Exchange (539)

### 16.8.14 TJSONArray.Exchange

Synopsis: Exchange 2 elements in the list

Declaration:

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

Visibility: public

Description: Exchange exchanges 2 elements at locations Index1 and Index2 in the list. This is more efficient than manually extracting and adding the elements to the list.

Errors: If an invalid index (for either element) is passed, an exception is raised.

### 16.8.15 TJSONArray.Extract

Synopsis: Extract an element from the array

Declaration:

```
function Extract(Item: TJSONData) : TJSONData
function Extract(Index: Integer) : TJSONData
```

Visibility: public

Description: `Extract` removes the element at position `Index` or the indicated element from the list, just as `Delete` (539) does. In difference with `Delete`, it does not free the object instance. Instead, it returns the extracted element.

See also: `Delete` (539), `Clear` (539), `Insert` (539), `Add` (539)

### 16.8.16 TJSONArray.Insert

Synopsis: Insert an element in the array.

Declaration: `procedure Insert (Index: Integer)`  
`procedure Insert (Index: Integer; Item: TJSONData)`  
`procedure Insert (Index: Integer; I: Integer)`  
`procedure Insert (Index: Integer; const S: string)`  
`procedure Insert (Index: Integer; F: TJSONFloat)`  
`procedure Insert (Index: Integer; B: Boolean)`  
`procedure Insert (Index: Integer; AnArray: TJSONArray)`  
`procedure Insert (Index: Integer; AnObject: TJSONObject)`

Visibility: public

Description: `Insert` adds a value or element to the array at position `Index`. Elements with index equal to or larger than `Index` are shifted. Like `Add` (539), it converts plain pascal values to JSON values.

Note that when inserting a `TJSONData` (539) instance to the array, it is owned by the array. `Index` must be a value between 0 and `Count-1`.

Errors: If an invalid index is specified, an exception is raised.

See also: `Add` (539), `Delete` (539), `Extract` (539), `Clear` (539)

### 16.8.17 TJSONArray.Move

Synopsis: Move a value from one location to another

Declaration: `procedure Move (CurIndex: Integer; NewIndex: Integer)`

Visibility: public

Description: `Move` moves the element at index `CurIndex` to the position `NewIndex`. It will shift the elements in between as needed. This operation is more efficient than extracting and inserting the element manually.

See also: `Exchange` (539)

### 16.8.18 TJSONArray.Remove

Synopsis: Remove an element from the list

Declaration: `procedure Remove (Item: TJSONData)`

Visibility: public

Description: `Remove` removes `item` from the array, if it is in the array. The object pointer is checked for presence in the array, not the JSON values. Note that the element is freed if it was in the array and is removed.

See also: `Delete` (539), `Extract` (539)

### 16.8.19 TArray.Sort

Synopsis: Sort the items in the array

Declaration: `procedure Sort (Compare: TListSortCompare)`

Visibility: `public`

Description: `Sort` can be used to perform a sort in an array. The array does not compare elements, for this the `Compare` callback must be used, to compare 2 elements from the array.

Errors: None.

See also: `#rtl.classes.TListSortCompare` (??)

### 16.8.20 TArray.Items

Synopsis: Indexed access to the values in the array

Declaration: `Property Items : ; default`

Visibility: `public`

Access:

Description: `Items` is introduced in `TJSONData.Items` (564). `TJSONArray` simply declares it as the default property.

See also: `TJSONData.Items` (564)

### 16.8.21 TArray.Types

Synopsis: JSON types of elements in the array

Declaration: `Property Types[Index: Integer]: TJSONType`

Visibility: `public`

Access: `Read`

Description: `Types` gives direct access to the `TJSONData.JSONType` (559) result of the elements in the array. Accessing it is equivalent to accessing

`Items[Index].JSONType`

See also: `TJSONData.JSONType` (559), `TJSONData.Items` (564)

### 16.8.22 TArray.Nulls

Synopsis: Check which elements are null

Declaration: `Property Nulls[Index: Integer]: Boolean`

Visibility: `public`

Access: `Read`

Description: `Nulls` gives direct access to the `TJSONData.IsNull` (566) property when reading. It is then equivalent to accessing

```
Items[Index].IsNull
```

See also: [TJSONData.JSONType \(559\)](#), [TJSONData.Items \(564\)](#), [TJSONData.IsNull \(566\)](#), [TJSONArray.Types \(554\)](#)

### 16.8.23 TJSONArray.Integers

Synopsis: Get or set elements as integer values

Declaration: `Property Integers[Index: Integer]: Integer`

Visibility: public

Access: Read,Write

Description: `Integers` gives direct access to the [TJSONData.AsInteger \(566\)](#) property when reading. Reading it is the equivalent to accessing

```
Items[Index].AsInteger
```

When writing, it will check if an integer JSON value is located at the given location, and replace it with the new value. If a non-integer JSON value is there, it is replaced with the written integer value.

See also: [TJSONData.Items \(564\)](#), [TJSONData.IsNull \(566\)](#), [TJSONArray.Types \(554\)](#), [TJSONArray.Int64s \(549\)](#), [TJSONArray.QWords \(549\)](#), [TJSONArray.Floats \(556\)](#), [TJSONArray.Strings \(555\)](#), [TJSONArray.Booleans \(556\)](#)

### 16.8.24 TJSONArray.Strings

Synopsis: Get or set elements as string values

Declaration: `Property Strings[Index: Integer]: TJSONStringType`

Visibility: public

Access: Read,Write

Description: `Strings` gives direct access to the [TJSONData.AsString \(565\)](#) property when reading. Reading it is the equivalent to accessing

```
Items[Index].AsString
```

When writing, it will check if a string JSON value is located at the given location, and replace it with the new value. If a non-string value is there, it is replaced with the written string value.

See also: [TJSONData.Items \(564\)](#), [TJSONData.IsNull \(566\)](#), [TJSONArray.Types \(554\)](#), [TJSONArray.Integers \(555\)](#), [TJSONArray.QWords \(549\)](#), [TJSONArray.Floats \(556\)](#), [TJSONArray.Int64s \(549\)](#), [TJSONArray.Booleans \(556\)](#)

### 16.8.25 TJSONArray.Floats

Synopsis: Get or set elements as floating-point numerical values

Declaration: `Property Floats[Index: Integer]: TJSONFloat`

Visibility: public

Access: Read,Write

Description: `Floats` gives direct access to the `TJSONData.AsFloat` (565) property when reading. Reading it is the equivalent to accessing

```
Items[Index].AsFloat
```

When writing, it will check if a floating point numerical JSON value is located at the given location, and replace it with the new value. If a non-floating point numerical value is there, it is replaced with the written floating point value.

See also: `TJSONData.Items` (564), `TJSONData.IsNull` (566), `TJSONArray.Types` (554), `TJSONArray.Integers` (555), `TJSONArray.Strings` (555), `TJSONArray.Int64s` (549), `TJSONArray.QWords` (549), `TJSONArray.Booleans` (556)

### 16.8.26 TJSONArray.Booleans

Synopsis: Get or set elements as boolean values

Declaration: `Property Booleans[Index: Integer]: Boolean`

Visibility: public

Access: Read,Write

Description: `Booleans` gives direct access to the `TJSONData.AsBoolean` (566) property when reading. Reading it is the equivalent to accessing

```
Items[Index].AsBoolean
```

When writing, it will check if a boolean JSON value is located at the given location, and replace it with the new value. If a non-boolean value is there, it is replaced with the written boolean value.

See also: `TJSONData.Items` (564), `TJSONData.IsNull` (566), `TJSONArray.Types` (554), `TJSONArray.Integers` (555), `TJSONArray.Strings` (555), `TJSONArray.Int64s` (549), `TJSONArray.QWords` (549), `TJSONArray.Floats` (556)

### 16.8.27 TJSONArray.Arrays

Synopsis: Get or set elements as JSON array values

Declaration: `Property Arrays[Index: Integer]: TJSONArray`

Visibility: public

Access: Read,Write

Description: `Arrays` gives direct access to JSON Array values when reading. Reading it is the equivalent to accessing

```
Items[Index] As TJSONArray
```

When writing, it will replace any previous value at that location with the written value. Note that the old value is freed, and the new value is owned by the array.

See also: [TJSONData.Items \(564\)](#), [TJSONData.IsNull \(566\)](#), [TJSONArray.Types \(554\)](#), [TJSONArray.Integers \(555\)](#), [TJSONArray.Strings \(555\)](#), [TJSONArray.Int64s \(549\)](#), [TJSONArray.QWords \(549\)](#), [TJSONArray.Floats \(556\)](#), [TJSONArray.Objects \(557\)](#)

### 16.8.28 TJSONArray.Objects

Synopsis: Get or set elements as JSON object values

Declaration: `Property Objects[Index: Integer]: TJSONObject`

Visibility: public

Access: Read,Write

Description: `Objects` gives direct access to JSON object values when reading. Reading it is the equivalent to accessing

```
Items[Index] As TJSONObject
```

When writing, it will replace any previous value at that location with the written value. Note that the old value is freed, and the new value is owned by the array.

See also: [TJSONData.Items \(564\)](#), [TJSONData.IsNull \(566\)](#), [TJSONArray.Types \(554\)](#), [TJSONArray.Integers \(555\)](#), [TJSONArray.Strings \(555\)](#), [TJSONArray.Int64s \(549\)](#), [TJSONArray.QWords \(549\)](#), [TJSONArray.Floats \(556\)](#), [TJSONArray.Arrays \(556\)](#)

## 16.9 TJSONBoolean

### 16.9.1 Description

`TJSONBoolean` must be used whenever boolean data must be represented. It has limited functionality to convert the value from or to integer or floating point data.

See also: [TJSONFloatNumber \(567\)](#), [TJSONIntegerNumber \(568\)](#), [TJSONInt64Number \(539\)](#), [TJSONBoolean \(557\)](#), [TJSONNull \(569\)](#), [TJSONArray \(549\)](#), [TJSONObject \(571\)](#)

### 16.9.2 Method overview

Page	Method	Description
<a href="#">558</a>	<code>Clear</code>	Clear data
<a href="#">558</a>	<code>Clone</code>	Clone boolean value
<a href="#">558</a>	<code>Create</code>	Create a new instance of boolean JSON data
<a href="#">558</a>	<code>JSONType</code>	native JSON data type

### 16.9.3 TJSONBoolean.Create

Synopsis: Create a new instance of boolean JSON data

Declaration: `constructor Create(AValue: Boolean);` Reintroduce

Visibility: public

Description: `Create` instantiates a new boolean JSON data and initializes the value with `AValue`.

See also: `TJSONIntegerNumber.Create` ([568](#)), `TJSONFloatNumber.Create` ([567](#)), `TJSONInt64Number.Create` ([539](#)), `TJSONString.Create` ([580](#)), `TJSONNull.Create` ([569](#)), `TJSONArray.Create` ([549](#)), `TJSONObject.Create` ([572](#))

### 16.9.4 TJSONBoolean.JSONType

Synopsis: native JSON data type

Declaration: `class function JSONType : TJSONType;` Override

Visibility: public

Description: `JSONType` is overridden by `TJSONString` to return `jtBoolean`.

See also: `TJSONData.JSONType` ([559](#))

### 16.9.5 TJSONBoolean.Clear

Synopsis: Clear data

Declaration: `procedure Clear;` Override

Visibility: public

Description: `Clear` is overridden by `TJSONBoolean` to set the value to `False`.

See also: `TJSONData.Clear` ([560](#))

### 16.9.6 TJSONBoolean.Clone

Synopsis: Clone boolean value

Declaration: `function Clone : TJSONData;` Override

Visibility: public

Description: `Clone` overrides `TJSONData.Clone` ([563](#)) and creates an instance of the same class with the same boolean value.

See also: `TJSONData.Clone` ([563](#))

## 16.10 TJSONData

### 16.10.1 Description

`TJSONData` is an abstract class which introduces all properties and methods needed to work with JSON-based data. It should never be instantiated. Based on the type of data that must be represented one of the following descendents must be instantiated instead.

**Numbers** must be represented using one of `TJSONIntegerNumber` (568), `TJSONFloatNumber` (567) or `TJSONInt64Number` (539), depending on the type of the number.

**Strings** can be represented with `TJSONString` (580).

**Boolean** can be represented with `TJSONBoolean` (557).

**null** is supported using `TJSONNull` (569)

**Array** data can be represented using `TJSONArray` (549)

**Object** data can be supported using `TJSONObject` (571)

See also: `TJSONIntegerNumber` (568), `TJSONString` (580), `TJSONBoolean` (557), `TJSONNull` (569), `TJSONArray` (549), `TJSONObject` (571)

### 16.10.2 Method overview

Page	Method	Description
560	<code>Clear</code>	Clear the raw value of this data object
563	<code>Clone</code>	Duplicate the value of the JSON data
560	<code>Create</code>	Create a new instance of <code>TJSONData</code> .
560	<code>DumpJSON</code>	Fast, memory efficient dump of JSON in stream
561	<code>FindPath</code>	Find data by name
563	<code>FormatJSON</code>	Return a formatted JSON representation of the data.
561	<code>GetEnumerator</code>	Return an enumerator for the data
563	<code>GetPath</code>	Get data by name
559	<code>JSONType</code>	The native JSON data type represented by this object

### 16.10.3 Property overview

Page	Properties	Access	Description
566	<code>AsBoolean</code>	rw	Access the raw JSON value as a boolean
565	<code>AsFloat</code>	rw	Access the raw JSON value as a float
566	<code>AsInteger</code>	rw	Access the raw JSON value as an 32-bit integer
567	<code>AsJSON</code>	r	Return a JSON representation of the value
565	<code>AsString</code>	rw	Access the raw JSON value as a string
564	<code>CompressedJSON</code>	rw	Compress JSON - skip whitespace
564	<code>Count</code>	r	Number of sub-items for this data element
566	<code>IsNull</code>	r	Is the data a null value ?
564	<code>Items</code>	rw	Indexed access to sub-items
565	<code>Value</code>	rw	The value of this data object as a variant.

### 16.10.4 TJSONData.JSONType

Synopsis: The native JSON data type represented by this object



Declaration: `class function JSONType : TJSONType; Virtual`

Visibility: `public`

Description: `JSONType` indicates the JSON data type that this object will be written as, or the JSON data type that instantiated this object. In `TJSONData`, this function returns `jtUnknown`. Descendents override this method to return the correct data type.

See also: `TJSONType` ([544](#))

### 16.10.5 TJSONData.Create

Synopsis: Create a new instance of `TJSONData`.

Declaration: `constructor Create; Virtual`

Visibility: `public`

Description: `Create` instantiates a new `TJSONData` object. It should never be called directly, instead one of the descendents should be instantiated.

See also: `TJSONIntegerNumber.Create` ([568](#)), `TJSONString.Create` ([580](#)), `TJSONBoolean.Create` ([558](#)), `TJSONNull.Create` ([569](#)), `TJSONArray.Create` ([549](#)), `TJSONObject.Create` ([572](#))

### 16.10.6 TJSONData.Clear

Synopsis: Clear the raw value of this data object

Declaration: `procedure Clear; Virtual; Abstract`

Visibility: `public`

Description: `Clear` is implemented by the descendents of `TJSONData` to clear the data. An array will be emptied, an object will remove all properties, numbers are set to zero, strings set to the empty string, etc.

See also: `Create` ([539](#))

### 16.10.7 TJSONData.DumpJSON

Synopsis: Fast, memory efficient dump of JSON in stream

Declaration: `procedure DumpJSON(S: TFPJSStream)`

Visibility: `public`

Description: `DumpJSON` writes the data as a JSON string to the stream `S`. No intermediate strings are created, making this a more fast and memory efficient operation than creating a string with `TJSONData.AsJSON` ([567](#)) and writing it to stream.

Errors: None.

See also: `TJSONData.FormatJSON` ([563](#)), `TJSONData.AsJSON` ([567](#))

### 16.10.8 TJSONData.GetEnumerator

Synopsis: Return an enumerator for the data

Declaration: `function GetEnumerator : TBaseJSONEnumerator; Virtual`

Visibility: public

Description: `GetEnumerator` returns an enumerator for the JSON data. For simple types, the enumerator will just contain the current value. For arrays and objects, the enumerator will loop over the values in the array. The return value is not a `TJSONData` (559) type, but a `TJSONEnum` (547) structure, which contains the value, and for structured types, the key (numerical or string).

See also: `TJSONEnum` (547), `TJSONArray` (549), `TJSONObject` (571)

### 16.10.9 TJSONData.FindPath

Synopsis: Find data by name

Declaration: `function FindPath(const APath: TJSONStringType) : TJSONData`

Visibility: public

Description: `FindPath` finds a value based on its path. If none is found, `Nil` is returned. The path elements are separated by dots and square brackets, as in object member notation or array notation. The path is case sensitive.

- For simple values, the path must be empty.
- For objects (571), a member can be specified using its name, and the object value itself can be retrieved with the empty path.
- For Arrays (571), the elements can be found based on an array index. The array value itself can be retrieved with the empty path.

The following code will return the value itself, i.e. `E` will contain the same element as `D`:

```
Var
  D, E : TJSONData;

begin
  D:=TJSONIntegerNumber.Create(123);
  E:=D.FindPath('');
end.
```

The following code will not return anything:

```
Var
  D, E : TJSONData;

begin
  D:=TJSONIntegerNumber.Create(123);
  E:=D.FindPath('a');
end.
```

The following code will return the third element from the array:

```

Var
  D,E : TJSONData;

begin
  D:=TJSONArray.Create([1,2,3,4,5]);
  E:=D.FindPath('[2]');
  Writeln(E.AsJSON);
end.

```

The output of this program is 3.

The following code returns the element Age from the object:

```

Var
  D,E : TJSONData;

begin
  D:=TJSONObject.Create(['Age',23,
                        'Lastame','Rodriguez',
                        'FirstName','Roberto']);
  E:=D.FindPath('Age');
  Writeln(E.AsJSON);
end.

```

The code will print 23.

Obviously, this can be combined:

```

Var
  D,E : TJSONData;

begin
  D:=TJSONObject.Create(['Age',23,
                        'Names', TJSONObject.Create([
                        'LastName','Rodriguez',
                        'FirstName','Roberto'])]);
  E:=D.FindPath('Names.LastName');
  Writeln(E.AsJSON);
end.

```

And mixed:

```

var
  D,E : TJSONData;

begin
  D:=TJSONObject.Create(['Children',
    TJSONArray.Create([
      TJSONObject.Create(['Age',23,
        'Names', TJSONObject.Create([
          'LastName','Rodriguez',
          'FirstName','Roberto'])
        ]),
      TJSONObject.Create(['Age',20,

```

```

        'Names', TJSONObject.Create([
            'LastName', 'Rodriguez',
            'FirstName', 'Maria'])
    ])
    ]);
    E:=D.FindPath('Children[1].Names.FirstName');
    Writeln(E.AsJSON);
end.

```

See also: [TJSONArray \(549\)](#), [TJSONObject \(571\)](#), [GetPath \(539\)](#)

### 16.10.10 TJSONData.GetPath

Synopsis: Get data by name

Declaration: `function GetPath(const APath: TJSONStringType) : TJSONData`

Visibility: public

Description: `GetPath` is identical to `FindPath` ([539](#)) but raises an exception if no element was found. The exception message contains the piece of path that was not found.

Errors: An `EJSON` ([547](#)) exception is raised if the path does not exist.

See also: `FindPath` ([539](#))

### 16.10.11 TJSONData.Clone

Synopsis: Duplicate the value of the JSON data

Declaration: `function Clone : TJSONData; Virtual; Abstract`

Visibility: public

Description: `Clone` returns a new instance of the `TJSONData` descendent that has the same value as the instance, i.e. the `AsJSON` property of the instance and its clone is the same.

Note that the clone must be freed by the caller. Freeing a JSON object will not free its clones.

Errors: Normally, no JSON-specific errors should occur, but an `EOutOfMemory` (??) exception can be raised.

See also: `Clear` ([539](#))

### 16.10.12 TJSONData.FormatJSON

Synopsis: Return a formatted JSON representation of the data.

Declaration: `function FormatJSON(Options: TFormatOptions; Indentsize: Integer) : TJSONStringType`

Visibility: public

**Description:** `FormatJSON` returns a formatted JSON representation of the data. For simple JSON values, this is the same representation as the `AsJSON` (539) property, but for complex values (`TJSONArray` (549) and `TJSONObject` (571)) the JSON is formatted differently.

There are some optional parameters to control the formatting. `Options` controls the use of whitespace and newlines. `IndentSize` controls the amount of indent applied when starting a new line.

The implementation is not optimized for speed.

See also: `AsJSON` (539), `TFormatOptions` (542)

### 16.10.13 TJSONData.CompressedJSON

**Synopsis:** Compress JSON - skip whitespace

**Declaration:** `Property CompressedJSON : Boolean`

**Visibility:** public

**Access:** Read,Write

**Description:** `CompressedJSON` can be used to let `TJSONData.AsJSON` (567) return JSON which does not contain any whitespace. By default it is `False` and whitespace is inserted. If set to `True`, output will contain no whitespace.

See also: `TJSONData.FormatJSON` (563), `TJSONData.AsJSON` (567), `TJSONData.UnquotedMemberNames` (559), `TJSONData.AsCompressedJSON` (559), `TJSONObject.UnquotedMemberNames` (577)

### 16.10.14 TJSONData.Count

**Synopsis:** Number of sub-items for this data element

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

**Visibility:** public

**Access:** Read

**Description:** `Count` is the amount of members of this data element. For simple values (null, boolean, number and string) this is zero. For complex structures, this is the amount of elements in the array or the number of properties of the object

See also: `Items` (539)

### 16.10.15 TJSONData.Items

**Synopsis:** Indexed access to sub-items

**Declaration:** `Property Items[Index: Integer]: TJSONData`

**Visibility:** public

**Access:** Read,Write

**Description:** `Items` allows indexed access to the sub-items of this data. The `Index` is 0-based, and runs from 0 to `Count-1`. For simple data types, this function always returns `Nil`, the complex data type descendents (`TJSONArray` (549) and `TJSONObject` (571)) override this method to return the `Index`-th element in the list.

See also: `Count` (539), `TJSONArray` (549), `TJSONObject` (571)

**16.10.16 TJSONData.Value**

Synopsis: The value of this data object as a variant.

Declaration: `Property Value : TJSONVariant`

Visibility: public

Access: Read,Write

Description: `Value` returns the value of the data object as a variant when read, and converts the variant value to the native JSON type of the object. It does not change the native JSON type (`JSONType` (539)), so the variant value must be convertible to the native JSON type.

For complex types, reading or writing this property will raise an `EConvertError` exception.

See also: `JSONType` (539)

**16.10.17 TJSONData.AsString**

Synopsis: Access the raw JSON value as a string

Declaration: `Property AsString : TJSONStringType`

Visibility: public

Access: Read,Write

Description: `AsString` allows access to the raw value as a string. When reading, it converts the native value of the data to a string. When writing, it attempts to transform the string to a native value. If this conversion fails, an `EConvertError` exception is raised.

For `TJSONString` (580) this will return the native value.

For complex values, reading or writing this property will result in an `EConvertError` exception.

See also: `AsInteger` (539), `Value` (539), `AsInt64` (539), `AsFloat` (539), `AsBoolean` (539), `IsNull` (539), `AsJSON` (539)

**16.10.18 TJSONData.AsFloat**

Synopsis: Access the raw JSON value as a float

Declaration: `Property AsFloat : TJSONFloat`

Visibility: public

Access: Read,Write

Description: `AsFloat` allows access to the raw value as a floating-point value. When reading, it converts the native value of the data to a floating-point. When writing, it attempts to transform the floating-point value to a native value. If this conversion fails, an `EConvertError` exception is raised.

For `TJSONFloatNumber` (567) this will return the native value.

For complex values, reading or writing this property will always result in an `EConvertError` exception.

See also: `AsInteger` (539), `Value` (539), `AsInt64` (539), `AsString` (539), `AsBoolean` (539), `IsNull` (539), `AsJSON` (539)

**16.10.19 TJSONData.AsInteger**

Synopsis: Access the raw JSON value as an 32-bit integer

Declaration: `Property AsInteger : Integer`

Visibility: public

Access: Read,Write

Description: `AsInteger` allows access to the raw value as a 32-bit integer value. When reading, it attempts to convert the native value of the data to a 32-bit integer value. When writing, it attempts to transform the 32-bit integer value to a native value. If either conversion fails, an `EConvertError` exception is raised.

For `TJSONIntegerNumber` (568) this will return the native value.

For complex values, reading or writing this property will always result in an `EConvertError` exception.

See also: `AsFloat` (539), `Value` (539), `AsInt64` (539), `AsString` (539), `AsBoolean` (539), `IsNull` (539), `AsJSON` (539)

**16.10.20 TJSONData.AsBoolean**

Synopsis: Access the raw JSON value as a boolean

Declaration: `Property AsBoolean : Boolean`

Visibility: public

Access: Read,Write

Description: `AsBoolean` allows access to the raw value as a boolean value. When reading, it attempts to convert the native value of the data to a boolean value. When writing, it attempts to transform the boolean value to a native value. For numbers this means that nonzero numbers result in `True`, a zero results in `False`. If either conversion fails, an `EConvertError` exception is raised.

For `TJSONBoolean` (557) this will return the native value.

For complex values, reading or writing this property will always result in an `EConvertError` exception.

See also: `AsFloat` (539), `Value` (539), `AsInt64` (539), `AsString` (539), `AsInteger` (539), `IsNull` (539), `AsJSON` (539)

**16.10.21 TJSONData.IsNull**

Synopsis: Is the data a null value ?

Declaration: `Property IsNull : Boolean`

Visibility: public

Access: Read

Description: `IsNull` is `True` only for `JSONType=jtNull`, i.e. for a `TJSONNull` (569) instance. In all other cases, it is `False`. This value cannot be set.

See also: `TJSONType` (544), `JSONType` (539), `TJSONNull` (569), `AsFloat` (539), `Value` (539), `AsInt64` (539), `AsString` (539), `AsInteger` (539), `AsBoolean` (539), `AsJSON` (539)

### 16.10.22 TJSONData.AsJSON

Synopsis: Return a JSON representation of the value

Declaration: `Property AsJSON : TJSONStringType`

Visibility: public

Access: Read

Description: `AsJSON` returns a JSON representation of the value of the data. For simple values, this is just a textual representation of the object. For objects and arrays, this is an actual JSON Object or array.

See also: `AsFloat` (539), `Value` (539), `AsInt64` (539), `AsString` (539), `AsInteger` (539), `AsBoolean` (539), `AsJSON` (539)

## 16.11 TJSONFloatNumber

### 16.11.1 Description

`TJSONFloatNumber` must be used whenever floating point data must be represented. It can handle `TJSONFloatType` (539) data (normally a double). For integer data, `TJSONIntegerNumber` (568) or `TJSONInt64Number` (539) are better suited.

See also: `TJSONIntegerNumber` (568), `TJSONInt64Number` (539)

### 16.11.2 Method overview

Page	Method	Description
568	<code>Clear</code>	Clear value
568	<code>Clone</code>	Clone floating point value
567	<code>Create</code>	Create a new floating-point value
567	<code>NumberType</code>	Kind of numerical data managed by this class.

### 16.11.3 TJSONFloatNumber.Create

Synopsis: Create a new floating-point value

Declaration: `constructor Create(AValue: TJSONFloat);` Reintroduce

Visibility: public

Description: `Create` instantiates a new JSON floating point value, and initializes it with `AValue`.

See also: `TJSONIntegerNumber.Create` (568), `TJSONInt64Number.Create` (539)

### 16.11.4 TJSONFloatNumber.NumberType

Synopsis: Kind of numerical data managed by this class.

Declaration: `class function NumberType : TJSONNumberType;` Override

Visibility: public

Description: `NumberType` is overridden by `TJSONFloatNumber` to return `ntFloat`.

See also: `TJSONNumberType` (543), `TJSONData.JSONtype` (559)



### 16.11.5 TJSONFloatNumber.Clear

Synopsis: Clear value

Declaration: `procedure Clear; Override`

Visibility: `public`

Description: `Clear` is overridden by `TJSONFloatNumber` to set the value to 0.0

See also: `TJSONData.Clear` ([560](#))

### 16.11.6 TJSONFloatNumber.Clone

Synopsis: Clone floating point value

Declaration: `function Clone : TJSONData; Override`

Visibility: `public`

Description: `Clone` overrides `TJSONData.Clone` ([563](#)) and creates an instance of the same class with the same floating-point value.

See also: `TJSONData.Clone` ([563](#))

## 16.12 TJSONIntegerNumber

### 16.12.1 Description

`TJSONIntegerNumber` must be used whenever 32-bit integer data must be represented. For 64-bit integer data, `TJSONInt64Number` ([539](#)) must be used.

See also: `TJSONFloatNumber` ([567](#)), `TJSONInt64Number` ([539](#))

### 16.12.2 Method overview

Page	Method	Description
<a href="#">569</a>	<code>Clear</code>	Clear value
<a href="#">569</a>	<code>Clone</code>	Clone 32-bit integer value
<a href="#">568</a>	<code>Create</code>	Create a new instance of 32-bit integer JSON data
<a href="#">569</a>	<code>NumberType</code>	Kind of numerical data managed by this class.

### 16.12.3 TJSONIntegerNumber.Create

Synopsis: Create a new instance of 32-bit integer JSON data

Declaration: `constructor Create(AValue: Integer); Reintroduce`

Visibility: `public`

Description: `Create` instantiates a new 32-bit integer JSON data and initializes the value with `AValue`.

See also: `TJSONFloatNumber.Create` ([567](#)), `TJSONInt64Number.Create` ([539](#))

### 16.12.4 TJSONIntegerNumber.NumberType

Synopsis: Kind of numerical data managed by this class.

Declaration: `class function NumberType : TJSONNumberType; Override`

Visibility: public

Description: `NumberType` is overridden by `TJSONIntegerNumber` to return `ntInteger`.

See also: `TJSONNumberType` (543), `TJSONData.JSONType` (559)

### 16.12.5 TJSONIntegerNumber.Clear

Synopsis: Clear value

Declaration: `procedure Clear; Override`

Visibility: public

Description: `Clear` is overridden by `TJSONIntegerNumber` to set the value to 0.

See also: `TJSONData.Clear` (560)

### 16.12.6 TJSONIntegerNumber.Clone

Synopsis: Clone 32-bit integer value

Declaration: `function Clone : TJSONData; Override`

Visibility: public

Description: `Clone` overrides `TJSONData.Clone` (563) and creates an instance of the same class with the same 32-bit integer value.

See also: `TJSONData.Clone` (563)

## 16.13 TJSONNull

### 16.13.1 Description

`TJSONNull` must be used whenever a `null` value must be represented.

See also: `TJSONFloatNumber` (567), `TJSONIntegerNumber` (568), `TJSONInt64Number` (539), `TJSONBoolean` (557), `TJSONString` (580), `TJSONArray` (549), `TJSONObject` (571)

### 16.13.2 Method overview

Page	Method	Description
570	<code>Clear</code>	Clear data
570	<code>Clone</code>	Clone boolean value
570	<code>JSONType</code>	native JSON data type

### 16.13.3 TJSONNull.JSONType

Synopsis: native JSON data type

Declaration: `class function JSONType : TJSONType; Override`

Visibility: public

Description: `JSONType` is overridden by `TJSONNull` to return `jtNull`.

See also: `TJSONData.JSONType` ([559](#))

### 16.13.4 TJSONNull.Clear

Synopsis: Clear data

Declaration: `procedure Clear; Override`

Visibility: public

Description: `Clear` does nothing.

See also: `TJSONData.Clear` ([560](#))

### 16.13.5 TJSONNull.Clone

Synopsis: Clone boolean value

Declaration: `function Clone : TJSONData; Override`

Visibility: public

Description: `Clone` overrides `TJSONData.Clone` ([563](#)) and creates an instance of the same class.

See also: `TJSONData.Clone` ([563](#))

## 16.14 TJSONNumber

### 16.14.1 Description

`TJSONNumber` is an abstract class which serves as the ancestor for the 3 numerical classes. It should never be instantiated directly. Instead, depending on the kind of data, one of `TJSONIntegerNumber` ([568](#)), `TJSONInt64Number` ([539](#)) or `TJSONFloatNumber` ([567](#)) should be instantiated.

See also: `TJSONIntegerNumber` ([568](#)), `TJSONInt64Number` ([539](#)), `TJSONFloatNumber` ([567](#))

### 16.14.2 Method overview

Page	Method	Description
<a href="#">571</a>	<code>JSONType</code>	native JSON data type
<a href="#">571</a>	<code>NumberType</code>	Kind of numerical data managed by this class.

### 16.14.3 TJSONNumber.JSONType

Synopsis: native JSON data type

Declaration: `class function JSONType : TJSONType; Override`

Visibility: public

Description: `JSONType` is overridden by `TJSONNumber` to return `jtNumber`.

See also: `TJSONData.JSONType` (559)

### 16.14.4 TJSONNumber.NumberType

Synopsis: Kind of numerical data managed by this class.

Declaration: `class function NumberType : TJSONNumberType; Virtual; Abstract`

Visibility: public

Description: `NumberType` is overridden by `TJSONNumber` descendents to return the kind of numerical data that can be managed by the class.

See also: `TJSONIntegerNumber` (568), `TJSONInt64Number` (539), `TJSONFloatNumber` (567), `JSONType` (539)

## 16.15 TJSONObject

### 16.15.1 Description

`TJSONObjectClass` is the class type of `TJSONObject` (571). It is used in the factory methods.

See also: `TJSONObject` (571), `SetJSONInstanceType` (546), `GetJSONInstanceType` (546)

### 16.15.2 Method overview

Page	Method	Description
<a href="#">576</a>	Add	Add a name, value to the object
<a href="#">575</a>	Clear	Clear the object
<a href="#">573</a>	Clone	Clone the JSON object
<a href="#">572</a>	Create	Create a new instance of JSON object data.
<a href="#">576</a>	Delete	Delete an element from the list by index
<a href="#">573</a>	Destroy	Free the JSON object
<a href="#">574</a>	Find	Find an element by name.
<a href="#">575</a>	Get	Retrieve a value by name
<a href="#">573</a>	GetEnumerator	Get an object enumerator
<a href="#">574</a>	IndexOf	Return index of <code>JSONData</code> instance in object
<a href="#">574</a>	IndexOfName	Return index of name in item list
<a href="#">574</a>	Iterate	Iterate over all elements in the object
<a href="#">573</a>	<code>JSONType</code>	native JSON data type
<a href="#">576</a>	Remove	Remove item by instance

### 16.15.3 Property overview

Page	Properties	Access	Description
<a href="#">579</a>	Arrays	rw	Named access to JSON array values
<a href="#">579</a>	Booleans	rw	Named access to boolean values
<a href="#">577</a>	Elements	rw	Name-based access to JSON values in the object.
<a href="#">578</a>	Floats	rw	Named access to float values
<a href="#">578</a>	Integers	rw	Named access to integer values
<a href="#">577</a>	Names	r	Indexed access to the names of elements.
<a href="#">578</a>	Nulls	rw	Named access to <code>null</code> values
<a href="#">580</a>	Objects	rw	Named access to JSON object values
<a href="#">579</a>	Strings	rw	Named access to string values
<a href="#">578</a>	Types	r	Types of values in the object.
<a href="#">577</a>	UnquotedMemberNames	rw	Should member names be written unquoted or quoted in JSON

### 16.15.4 TJSONObject.Create

**Synopsis:** Create a new instance of JSON object data.

**Declaration:** `constructor Create; Reintroduce`  
`constructor Create(const Elements: Array of const); Overload`

**Visibility:** `public`

**Description:** `Create` creates a new JSON object instance, and initializes the data with `Elements`. `Elements` is an array containing an even number of items, alternating a name and a value. The names must be strings, and the values are converted to various `TJSONData` ([559](#)) instances. If a value is an instance of `TJSONData`, it is added to the object array as-is.

The data type of the inserted objects is determined from the type of data passed to it, with a natural mapping. A `Nil` pointer will be inserted as a `TJSONNull` value. The following gives an example:

```
Var
  O : TJSONObject;

begin
  O:=TJSONObject.Create(['Age', 44,
                        'Firstname', 'Michael',
                        'Lastname', 'Van Canneyt']);
```

**Errors:** An `EConvertError` exception is raised in one of the following cases:

- 1.If an odd number of arguments is passed
- 2.an item where a name is expected does not contain a string
- 3.A value contains an invalid class
- 4.A value of a not recognized data type (pointer) is inserted in the elements

See also: `Add` ([539](#)), `GetJSONInstanceType` ([546](#))

### 16.15.5 TJSONObject.Destroy

Synopsis: Free the JSON object

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` will delete all elements in the array and clean up the `TJSONObject` (571) instance.

See also: `TJSONObject.Clear` (575), `TJSONObject.Create` (572)

### 16.15.6 TJSONObject.JSONType

Synopsis: native JSON data type

Declaration: `class function JSONType : TJSONType; Override`

Visibility: `public`

Description: `JSONType` is overridden by `TJSONObject` to return `jtObject`.

See also: `TJSONData.JSONType` (559)

### 16.15.7 TJSONObject.Clone

Synopsis: Clone the JSON object

Declaration: `function Clone : TJSONData; Override`

Visibility: `public`

Description: `Clone` creates a new `TJSONObject`, clones all elements in the array and adds them to the newly created array with the same names as they were in the array.

See also: `TJSONData.Clone` (563)

### 16.15.8 TJSONObject.GetEnumerator

Synopsis: Get an object enumerator

Declaration: `function GetEnumerator : TBaseJSONEnumerator; Override`

Visibility: `public`

Description: `GetEnumerator` is overridden in `TJSONObject` so it returns an object enumerator. The array enumerator will return all the elements in the array, and stores their name in the `Key` and index in the `KeyNum` members of `TJSONEnum` (547).

See also: `TJSONEnum` (547), `TJSONData.GetEnumerator` (561)

### 16.15.9 TJSONObject.Iterate

Synopsis: Iterate over all elements in the object

Declaration: `procedure Iterate(Iterator: TJSONObjectIterator; Data: TObject)`

Visibility: public

Description: `Iterate` iterates over all elements in the object, passing them one by one with name and value to the `Iterator` callback, together with the `Data` parameter. The iteration stops when all elements have been passed or when the iterator callback returned `False` in the `Continue` parameter.

See also: `TJSONObjectIterator` ([544](#))

### 16.15.10 TJSONObject.IndexOf

Synopsis: Return index of JSONData instance in object

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

Visibility: public

Description: `IndexOf` compares all elements in the object with `Obj` and returns the index (in the `TJSONData.Items` ([564](#)) property) of the element instance that equals `Obj`. The actual instances are compared, not the JSON value. If none of the elements match, the function returns -1.

See also: `Clear` ([539](#)), `IndexOfName` ([539](#))

### 16.15.11 TJSONObject.IndexOfName

Synopsis: Return index of name in item list

Declaration: `function IndexOfName(const AName: TJSONStringType;  
CaseInsensitive: Boolean) : Integer`

Visibility: public

Description: `IndexOfName` compares the names of all elements in the object with `AName` and returns the index (in the `TJSONData.Items` ([564](#)) property) of the element instance whose name matched `AName`. If none of the element's names match, the function returns -1.

Since JSON is a case-sensitive specification, the names are searched case-sensitively by default. This can be changed by setting the optional `CaseInsensitive` parameter to `True`

See also: `IndexOf` ([539](#))

### 16.15.12 TJSONObject.Find

Synopsis: Find an element by name.

Declaration: `function Find(const AName: string) : TJSONData; Overload  
function Find(const AName: string; AType: TJSONtype) : TJSONData  
; Overload  
function Find(const key: TJSONStringType; out AValue: TJSONData)  
: Boolean  
function Find(const key: TJSONStringType; out AValue: TJSONObject)  
: Boolean  
function Find(const key: TJSONStringType; out AValue: TJSONArray)`

```

        : Boolean
function Find(const key: TJSONStringType; out AValue: TJSONString)
        : Boolean
function Find(const key: TJSONStringType; out AValue: TJSONBoolean)
        : Boolean
function Find(const key: TJSONStringType; out AValue: TJSONNumber)
        : Boolean

```

Visibility: public

**Description:** Find compares the names of all elements in the object with AName and returns the matching element. If none of the element's names match, the function returns Nil

Since JSON is a case-sensitive specification, the names are searched case-sensitively.

If AType is specified then the element's type must also match the specified type.

See also: IndexOf ([539](#)), IndexOfName ([539](#))

### 16.15.13 TJSONObject.Get

Synopsis: Retrieve a value by name

**Declaration:**

```

function Get(const AName: string) : TJSONVariant
function Get(const AName: string; ADefault: TJSONFloat) : TJSONFloat
function Get(const AName: string; ADefault: Integer) : Integer
function Get(const AName: string; ADefault: Boolean) : Boolean
function Get(const AName: string; ADefault: TJSONStringType)
        : TJSONStringType
function Get(const AName: string; ADefault: TJSONArray) : TJSONArray
function Get(const AName: string; ADefault: TJSONObject) : TJSONObject

```

Visibility: public

**Description:** Get can be used to retrieve a value by name. If an element with name equal to AName exists, and its type corresponds to the type of the ADefault, then the value is returned. If no element with the correct type exists, the ADefault value is returned.

If no default is specified, the value is returned as a variant type, or Null if no value was found.

The other value retrieval properties such as Integers ([539](#))Int64s ([539](#)), Booleans ([539](#)), Strings ([539](#)), Floats ([539](#)), Arrays ([539](#)), Objects ([539](#)) will raise an exception if the name is not found. The Get function does not raise an exception.

See also: Integers ([539](#)), Int64s ([539](#)), Booleans ([539](#)), Strings ([539](#)), Floats ([539](#)), Arrays ([539](#)), Objects ([539](#))

### 16.15.14 TJSONObject.Clear

Synopsis: Clear the object

**Declaration:** procedure Clear; Override

Visibility: public

**Description:** Clear clears the object and frees all elements in it. After the call to Clear, Count ([564](#)) returns 0.

See also: Delete ([539](#)), Extract ([539](#))



**16.15.15 TJSONObject.Add**

Synopsis: Add a name, value to the object

Declaration: 

```
function Add(const AName: TJSONStringType; AValue: TJSONData) : Integer
    ; Overload
function Add(const AName: TJSONStringType; AValue: Boolean) : Integer
    ; Overload
function Add(const AName: TJSONStringType; AValue: TJSONFloat) : Integer
    ; Overload
function Add(const AName: TJSONStringType;
    const AValue: TJSONStringType) : Integer; Overload
function Add(const AName: TJSONStringType; AValue: Integer) : Integer
    ; Overload
function Add(const AName: TJSONStringType) : Integer; Overload
function Add(const AName: TJSONStringType; AValue: TJSONArray) : Integer
    ; Overload
```

Visibility: public

Description: Add adds the value AValue with name AName to the object. If the value is not a TJSONData (539) descendent, then it is converted to a TJSONData value, and it returns the TJSONData descendent that was created to add the value.

The properties Integers (539), Int64s (539), Booleans (539)Strings (539), Floats (539), Arrays (539) and Objects (539) will not raise an exception if an existing name is used, they will overwrite any existing value.

Errors: If a value with the same name already exists, an exception is raised.

See also: Integers (539), Int64s (539), Booleans (539), Strings (539), Floats (539), Arrays (539), Objects (539)

**16.15.16 TJSONObject.Delete**

Synopsis: Delete an element from the list by index

Declaration: 

```
procedure Delete(Index: Integer)
procedure Delete(const AName: string)
```

Visibility: public

Description: Delete deletes the element with given Index or AName from the list. The TJSONData (539) element is freed. If a non-existing name is specified, no value is deleted.

Errors: If an invalid index is passed, an exception is raised.

See also: Clear (539), Add (539), Extract (539), Exchange (539)

**16.15.17 TJSONObject.Remove**

Synopsis: Remove item by instance

Declaration: 

```
procedure Remove(Item: TJSONData)
```

Visibility: public

Description: Remove will locate the value Item in the list of values, and removes it if it exists. The item is freed.

See also: Delete (539), Extract (539)

**16.15.18 TJSONObject.UnquotedMemberNames**

Synopsis: Should member names be written unquoted or quoted in JSON

Declaration: Property `UnquotedMemberNames` : `Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `UnquotedMemberNames` can be set to let `AsJSON` write the member names of a JSON object without quotes (`True`) or with quotes (`False`) around the member names.

When the value is `False`, JSON is written as:

```
{
  "name" : "Free Pascal",
  "type" : "Compiler"
}
```

When the value is `True`, JSON is written as:

```
{
  name : "Free Pascal",
  type : "Compiler"
}
```

Care must be taken when setting this property: The JSON standard requires the quotes to be written, but since JSON is mostly consumed in a Javascript engine, the unquoted values are usually also accepted.

See also: `TJSONData.CompressedJSON` ([564](#))

**16.15.19 TJSONObject.Names**

Synopsis: Indexed access to the names of elements.

Declaration: Property `Names[Index: Integer]` : `TJSONStringType`

Visibility: `public`

Access: `Read`

Description: `Names` allows to retrieve the names of the elements in the object. The index is zero-based, running from 0 to `Count-1`.

See also: `Types` ([539](#)), `Elements` ([539](#))

**16.15.20 TJSONObject.Elements**

Synopsis: Name-based access to JSON values in the object.

Declaration: Property `Elements[AName: string]` : `TJSONData`; default

Visibility: `public`

Access: `Read,Write`

Description: `Elements` allows to retrieve the JSON values of the elements in the object by name. If a non-existent name is specified, an `EJSON` ([547](#)) exception is raised.

See also: `TJSONData.Items` ([564](#)), `Names` ([539](#)), `Types` ([539](#)), `Integers` ([539](#)), `Int64s` ([539](#)), `Booleans` ([539](#)), `Strings` ([539](#)), `Floats` ([539](#)), `Arrays` ([539](#)), `Objects` ([539](#))

### 16.15.21 TJSONObject.Types

Synopsis: Types of values in the object.

Declaration: `Property Types[AName: string]: TJSONType`

Visibility: public

Access: Read

Description: `Types` allows to retrieve the JSON types of the elements in the object by name. If a non-existent name is specified, an `EJSON` (547) exception is raised.

See also: `TJSONData.Items` (564), `Names` (539), `Elements` (539), `Integers` (539), `Int64s` (539), `Booleans` (539), `Strings` (539), `Floats` (539), `Arrays` (539), `Nulls` (539), `Objects` (539)

### 16.15.22 TJSONObject.Nulls

Synopsis: Named access to null values

Declaration: `Property Nulls[AName: string]: Boolean`

Visibility: public

Access: Read,Write

Description: `Nulls` allows to retrieve or set the NULL values in the object by name. If a non-existent name is specified, an `EJSON` (547) exception is raised when reading. When writing, any existing value is replaced by a null value.

See also: `TJSONData.Items` (564), `Names` (539), `Elements` (539), `Integers` (539), `Int64s` (539), `Booleans` (539), `Strings` (539), `Floats` (539), `Arrays` (539), `Types` (539), `Objects` (539)

### 16.15.23 TJSONObject.Floats

Synopsis: Named access to float values

Declaration: `Property Floats[AName: string]: TJSONFloat`

Visibility: public

Access: Read,Write

Description: `Floats` allows to retrieve or set the float values in the object by name. If a non-existent name is specified, an `EJSON` (547) exception is raised when reading. When writing, any existing value is replaced by the specified floating-point value.

See also: `TJSONData.Items` (564), `Names` (539), `Elements` (539), `Integers` (539), `Int64s` (539), `Booleans` (539), `Strings` (539), `Nulls` (539), `Arrays` (539), `Types` (539), `Objects` (539)

### 16.15.24 TJSONObject.Integers

Synopsis: Named access to integer values

Declaration: `Property Integers[AName: string]: Integer`

Visibility: public

Access: Read,Write

**Description:** `Integers` allows to retrieve or set the integer values in the object by name. If a non-existent name is specified, an `EJSON` (547) exception is raised when reading. When writing, any existing value is replaced by the specified integer value.

See also: `TJSONData.Items` (564), `Names` (539), `Elements` (539), `Floats` (539), `Int64s` (539), `Booleans` (539), `Strings` (539), `Nulls` (539), `Arrays` (539), `Types` (539), `Objects` (539)

### 16.15.25 TJSONObject.Strings

**Synopsis:** Named access to string values

**Declaration:** `Property Strings[AName: string]: TJJSONStringType`

**Visibility:** public

**Access:** Read,Write

**Description:** `Strings` allows to retrieve or set the string values in the object by name. If a non-existent name is specified, an `EJSON` (547) exception is raised when reading. When writing, any existing value is replaced by the specified string value.

See also: `TJSONData.Items` (564), `Names` (539), `Elements` (539), `Floats` (539), `Integers` (539), `Booleans` (539), `Int64s` (539), `Nulls` (539), `Arrays` (539), `Types` (539), `Objects` (539), `UnicodeStrings` (539)

### 16.15.26 TJSONObject.Booleans

**Synopsis:** Named access to boolean values

**Declaration:** `Property Booleans[AName: string]: Boolean`

**Visibility:** public

**Access:** Read,Write

**Description:** `Booleans` allows to retrieve or set the boolean values in the object by name. If a non-existent name is specified, an `EJSON` (547) exception is raised when reading. When writing, any existing value is replaced by the specified boolean value.

See also: `TJSONData.Items` (564), `Names` (539), `Elements` (539), `Floats` (539), `Integers` (539), `Strings` (539), `Int64s` (539), `Nulls` (539), `Arrays` (539), `Types` (539), `Objects` (539)

### 16.15.27 TJSONObject.Arrays

**Synopsis:** Named access to JSON array values

**Declaration:** `Property Arrays[AName: string]: TJJSONArray`

**Visibility:** public

**Access:** Read,Write

**Description:** `Arrays` allows to retrieve or set the JSON array values in the object by name. If a non-existent name is specified, an `EJSON` (547) exception is raised when reading. When writing, any existing value is replaced by the specified JSON array.

See also: `TJSONData.Items` (564), `Names` (539), `Elements` (539), `Floats` (539), `Integers` (539), `Strings` (539), `Int64s` (539), `Nulls` (539), `Booleans` (539), `Types` (539), `Objects` (539)

### 16.15.28 TJSONObject.Objects

Synopsis: Named access to JSON object values

Declaration: `Property Objects[AName: string]: TJSONObject`

Visibility: public

Access: Read, Write

Description: `Objects` allows to retrieve or set the JSON object values in the object by name. If a non-existent name is specified, an `EJSON` (547) exception is raised when reading. When writing, any existing value is replaced by the specified JSON object.

See also: `TJSONData.Items` (564), `Names` (539), `Elements` (539), `Floats` (539), `Integers` (539), `Strings` (539), `Int64s` (539), `Nulls` (539), `Booleans` (539), `Types` (539), `Arrays` (539)

## 16.16 TJSONString

### 16.16.1 Description

`TJSONString` must be used whenever string data must be represented. Currently the implementation uses an ANSI string to hold the data. This means that to correctly hold Unicode data, a UTF-8 encoding must be used.

See also: `TJSONFloatNumber` (567), `TJSONIntegerNumber` (568), `TJSONInt64Number` (539), `TJSONBoolean` (557), `TJSONNull` (569), `TJSONArray` (549), `TJSONObject` (571)

### 16.16.2 Method overview

Page	Method	Description
<a href="#">581</a>	<code>Clear</code>	Clear value
<a href="#">581</a>	<code>Clone</code>	Clone string value
<a href="#">580</a>	<code>Create</code>	Create a new instance of string JSON data
<a href="#">581</a>	<code>JSONType</code>	native JSON data type

### 16.16.3 TJSONString.Create

Synopsis: Create a new instance of string JSON data

Declaration: `constructor Create(const AValue: TJSONStringType);` Reintroduce

Visibility: public

Description: `Create` instantiates a new string JSON data and initializes the value with `AValue`. Currently the implementation uses an ANSI string to hold the data. This means that to correctly hold Unicode data, a UTF-8 encoding must be used.

See also: `TJSONIntegerNumber.Create` (568), `TJSONFloatNumber.Create` (567), `TJSONInt64Number.Create` (539), `TJSONBoolean.Create` (558), `TJSONNull.Create` (569), `TJSONArray.Create` (549), `TJSONObject.Create` (572)

### 16.16.4 TJSONString.JSONType

Synopsis: native JSON data type

Declaration: `class function JSONType : TJSONType; Override`

Visibility: public

Description: `JSONType` is overridden by `TJSONString` to return `jtString`.

See also: `TJSONData.JSONType` ([559](#))

### 16.16.5 TJSONString.Clear

Synopsis: Clear value

Declaration: `procedure Clear; Override`

Visibility: public

Description: `Clear` is overridden by `TJSONString` to set the value to the empty string "".

See also: `TJSONData.Clear` ([560](#))

### 16.16.6 TJSONString.Clone

Synopsis: Clone string value

Declaration: `function Clone : TJSONData; Override`

Visibility: public

Description: `Clone` overrides `TJSONData.Clone` ([563](#)) and creates an instance of the same class with the same string value.

See also: `TJSONData.Clone` ([563](#))

## Chapter 17

# Reference for unit 'fpmimeTypes'

### 17.1 Used units

Table 17.1: Used units by unit 'fpmimeTypes'

Name	Page
Classes	??
Contrns	<a href="#">200</a>
sysutils	??

### 17.2 Overview

The `fpmimeTypes` unit contains a class which handles mapping of filename extensions to MIME (Multipurpose Internet Mail Extensions) types. The `TFPMimeTypes` ([583](#)) class handles this mapping. A global instance of this class is available through the `MimeTypes` ([582](#)) function. The list of known mime types can be instantiated through the `LoadKnownTypes` ([584](#)) method, or a file in the standard `mime.types` format can be loaded through the `LoadFromFile` ([585](#)) method.

### 17.3 Procedures and functions

#### 17.3.1 MimeTypes

Synopsis: Global `TFPMimeTypes` instance

Declaration: `function MimeTypes : TFPMimeTypes`

Visibility: default

Description: `MimeTypes` returns a global instance of the `TFPMimeTypes` ([583](#)) class. It is not initialized with a list of extensions, so this instance must still be explicitly initialized with `TFPMimeTypes.LoadKnownTypes` ([584](#))

This function is not thread-safe, so be sure to call it once from the main thread and initialize the resulting list.

See also: `TFPMimeTypes` ([583](#)), `TFPMimeTypes.LoadKnownTypes` ([584](#))

## 17.4 TFPMimeType

### 17.4.1 Description

TFPMimeTypes manages a list of MIME types.

The list of types can be initialized with the OS list of known MIME types through the LoadKnownTypes (584) method, or a file in the standard mime.types format can be loaded through the LoadFromFile (585) method.

The associated mime type of a file extension can be retrieved with TFPMimeTypes.GetMimeType (585).

See also: TFPMimeTypes.LoadKnownTypes (584), TFPMimeTypes.LoadFromFile (585), TFPMimeTypes.GetMimeType (585)

### 17.4.2 Method overview

Page	Method	Description
<a href="#">585</a>	AddType	Add a MIME type to the list.
<a href="#">584</a>	Clear	Clear the list of known MIME types
<a href="#">583</a>	Create	Create a new instance of the TFPMimeTypes class
<a href="#">583</a>	Destroy	Remove instance from memory
<a href="#">586</a>	GetKnownExtensions	Get a list of all known extensions
<a href="#">586</a>	GetKnownMimeType	Get a list of all known MIME types
<a href="#">585</a>	GetMimeTypeExtensions	Get the extensions associated with a MIME type
<a href="#">585</a>	GetMimeType	Get MIME type of an extension
<a href="#">584</a>	GetNextExtension	Extract an extension from an extension list as returned by GetMimeTypeExtensions
<a href="#">585</a>	LoadFromFile	Load mime types from a file in mime.types file format
<a href="#">584</a>	LoadFromStream	Load mime types from a stream in mime.types file format
<a href="#">584</a>	LoadKnownTypes	Queries the OS for a list of known MIME types

### 17.4.3 TFPMimeTypes.Create

Synopsis: Create a new instance of the TFPMimeTypes class

Declaration: constructor Create(AOwner: TComponent); Override

Visibility: public

Description: The Create method sets up the necessary internal structures.

See also: TFPMimeTypes.Destroy (583)

### 17.4.4 TFPMimeTypes.Destroy

Synopsis: Remove instance from memory

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy destroys the TFPMimeTypes instance and removes it from memory.

See also: TFPMimeTypes.Create (583)



### 17.4.5 TFPMimeType.Clear

Synopsis: Clear the list of known MIME types

Declaration: `procedure Clear`

Visibility: `public`

Description: `Clear` clears the list of known mime types.

See also: `TFPMimeTypes.LoadKnownTypes` ([584](#))

### 17.4.6 TFPMimeType.LoadKnownTypes

Synopsis: Queries the OS for a list of known MIME types

Declaration: `procedure LoadKnownTypes; Virtual`

Visibility: `public`

Description: `LoadKnownTypes` uses the default mechanism of the OS to initialize the list of MIME types. Under windows, this loads a list of known extensions from the registry (under `HKEY_CLASSES_ROOT`) and attempts to load a `mime.types` located next to the application binary. Under unixlike OS-es, the system location for the `mime.types` is used to load the `mime.types` file.

See also: `TFPMimeTypes.LoadFromFile` ([585](#))

### 17.4.7 TFPMimeType.GetNextExtension

Synopsis: Extract an extension from an extension list as returned by `GetMimeExtensions`

Declaration: `class function GetNextExtension(var E: string) : string`

Visibility: `public`

Description: `GetNextExtension` is a helper function that extracts and returns the next extension from the semicolon separated list of extensions `E` and removes the extension from the list. If there is no more extension, the empty string is returned.

See also: `TFPMimeTypes.GetMimeExtensions` ([585](#))

### 17.4.8 TFPMimeType.LoadFromStream

Synopsis: Load mime types from a stream in `mime.types` file format

Declaration: `procedure LoadFromStream(const Stream: TStream); Virtual`

Visibility: `public`

Description: `LoadFromStream` parses the stream for MIME type definitions and extensions and adds them to the list of known MIME types. The stream is expected to have the text format of the `mime.types` as found in unix systems.

See also: `TFPMimeTypes.LoadFromFile` ([585](#))

### 17.4.9 TFPMimeType.LoadFromFile

Synopsis: Load mime types from a file in mime.types file format

Declaration: `procedure LoadFromFile(const AFileName: string)`

Visibility: public

Description: `LoadFromFile` loads the file `aFileName` and parses the file for MIME type definitions and extensions and adds them to the list of known MIME types. The file is expected to have the text format of the `mime.types` as found in unix systems.

Errors: if the file `aFileName` does not exist, an exception is raised.

See also: `TFPMimeTypes.LoadFromStream` ([584](#))

### 17.4.10 TFPMimeType.AddType

Synopsis: Add a MIME type to the list.

Declaration: `procedure AddType(const AMimeType: string; const AExtensions: string)`

Visibility: public

Description: `AddType` can be used to add `AMimeType` to the list of known MIME types, and associate a semicolon-separated list of extensions `AExtensions` with it. If `AMimeType` is already present in the list of MIME types, then the list of extensions in `AExtensions` is merged with the existing extensions. The extensions may not have a dot character prepended to them.

See also: `TFPMimeTypes.GetMimeExtensions` ([585](#)), `TFPMimeTypes.GetMimeType` ([585](#))

### 17.4.11 TFPMimeType.GetMimeExtensions

Synopsis: Get the extensions associated with a MIME type

Declaration: `function GetMimeExtensions(const AMimeType: string) : string`

Visibility: public

Description: `GetMimeExtensions` returns the list of extensions associated with a MIME type (`AMimeType`). If none are known, an empty string is returned. `AMimeType` is searched case insensitively.

See also: `TFPMimeTypes.GetMimeType` ([585](#))

### 17.4.12 TFPMimeType.GetMimeType

Synopsis: Get MIME type of an extension

Declaration: `function GetMimeType(const AExtension: string) : string`

Visibility: public

Description: `GetMimeType` returns the MIME type of the extension `AExtension`. The extension is searched case-insensitive. If no MIME type is found, an empty string is returned. The extension may start with a dot character or not.

See also: `TFPMimeTypes.GetMimeExtensions` ([585](#))

### 17.4.13 TFPMimeType.GetKnownMimeType

Synopsis: Get a list of all known MIME types

Declaration: `function GetKnownMimeType (AList: TStrings) : Integer`

Visibility: public

Description: `GetKnownMimeType` fills `AList` with the list of known MIME types (one per line) in random order. It clears the list first, and returns the number of entries added to the list.

See also: `TFPMimeType.GetKnownExtensions` ([586](#))

### 17.4.14 TFPMimeType.GetKnownExtensions

Synopsis: Get a list of all known extensions

Declaration: `function GetKnownExtensions (AList: TStrings) : Integer`

Visibility: public

Description: `GetKnownExtensions` fills `AList` with the list of known extensions (one per line) in random order. It clears the list first, and returns the number of entries added to the list. The extensions do not have a dot (.) character prepended.

See also: `TFPMimeType.GetKnownMimeType` ([586](#))

## 17.5 TMimeType

### 17.5.1 Description

`TMimeType` is a helper class which stores a MIME type and its associated extensions. It should not be necessary to create instances of this class manually, the creation is handled entirely through the `TFPMimeType` ([583](#)) class.

See also: `TMimeType.MimeType` ([587](#)), `TMimeType.Extensions` ([587](#)), `TFPMimeType` ([583](#))

### 17.5.2 Method overview

Page	Method	Description
<a href="#">587</a>	Create	Create a new instance of <code>TMimeType</code>
<a href="#">587</a>	MergeExtensions	Merge extensions in the list of extensions

### 17.5.3 Property overview

Page	Properties	Access	Description
<a href="#">587</a>	Extensions	rw	Semicolon-separated list of extensions associated with <code>MimeType</code>
<a href="#">587</a>	MimeType	rw	Mime type name

### 17.5.4 TMimeType.Create

Synopsis: Create a new instance of TMimeType

Declaration: `constructor Create(const AMimeType: string; const AExtensions: string)`

Visibility: public

Description: `Create` initializes a new instance of TMimeType and sets the TMimeType.MimeType (587) TMimeType.Extensions (587) properties to `aMimeType` and `aExtensions`.

See also: TMimeType.MimeType (587), TMimeType.Extensions (587)

### 17.5.5 TMimeType.MergeExtensions

Synopsis: Merge extensions in the list of extensions

Declaration: `procedure MergeExtensions(AExtensions: string)`

Visibility: public

Description: `MergeExtensions` merges the comma-separated list of extensions in `AExtensions` into TMimeType.Extensions (587) in such a way that there are no duplicates.

See also: TMimeType.Extensions (587)

### 17.5.6 TMimeType.MimeType

Synopsis: Mime type name

Declaration: `Property MimeType : string`

Visibility: public

Access: Read,Write

Description: `MimeType` is the lowercase name of the mime type.

See also: TMimeType.Extensions (587)

### 17.5.7 TMimeType.Extensions

Synopsis: Semicolon-separated list of extensions associated with MimeType

Declaration: `Property Extensions : string`

Visibility: public

Access: Read,Write

Description: `Extensions` is the comma-separated list of extensions that is associated with MimeType (587)

See also: TMimeType.MimeType (587)

## Chapter 18

# Reference for unit 'fptimer'

### 18.1 Used units

Table 18.1: Used units by unit 'fptimer'

Name	Page
Classes	??

### 18.2 Overview

The `fpTimer` unit implements a timer class `TFPTimer` (590) which can be used on all supported platforms. The timer class uses a driver class `TFPTimerDriver` (592) which does the actual work.

A default timer driver class is implemented on all platforms. It will work in GUI and non-gui applications, but only in the application's main thread.

An alternative driver class can be used by setting the `DefaultTimerDriverClass` (588) variable to the class pointer of the driver class. The driver class should descend from `TFPTimerDriver` (592).

### 18.3 Constants, types and variables

#### 18.3.1 Types

```
TFPTimerDriverClass = Class of TFPTimerDriver
```

`TFPTimerDriverClass` is the class pointer of `TFPTimerDriver` (592) it exists mainly for the purpose of being able to set `DefaultTimerDriverClass` (588), so a custom timer driver can be used for the timer instances.

#### 18.3.2 Variables

```
DefaultTimerDriverClass : TFPTimerDriverClass = Nil
```

`DefaultTimerDriverClass` contains the `TFPTimerDriver` (592) class pointer that should be used when a new instance of `TFPCustomTimer` (589) is created. It is by default set to the system timer class.

Setting this class pointer to another descendent of `TFPTimerDriver` allows to customize the default timer implementation used in the entire application.

## 18.4 TFPCustomTimer

### 18.4.1 Description

`TFPCustomTimer` is the timer class containing the timer's implementation. It relies on an extra driver instance (of type `TFPTimerDriver` ([592](#))) to do the actual work.

`TFPCustomTimer` publishes no events or properties, so it is unsuitable for handling in an IDE. The `TFPTimer` ([590](#)) descendent class publishes all needed events of `TFPCustomTimer`.

See also: `TFPTimerDriver` ([592](#)), `TFPTimer` ([590](#))

### 18.4.2 Method overview

Page	Method	Description
<a href="#">589</a>	Create	Create a new timer
<a href="#">589</a>	Destroy	Release a timer instance from memory
<a href="#">589</a>	StartTimer	Start the timer
<a href="#">590</a>	StopTimer	Stop the timer

### 18.4.3 TFPCustomTimer.Create

Synopsis: Create a new timer

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

Description: `Create` instantiates a new `TFPCustomTimer` instance. It creates the timer driver instance from the `DefaultTimerDriverClass` class pointer.

See also: `TFPCustomTimer.Destroy` ([589](#))

### 18.4.4 TFPCustomTimer.Destroy

Synopsis: Release a timer instance from memory

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` releases the timer driver component from memory, and then calls `Inherited` to clean the `TFPCustomTimer` instance from memory.

See also: `TFPCustomTimer.Create` ([589](#))

### 18.4.5 TFPCustomTimer.StartTimer

Synopsis: Start the timer

Declaration: `procedure StartTimer; Virtual`

Visibility: public

Description: `StartTimer` starts the timer. After a call to `StartTimer`, the timer will start producing timer ticks.

The timer stops producing ticks only when the `StopTimer` (590) event is called.

See also: `StopTimer` (590), `Enabled` (590), `OnTimer` (591)

### 18.4.6 `TFPCustomTimer.StopTimer`

Synopsis: Stop the timer

Declaration: `procedure StopTimer; Virtual`

Visibility: public

Description: `StopTimer` stops a started timer. After a call to `StopTimer`, the timer no longer produces timer ticks.

See also: `StartTimer` (589), `Enabled` (590), `OnTimer` (591)

## 18.5 `TFPTimer`

### 18.5.1 Description

`TFPTimer` implements no new events or properties, but merely publishes events and properties already implemented in `TFPCustomTimer` (589): `Enabled` (590), `OnTimer` (591) and `Interval` (591).

The `TFPTimer` class is suitable for use in an IDE.

See also: `TFPCustomTimer` (589), `Enabled` (590), `OnTimer` (591), `Interval` (591)

### 18.5.2 Property overview

Page	Properties	Access	Description
590	<code>Enabled</code>		Start or stop the timer
591	<code>Interval</code>		Timer tick interval in milliseconds.
591	<code>OnStartTimer</code>		
591	<code>OnStopTimer</code>		
591	<code>OnTimer</code>		Event called on each timer tick.
591	<code>UseTimerThread</code>		

### 18.5.3 `TFPTimer.Enabled`

Synopsis: Start or stop the timer

Declaration: `Property Enabled :`

Visibility: published

Access:

Description: `Enabled` controls whether the timer is active. Setting `Enabled` to `True` will start the timer (calling `StartTimer` (589)), setting it to `False` will stop the timer (calling `StopTimer` (590)).

See also: `StartTimer` (589), `StopTimer` (590), `OnTimer` (591), `Interval` (591)

### 18.5.4 TFPTimer.Interval

Synopsis: Timer tick interval in milliseconds.

Declaration: `Property Interval :`

Visibility: published

Access:

Description: `Interval` specifies the timer interval in milliseconds. Every `Interval` milliseconds, the `OnTimer` (591) event handler will be called.

Note that the milliseconds interval is a minimum interval. Under high system load, the timer tick may arrive later.

See also: `OnTimer` (591), `Enabled` (590)

### 18.5.5 TFPTimer.UseTimerThread

Declaration: `Property UseTimerThread :`

Visibility: published

Access:

### 18.5.6 TFPTimer.OnTimer

Synopsis: Event called on each timer tick.

Declaration: `Property OnTimer :`

Visibility: published

Access:

Description: `OnTimer` is called on each timer tick. The event handler must be assigned to a method that will do the actual work that should occur when the timer fires.

See also: `Interval` (591), `Enabled` (590)

### 18.5.7 TFPTimer.OnStartTimer

Declaration: `Property OnStartTimer :`

Visibility: published

Access:

### 18.5.8 TFPTimer.OnStopTimer

Declaration: `Property OnStopTimer :`

Visibility: published

Access:



## 18.6 TFPTimerDriver

### 18.6.1 Description

`TFPTimerDriver` is the abstract timer driver class: it simply provides an interface for the `TFP-CustomTimer` (589) class to use.

The `fpTimer` unit implements a descendent of this class which implements the default timer mechanism.

See also: `TFPCustomTimer` (589), `DefaultTimerDriverClass` (588)

### 18.6.2 Method overview

Page	Method	Description
<a href="#">592</a>	Create	Create new driver instance
<a href="#">592</a>	StartTimer	Start the timer
<a href="#">593</a>	StopTimer	Stop the timer

### 18.6.3 Property overview

Page	Properties	Access	Description
<a href="#">593</a>	Timer	r	Timer tick
<a href="#">593</a>	TimerStarted	r	

### 18.6.4 TFPTimerDriver.Create

Synopsis: Create new driver instance

Declaration: `constructor Create(ATimer: TFPCustomTimer); Virtual`

Visibility: public

Description: `Create` should be overridden by descendents of `TFPTimerDriver` to do additional initialization of the timer driver. `Create` just stores (in `Timer` (593)) a reference to the `ATimer` instance which created the driver instance.

See also: `Timer` (593), `TFPTimer` (590)

### 18.6.5 TFPTimerDriver.StartTimer

Synopsis: Start the timer

Declaration: `procedure StartTimer; Virtual; Abstract`

Visibility: public

Description: `StartTimer` is called by `TFPCustomTimer.StartTimer` (589). It should be overridden by descendents of `TFPTimerDriver` to actually start the timer.

See also: `TFPCustomTimer.StartTimer` (589), `TFPTimerDriver.StopTimer` (593)

### 18.6.6 TFPTimerDriver.StopTimer

Synopsis: Stop the timer

Declaration: `procedure StopTimer; Virtual; Abstract`

Visibility: `public`

Description: `StopTimer` is called by `TFPCustomTimer.StopTimer` ([590](#)). It should be overridden by descendants of `TFPTimerDriver` to actually stop the timer.

See also: `TFPCustomTimer.StopTimer` ([590](#)), `TFPTimerDriver.StartTimer` ([592](#))

### 18.6.7 TFPTimerDriver.Timer

Synopsis: Timer tick

Declaration: `Property Timer : TFPCustomTimer`

Visibility: `public`

Access: `Read`

Description: `Timer` calls the `TFPCustomTimer` ([589](#)) timer event. Descendants of `TFPTimerDriver` should call `Timer` whenever a timer tick occurs.

See also: `TFPTimer.OnTimer` ([591](#)), `TFPTimerDriver.StartTimer` ([592](#)), `TFPTimerDriver.StopTimer` ([593](#))

### 18.6.8 TFPTimerDriver.TimerStarted

Declaration: `Property TimerStarted : Boolean`

Visibility: `public`

Access: `Read`

## Chapter 19

# Reference for unit 'gettext'

### 19.1 Used units

Table 19.1: Used units by unit 'gettext'

Name	Page
Classes	??
sysutils	??

### 19.2 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` ([597](#))) 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.

### 19.3 Constants, types and variables

#### 19.3.1 Constants

```
MOFileHeaderMagic = $950412DE
```

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

#### 19.3.2 Types

```
PLongWordArray = ^TLongWordArray
```

Pointer to a `TLongWordArray` ([595](#)) array.

```
PMOStringTable = ^TMOStringTable
```

Pointer to a `TMOStringTable` ([595](#)) array.

```
PPCharArray = ^TPCharArray
```

Pointer to a TPCharArray (595) array.

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

TLongWordArray is an array used to define the PLongWordArray (594) 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.

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

TMOStringTable is an array type containing TMOStringInfo (596) records. It should never be used directly, as it would occupy too much memory.

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

TLongWordArray is an array used to define the PPCharArray (595) 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.

## 19.4 Procedures and functions

### 19.4.1 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.

### 19.4.2 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 (597) 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 (596), TMOFile (597)

### 19.4.3 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` (595), 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` (595), `TMOFile` (597)

## 19.5 TMOFileHeader

```
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.

## 19.6 TMOStringInfo

```
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` (596) record at the start of the file.

## 19.7 EMOFileError

### 19.7.1 Description

`EMOFileError` is raised in case an `TMOFile` (597) instance is created with an invalid .mo.

See also: `TMOFile` (597)



**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 20

# Reference for unit 'IBConnection'

### 20.1 Used units

Table 20.1: Used units by unit 'IBConnection'

Name	Page
BufDataset	<a href="#">130</a>
Classes	??
DB	<a href="#">316</a>
dbconst	??
ibase60dyn	??
SQLDB	<a href="#">769</a>
sysutils	??

### 20.2 Constants, types and variables

#### 20.2.1 Constants

`DEFDIALECT = 3`

Default dialect that will be used when connecting to databases. See `TIBConnection.Dialect` ([604](#)) for more details on dialects.

`MAXBLOBSEGMENTSIZ = 65535`

#### 20.2.2 Types

### 20.3 TDatabaseInfo

```
TDatabaseInfo = record
  Dialect : Integer;
  ODSMajorVersion : Integer
;
```



```

    ServerVersion : string;
    ServerVersionString : string;
end

```

## 20.4 EIBDatabaseError

### 20.4.1 Description

Firebird/Interbase database error, a descendant of `db.EDatabaseError` (599).

See also: `db.EDatabaseError` (599)

### 20.4.2 Property overview

Page	Properties	Access	Description
<a href="#">600</a>	<code>GDSErrorCode</code>	<code>r</code>	Firebird/Interbase GDS error code.

### 20.4.3 EIBDatabaseError.GDSErrorCode

Synopsis: Firebird/Interbase GDS error code.

Declaration: `Property GDSErrorCode : Integer; deprecated;`

Visibility: `public`

Access: `Read`

Description: Firebird/Interbase-specific error code, the GDS error code. From a Firebird perspective: Firebird throws 2 error codes for an exception. The high-level one is the `SQLCODE`, which is a negative 3-digit code. The lower-level one is the `ISC` code (or `GDSCODE`) which has 9 digits. Related `ISC` error types are grouped under the same `SQLCODE`. In some cases, each successive `gdscode` error gives you further information about the error condition. Note: `SQLCODE` is a deprecated SQL standard; its successor is `SQLSTATE`.

## 20.5 TIBConnection

### 20.5.1 Description

`TIBConnection` is a descendant of `TSQLConnection` (599) and represents a connection to a Firebird/Interbase server.

It is designed to work with Interbase 6, Firebird 1 and newer database servers.

`TIBConnection` by default requires the Firebird/Interbase client library (e.g. `gds32.dll`, `libfbclient.so`, `fbclient.dll`, `fbembed.dll`) and its dependencies to be installed on the system. The bitness between library and your application must match: e.g. use 32 bit `fbclient` when developing a 32 bit application on 64 bit Linux.

On Windows, in accordance with the regular Windows way of loading DLLs, the library can also be in the executable directory. In fact, this directory is searched first, and might be a good option for distributing software to end users as it eliminates problems with incompatible DLL versions.

`TIBConnection` is based on FPC Interbase/Firebird code (`ibase60.inc`) that tries to load the client library. If you want to use Firebird embedded, make sure the embedded library is searched/loaded first. There are several ways to do this:

- Include `ibase60` in your `uses` clause, set `UseEmbeddedFirebird` to `true`
- On Windows, with FPC newer than 2.5.1, put `fbembed.dll` in your application directory
- On Windows, put the `fbembed.dll` in your application directory and rename it to `fbclient.dll`

Pre 2.5.1 versions of FPC did not try to load the `fbembed` library by default. See [FPC bug 17664](#) for more details.

An indication of which DLLs need to be installed on Windows (Firebird 2.5, differs between versions:

- `fbclient.dll` (or `fbembed.dll`)
- `firebird.msg`
- `ib_util.dll`
- `icudt30.dll`
- `icuin30.dll`
- `icuuc30.dll`
- `msvcp80.dll`
- `msvcr80.dll`

Please see your database documentation for details.

The `TIBConnection` component does not reliably detect computed fields as such. This means that automatically generated update SQL statements will attempt to update these fields, resulting in SQL errors. These errors can be avoided by removing the `pfInUpdate` flag from the `provideroptions` from a field, once it has been created:

```
MyQuery.FieldName('full_name').ProviderFlags:=[];
```

See also: `TSQLConnection` ([599](#))

## 20.5.2 Method overview

Page	Method	Description
<a href="#">602</a>	<code>Create</code>	Creates a <code>TIBConnection</code> object
<a href="#">602</a>	<code>CreateDB</code>	Creates a database on disk
<a href="#">603</a>	<code>DropDB</code>	Deletes a database from disk
<a href="#">602</a>	<code>GetConnectionInfo</code>	

### 20.5.3 Property overview

Page	Properties	Access	Description
<a href="#">603</a>	BlobSegmentSize	rw	Write this amount of bytes per BLOB segment
<a href="#">604</a>	CheckTransactionParams	rw	
<a href="#">603</a>	DatabaseName		Name of the database to connect to
<a href="#">604</a>	Dialect	rws	Database dialect
<a href="#">604</a>	KeepConnection		Keep open connection after first query
<a href="#">604</a>	LoginPrompt		Switch for showing custom login prompt
<a href="#">603</a>	ODSMajorVersion	r	
<a href="#">605</a>	OnLogin		Event triggered when a login prompt needs to be shown.
<a href="#">605</a>	Params		Firebird/Interbase specific parameters
<a href="#">605</a>	Port	s	
<a href="#">605</a>	UseConnectionCharSetIfNone	rw	
<a href="#">605</a>	WireCompression	rw	

### 20.5.4 TIBConnection.Create

Synopsis: Creates a `TIBConnection` object

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: Creates a `TIBConnection` object

### 20.5.5 TIBConnection.GetConnectionInfo

Declaration: `function GetConnectionInfo(InfoType: TConnInfoType) : string; Override`

Visibility: `public`

### 20.5.6 TIBConnection.CreateDB

Synopsis: Creates a database on disk

Declaration: `procedure CreateDB; Override`

Visibility: `public`

Description: Instructs the Interbase or Firebird database server to create a new database.

If set, the `TSQLConnection.Params` ([599](#)) (specifically, `PAGE_SIZE`) and `TSQLConnection.CharSet` ([599](#)) properties influence the database creation.

If creating a database using a client/server environment, the `TIBConnection` code will connect to the database server before trying to create the database. Therefore make sure the connection properties are already correctly set, e.g. `TSQLConnection.HostName` ([599](#)), `TSQLConnection.UserName` ([599](#)), `TSQLConnection.Password` ([599](#)).

If creating a database using Firebird embedded, make sure the embedded library is loaded, the `TSQLConnection.HostName` ([599](#)) property is empty, and set the `TSQLConnection.UserName` ([599](#)) to e.g. 'SYSDBA'. See `TIBConnection: Firebird/Interbase specific TSQLConnection` ([599](#)) for details on loading the embedded database library.

See also: `TSQLConnection.Params` ([599](#)), `TSQLConnection.DropDB` ([599](#)), `TIBConnection` ([600](#))

### 20.5.7 TIBConnection.DropDB

Synopsis: Deletes a database from disk

Declaration: `procedure DropDB; Override`

Visibility: `public`

Description: `DropDB` instructs the Interbase/Firebird database server to delete the database that is specified in the `TIBConnection` (600).

In a client/server environment, the `TIBConnection` code will connect to the database server before telling it to drop the database. Therefore make sure the connection properties are already correctly set, e.g. `TSQLConnection.HostName` (599), `TSQLConnection.UserName` (599), `TSQLConnection.Password` (599).

When using Firebird embedded, make sure the embedded connection library is loaded, the `TSQLConnection.HostName` (599) property is empty, and set the `TSQLConnection.UserName` (599) to e.g. 'SYSDBA'. See `TIBConnection: Firebird/Interbase specific TSQLConnection` (599) for more details on loading the embedded library.

See also: `TSQLConnection.CreateDB` (599), `TSQLConnection.HostName` (599), `TSQLConnection.UserName` (599), `TSQLConnection.Password` (599)

### 20.5.8 TIBConnection.BlobSegmentSize

Synopsis: Write this amount of bytes per BLOB segment

Declaration: `Property BlobSegmentSize : Word; deprecated;`

Visibility: `public`

Access: `Read,Write`

Description: **Deprecated** since FPC 2.7.1 revision 19659

When sending BLOBs to the database, the code writes them in segments.

Before FPC 2.7.1 revision 19659, these segments were 80 bytes and could be changed using `BlobSegmentSize`. Please set `BlobSegmentSize` to 65535 for better write performance.

In newer FPC versions, the `BlobSegmentSize` property is ignored and segments of 65535 bytes are always used.

### 20.5.9 TIBConnection.ODSMajorVersion

Declaration: `Property ODSMajorVersion : Integer`

Visibility: `public`

Access: `Read`

### 20.5.10 TIBConnection.DatabaseName

Synopsis: Name of the database to connect to

Declaration: `Property DatabaseName :`

Visibility: `published`

Access:

**Description:** Name of the Interbase/Firebird database to connect to.

This can be either the path to the database or an alias name. Please see your database documentation for details.

In a client/server environment, the name indicates the location of the database on the server's file system, so if you have a Linux Firebird server, you might have something like /var/lib/firebird/2.5/data/employee.fdb

If using an embedded Firebird database, the name is a relative path relative to the fbembed library.

Note that the path is specified as an Ansistring, meaning that databases residing in directories that rely on Unicode characters will not work. (firebird itself also cannot handle this).

### 20.5.11 TIBConnection.Dialect

**Synopsis:** Database dialect

**Declaration:** Property Dialect : Integer

**Visibility:** published

**Access:** Read,Write

**Description:** Firebird/Interbase servers since Interbase 6 have a dialect setting for backwards compatibility. It can be 1, 2 or 3, the default is 3.

**Note:** the dialect for new Interbase/Firebird databases is 3; dialects 1 and 2 are only used in legacy environments. In practice, you can ignore this setting for newly created databases.

### 20.5.12 TIBConnection.CheckTransactionParams

**Declaration:** Property CheckTransactionParams : Boolean

**Visibility:** published

**Access:** Read,Write

### 20.5.13 TIBConnection.KeepConnection

**Synopsis:** Keep open connection after first query

**Declaration:** Property KeepConnection :

**Visibility:** published

**Access:**

**Description:** Determines whether to keep the connection open once it is established and the first query has been executed.

### 20.5.14 TIBConnection.LoginPrompt

**Synopsis:** Switch for showing custom login prompt

**Declaration:** Property LoginPrompt :

**Visibility:** published

**Access:**

**Description:** If true, the OnLogin (599) event will fire, allowing you to handle supplying of credentials yourself.

**See also:** TSQLConnection.OnLogin (599)

### 20.5.15 TIBConnection.Params

Synopsis: Firebird/Interbase specific parameters

Declaration: `Property Params :`

Visibility: published

Access:

Description: `Params` is a `#rtl.classes.TStringList` (??) of name=value combinations that set database-specific parameters.

The following parameter is supported:

- `PAGE_SIZE`: size of database pages (an integer), e.g. 16384.

See your database documentation for more details.

See also: `#fcl.sqlldb.TSQLConnection.Params` ([807](#))

### 20.5.16 TIBConnection.OnLogin

Synopsis: Event triggered when a login prompt needs to be shown.

Declaration: `Property OnLogin :`

Visibility: published

Access:

Description: `OnLogin` is triggered when the connection needs a login prompt when connecting: it is triggered when the `LoginPrompt` ([599](#)) property is `True`, after the `BeforeConnect` ([361](#)) event, but before the connection is actually established.

See also: `#fcl.db.TCustomConnection.BeforeConnect` ([361](#)), `TSQLConnection.LoginPrompt` ([599](#)), `#fcl.db.TCustomConnection.Open` ([359](#)), `TSQLConnection.OnLogin` ([599](#))

### 20.5.17 TIBConnection.Port

Declaration: `Property Port :`

Visibility: published

Access:

### 20.5.18 TIBConnection.UseConnectionCharSetIfNone

Declaration: `Property UseConnectionCharSetIfNone : Boolean`

Visibility: published

Access: Read,Write

### 20.5.19 TIBConnection.WireCompression

Declaration: `Property WireCompression : Boolean`

Visibility: published

Access: Read,Write

## 20.6 TIBConnectionDef

### 20.6.1 Description

Child of TConnectionDef (599) used to register an Interbase/Firebird connection, so that it is available in "connection factory" scenarios where database drivers/connections are loaded at runtime and it is unknown at compile time whether the required database libraries are present on the end user's system.

See also: TConnectionDef (599)

### 20.6.2 Method overview

Page	Method	Description
606	ConnectionClass	Firebird/Interbase child of ConnectionClass (785)
607	DefaultLibraryName	
606	Description	Description for the Firebird/Interbase child of #fcl.sqlldb.TConnectionDef.ConnectionClass (785)
607	LoadedLibraryName	
607	LoadFunction	
606	TypeName	Firebird/Interbase child of TConnectionDef.TypeName (599)
607	UnLoadFunction	

### 20.6.3 TIBConnectionDef.TypeName

Synopsis: Firebird/Interbase child of TConnectionDef.TypeName (599)

Declaration: `class function TypeName : string; Override`

Visibility: default

See also: TConnectionDef.TypeName (599), TIBConnection (600)

### 20.6.4 TIBConnectionDef.ConnectionClass

Synopsis: Firebird/Interbase child of ConnectionClass (785)

Declaration: `class function ConnectionClass : TSQLConnectionClass; Override`

Visibility: default

See also: TConnectionDef.ConnectionClass (599), TIBConnection (600)

### 20.6.5 TIBConnectionDef.Description

Synopsis: Description for the Firebird/Interbase child of #fcl.sqlldb.TConnectionDef.ConnectionClass (785)

Declaration: `class function Description : string; Override`

Visibility: default

Description: The description identifies this ConnectionDef object as a Firebird/Interbase connection.

See also: #fcl.sqlldb.TConnectionDef.Description (786), TIBConnection (600)

### 20.6.6 TIBConnectionDef.DefaultLibraryName

Declaration: `class function DefaultLibraryName : string; Override`

Visibility: default

### 20.6.7 TIBConnectionDef.LoadFunction

Declaration: `class function LoadFunction : TLibraryLoadFunction; Override`

Visibility: default

### 20.6.8 TIBConnectionDef.UnLoadFunction

Declaration: `class function UnLoadFunction : TLibraryUnLoadFunction; Override`

Visibility: default

### 20.6.9 TIBConnectionDef.LoadedLibraryName

Declaration: `class function LoadedLibraryName : string; Override`

Visibility: default

## 20.7 TIBCursor

### 20.7.1 Description

A cursor that keeps track of where you are in a Firebird/Interbase dataset. It is a descendent of [TSQLCursor \(599\)](#).

See also: [TSQLCursor \(599\)](#), [TIBConnection \(600\)](#)

## 20.8 TIBTrans

### 20.8.1 Description

Firebird/Interbase database transaction object. Descendant of [TSQLHandle \(599\)](#).

See also: [TSQLHandle \(599\)](#), [TIBConnection \(600\)](#)



# Chapter 21

## Reference for unit 'idea'

### 21.1 Used units

Table 21.1: Used units by unit 'idea'

Name	Page
Classes	??
sysutils	??

### 21.2 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.

### 21.3 Constants, types and variables

#### 21.3.1 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

### 21.3.2 Types

`IdeaCryptData = TIdeaCryptData`

Provided for backward functionality.

`IdeaCryptKey = TIdeaCryptKey`

Provided for backward functionality.

`IDEAkey = TIDEAKey`

Provided for backward functionality.

`Array = Array[0..3] of Word`

`TIdeaCryptData` is an internal type, defined to hold data for encryption/decryption.

`Array = 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` (610) or `DeKeyIDEA` (609) algorithms depending on whether an encryption or decryption key is needed.

`Array = 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.

## 21.4 Procedures and functions

### 21.4.1 CipherIdea

**Synopsis:** Encrypt or decrypt a buffer.

**Declaration:** `procedure CipherIdea(Input: TIdeaCryptData;  
out 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` (610), `DeKeyIdea` (609), `TIDEAEncryptStream` (612), `TIDEADecryptStream` (610)

### 21.4.2 DeKeyIdea

**Synopsis:** Create a decryption key from an encryption key.

**Declaration:** `procedure DeKeyIdea(z: TIDEAKey; out 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` (610), `CipherIdea` (609)

### 21.4.3 EnKeyIdea

**Synopsis:** Create an IDEA encryption key from a user key.

**Declaration:** `procedure EnKeyIdea (UserKey: TIDEACryptKey; out 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` (609), `CipherIdea` (609)

## 21.5 EIDEAError

### 21.5.1 Description

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

## 21.6 TIDEADeCryptStream

### 21.6.1 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` (609) 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` (612) encryption stream.

See also: `TIDEAEncryptStream` (612), `TIDEAStream.Create` (614), `CipherIDEA` (609)

### 21.6.2 Method overview

Page	Method	Description
611	Create	Constructor to create a new <code>TIDEADeCryptStream</code> instance
611	Read	Reads data from the stream, decrypting it as needed
611	Seek	Set position on the stream

### 21.6.3 TIDEADeCryptStream.Create

Synopsis: Constructor to create a new `TIDEADeCryptStream` instance

Declaration: `constructor Create(const AKey: string; Dest: TStream); Overload`

Visibility: `public`

Description: `Create` creates a new `TIDEADeCryptStream` instance using the string `AKey` to compute the encryption key (609), which is then passed on to the inherited constructor `TIDEAStream.Create` (614). It is an easy-access function which introduces no new functionality.

The string is truncated at the maximum length of the `TIdeaCryptKey` (609) structure, so it makes no sense to provide a string with length longer than this structure.

See also: `TIdeaCryptKey` (609), `TIDEAStream.Create` (614), `TIDEAEnCryptStream.Create` (612)

### 21.6.4 TIDEADeCryptStream.Read

Synopsis: Reads data from the stream, decrypting it as needed

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

Visibility: `public`

Description: `Read` attempts to read `Count` bytes from the stream, placing them in `Buffer` the bytes are read from the source stream and decrypted as they are read. (bytes are read from the source stream in blocks of 8 bytes. The function returns the number of bytes actually read.

Errors: If an error occurs when reading data from the source stream, an exception may be raised.

See also: `Write` (610), `Seek` (611), `TIDEAEncryptStream` (612)

### 21.6.5 TIDEADeCryptStream.Seek

Synopsis: Set position on the stream

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; 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` (610) exception is raised if the stream does not allow the requested seek operation.

See also: `Read` (611)

## 21.7 TIDEAEncryptStream

### 21.7.1 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` (609) 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` (610) decryption stream.

See also: `TIDEADecryptStream` (610), `TIDEAStream.Create` (614), `CipherIDEA` (609)

### 21.7.2 Method overview

Page	Method	Description
<a href="#">612</a>	Create	Constructor to create a new <code>TIDEAEncryptStream</code> instance
<a href="#">612</a>	Destroy	Flush data buffers and free the stream instance.
<a href="#">613</a>	Flush	Write remaining bytes from the stream
<a href="#">613</a>	Seek	Set stream position
<a href="#">613</a>	Write	Write bytes to the stream to be encrypted

### 21.7.3 TIDEAEncryptStream.Create

Synopsis: Constructor to create a new `TIDEAEncryptStream` instance

Declaration: `constructor Create(const AKey: string; Dest: TStream); Overload`

Visibility: public

Description: `Create` creates a new `TIDEAEncryptStream` instance using the string `AKey` to compute the encryption key (609), which is then passed on to the inherited constructor `TIDEAStream.Create` (614). It is an easy-access function which introduces no new functionality.

The string is truncated at the maximum length of the `TIdeaCryptKey` (609) structure, so it makes no sense to provide a string with length longer than this structure.

See also: `TIdeaCryptKey` (609), `TIDEAStream.Create` (614), `TIDEADeCryptStream.Create` (611)

### 21.7.4 TIDEAEncryptStream.Destroy

Synopsis: Flush data buffers and free the stream instance.

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` flushes any data still remaining in the internal encryption buffer, and then calls the inherited `Destroy`

By default, the destination stream is not freed when the encryption stream is freed.

Errors: None.

See also: `TIDEAStream.Create` (614)

### 21.7.5 TIDEAEncryptStream.Write

Synopsis: Write bytes to the stream to be encrypted

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

Visibility: public

Description: `Write` writes `Count` bytes from `Buffer` to the stream, encrypting the bytes as they are written (encryption in blocks of 8 bytes).

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

See also: `Read` ([611](#))

### 21.7.6 TIDEAEncryptStream.Seek

Synopsis: Set stream position

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

Visibility: public

Description: `Seek` return the current position if called with 0 and `soFromCurrent` as arguments. With all other values, it will always raise an exception, since it is impossible to set the position on an encryption stream.

Errors: An `EIDEAError` ([610](#)) will be raised unless called with 0 and `soFromCurrent` as arguments.

See also: `Write` ([613](#)), `EIDEAError` ([610](#))

### 21.7.7 TIDEAEncryptStream.Flush

Synopsis: Write remaining bytes from the stream

Declaration: `procedure Flush`

Visibility: public

Description: `Flush` writes the current encryption buffer to the stream. Encryption always happens in blocks of 8 bytes, so if the buffer is not completely filled at the end of the writing operations, it must be flushed. It should never be called directly, unless at the end of all writing operations. It is called automatically when the stream is destroyed.

Errors: None.

See also: `Write` ([613](#))

## 21.8 TIDEAStream

### 21.8.1 Description

Do not create instances of `TIDEAStream` directly. It implements no useful functionality: it serves as a common ancestor of the `TIDEAEncryptStream` ([612](#)) and `TIDEADeCryptStream` ([610](#)), 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.

See also: `TIDEAEncryptStream` ([612](#)), `TIDEADeCryptStream` ([610](#))

### 21.8.2 Method overview

Page	Method	Description
<a href="#">614</a>	Create	Creates a new instance of the <code>TIDEASStream</code> class

### 21.8.3 Property overview

Page	Properties	Access	Description
<a href="#">614</a>	Key	r	Key used when encrypting/decrypting

### 21.8.4 TIDEASStream.Create

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

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

Visibility: public

Description: `Create` stores the encryption/decryption key and then calls the inherited `Create` to store the `Dest` stream.

Errors: None.

See also: `TIDEAEncryptStream` ([612](#)), `TIDEADeCryptStream` ([610](#))

### 21.8.5 TIDEASStream.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` ([609](#))

## Chapter 22

# Reference for unit 'inicol'

### 22.1 Used units

Table 22.1: Used units by unit 'inicol'

Name	Page
Classes	??
IniFiles	<a href="#">625</a>
sysutils	??

### 22.2 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` ([627](#)) class for this.

### 22.3 Constants, types and variables

#### 22.3.1 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.

### 22.4 EIniCol

#### 22.4.1 Description

`EIniCol` is used to report error conditions in the load and save methods of `TIniCollection` ([616](#)).



## 22.5 TIniCollection

### 22.5.1 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` (619).

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.

See also: `TIniCollection.LoadFromFile` (618), `TIniCollection.LoadFromIni` (618), `TIniCollection.SaveToIni` (617), `TIniCollection.SaveToFile` (617)

### 22.5.2 Method overview

Page	Method	Description
616	<code>Load</code>	Loads the collection from the default filename.
618	<code>LoadFromFile</code>	Load collection from file.
618	<code>LoadFromIni</code>	Load collection from a file in .ini file format.
616	<code>Save</code>	Save the collection to the default filename.
617	<code>SaveToFile</code>	Save collection to a file in .ini file format
617	<code>SaveToIni</code>	Save the collection to a <code>TCustomIniFile</code> descendent

### 22.5.3 Property overview

Page	Properties	Access	Description
619	<code>FileName</code>	rw	Filename of the collection
619	<code>GlobalSection</code>	rw	Name of the global section
618	<code>Prefix</code>	r	Prefix used in global section
619	<code>SectionPrefix</code>	r	Prefix string for section names

### 22.5.4 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` (619) property. It calls the `LoadFromFile` (618) 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` (618), `TIniCollection.LoadFromIni` (618), `TIniCollection.Save` (616), `FileName` (619)

### 22.5.5 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` (619) property, using `GlobalSection` (619) as the section. It calls the `SaveToFile` (617) 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` (617), `TIniCollection.SaveToIni` (617), `TIniCollection.Load` (616), `FileName` (619)

### 22.5.6 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` (618) property for the key. After this it calls the `SaveToIni` (620) method of all `TIniCollectionItem` (619) 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` (619) property contains the value of `Section`, it will be used in the `Save` (619) method.

**See also:** `TIniCollectionItem.SaveToIni` (620)

### 22.5.7 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` (617) 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` (617), `TIniCollection.LoadFromFile` (618)

### 22.5.8 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` (620) 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` (617) method description.

See also: `TIniCollection.LoadFromFile` (618), `TIniCollectionItem.LoadFromIni` (620), `TIniCollection.SaveToIni` (617)

### 22.5.9 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` (618) using this instance and `Section` as the parameters.

See also: `TIniCollection.LoadFromIni` (618), `TIniCollection.Load` (616), `TIniCollection.SaveToIni` (617), `TIniCollection.SaveToFile` (617)

### 22.5.10 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` (619), `TIniCollection.GlobalSection` (619)

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

See also: `TIniCollection.Section` (616), `TIniCollection.GlobalSection` (619)

### 22.5.12 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` (618) or `SaveToFile` (617) operation. It is used in the `Load` (616) or `Save` (616) calls.

See also: `Save` (616), `LoadFromFile` (618), `SaveToFile` (617), `Load` (616)

### 22.5.13 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` (618) or `SaveToIni` (617) calls. It's used in the `Load` (616) or `Save` (616) calls.

See also: `Save` (616), `LoadFromFile` (618), `SaveToFile` (617), `Load` (616)

## 22.6 TIniCollectionItem

### 22.6.1 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` (620) and `LoadFromIni` (620) methods should be overridden. They should implement the actual loading and saving. The loading and saving is always initiated by the methods in `TIniCollection` (616), `TIniCollection.LoadFromIni` (618) and `TIniCollection.SaveToIni` (617) respectively.

See also: `TIniCollection` (616), `TIniCollectionItem.SaveToIni` (620), `TIniCollectionItem.LoadFromIni` (620), `TIniCollection.LoadFromIni` (618), `TIniCollection.SaveToIni` (617)

### 22.6.2 Method overview

Page	Method	Description
<a href="#">621</a>	LoadFromFile	Load item from a file
<a href="#">620</a>	LoadFromIni	Method called when the item must be loaded
<a href="#">620</a>	SaveToFile	Save item to a file
<a href="#">620</a>	SaveToIni	Method called when the item must be saved

### 22.6.3 Property overview

Page	Properties	Access	Description
<a href="#">621</a>	SectionName	rw	Default section name

### 22.6.4 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` ([617](#)) 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` ([617](#)), `LoadFromIni` ([620](#)), `SaveToFile` ([620](#)), `LoadFromFile` ([621](#))

### 22.6.5 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` ([618](#)) 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` ([618](#)), `SaveToIni` ([620](#)), `LoadFromFile` ([621](#)), `SaveToFile` ([620](#))

### 22.6.6 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` (620) 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:** `SaveToIni` (620), `LoadFromFile` (621)

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

**Errors:** None.

**See also:** `SaveToFile` (620), `LoadFromIni` (620)

### 22.6.8 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:** `SaveToFile` (620), `LoadFromIni` (620)

## 22.7 TNamedIniCollection

### 22.7.1 Description

`TNamedIniCollection` is the collection to go with the `TNamedIniCollectionItem` (623) item class. it provides some functions to look for items based on the `UserData` (622) or based on the `Name` (622).

**See also:** `TNamedIniCollectionItem` (623), `IndexOfUserData` (622), `IndexOfName` (622)

### 22.7.2 Method overview

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

### 22.7.3 Property overview

Page	Properties	Access	Description
<a href="#">623</a>	NamedItems	rw	Indexed access to the <code>TNamedIniCollectionItem</code> items

### 22.7.4 `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` ([622](#)) 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: `IndexOfName` ([622](#)), `TNamedIniCollectionItem.UserData` ([623](#))

### 22.7.5 `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: `IndexOfUserData` ([622](#)), `TNamedIniCollectionItem.Name` ([624](#))

### 22.7.6 `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` ([622](#)) 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: `IndexOfName` ([622](#)), `FindByUserData` ([623](#))

### 22.7.7 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` (623) property value matches the `UserData` parameter. If no item is found, `Nil` is returned.

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

### 22.7.8 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` (623) items. The index is zero based.

See also: `TNamedIniCollectionItem` (623)

## 22.8 TNamedIniCollectionItem

### 22.8.1 Description

`TNamedIniCollectionItem` is a `TIniCollectionItem` (619) descent with a published name property. The name is used as the section name when saving the item to the ini file.

See also: `TIniCollectionItem` (619)

### 22.8.2 Property overview

Page	Properties	Access	Description
<a href="#">624</a>	<code>Name</code>	<code>rw</code>	Name of the item
<a href="#">623</a>	<code>UserData</code>	<code>rw</code>	User-defined data

### 22.8.3 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`.



#### 22.8.4 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` ([623](#))

## Chapter 23

# Reference for unit 'IniFiles'

### 23.1 Used units

Table 23.1: Used units by unit 'IniFiles'

Name	Page
Classes	??
Contrns	<a href="#">200</a>
sysutils	??

### 23.2 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 ([627](#)) and is implemented in TIniFile ([640](#)) and TMemIniFile ([650](#)). The API presented by these components is Delphi compatible.

### 23.3 Constants, types and variables

#### 23.3.1 Types

```
TIniFileOption = (ifoStripComments, ifoStripInvalid, ifoEscapeLineFeeds  
,  
                 ifoCaseSensitive, ifoStripQuotes,  
                 ifoFormatSettingsActive, ifoWriteStringBoolean)
```

Table 23.2: Enumeration values for type TIniFileOption

Value	Explanation
ifoCaseSensitive	Key and section names are case sensitive
ifoEscapeLineFeeds	Observe backslash as linefeed escape character
ifoFormatSettingsActive	Observe the values in FormatSettings
ifoStripComments	Strip comments from file
ifoStripInvalid	Strip invalid lines from file
ifoStripQuotes	Strip double quotes from values
ifoWriteStringBoolean	Read/Write booleans as strings instead of 0/1

TIniFileOption enumerates the possible options when creating a new TCustomIniFileoption (625) instance.

**ifoStripComments** Strip comments from file

**ifoStripInvalid** Strip invalid lines from file

**ifoEscapeLineFeeds** Observe backslash as linefeed escape character

**ifoCaseSensitive** Key and section names are case sensitive

**ifoStripQuotes** Strip double quotes from values

**ifoFormatSettingsActive** Observe the values in FormatSettings

TIniFileOptions = Set of TIniFileOption

TIniFileOptions is the set for TIniFileOption (625). It is used in the TCustomIniFile.Create (628) constructor and TCustomIniFile.Options (637) property.

```
TSectionValuesOption = (svoIncludeComments, svoIncludeInvalid,
    svoIncludeQuotes)
```

Table 23.3: Enumeration values for type TSectionValuesOption

Value	Explanation
svoIncludeComments	Include comment lines
svoIncludeInvalid	Include invalid lines
svoIncludeQuotes	Include existing quotes around values

TSectionValuesOption is used to control the behaviour of TCustomIniFile.ReadSectionValues (635)

**svoIncludeComments** Include comment lines

**svoIncludeInvalid** Include invalid lines

**svoIncludeQuotes** Include existing quotes around values

TSectionValuesOptions = Set of TSectionValuesOption

TSectionValuesOptions is the set for TSectionValuesOptions (626). It is used in the TCustomIniFile.ReadSectionValues (635) call.

## 23.4 TCustomIniFile

### 23.4.1 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 (640) or TMemIniFile (650).

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

See also: TIniFile (640), TMemIniFile (650)

### 23.4.2 Method overview

Page	Method	Description
628	Create	Instantiate a new instance of TCustomIniFile.
636	DeleteKey	Delete a key from a section
629	Destroy	Remove the TCustomIniFile instance from memory
635	EraseSection	Clear a section
633	ReadBinaryStream	Read binary data
631	ReadBool	
632	ReadDate	Read a date value
632	ReadDateTime	Read a Date/Time value
632	ReadFloat	Read a floating point value
630	ReadInt64	Read Int64 value
630	ReadInteger	Read an integer value from the file
634	ReadSection	Read the key names in a section
635	ReadSections	Read the list of sections
635	ReadSectionValues	Read names and values of a section
629	ReadString	Read a string valued key
632	ReadTime	Read a time value
629	SectionExists	Check if a section exists.
629	SetBoolStringValues	Set the boolean string values to use when writing to file
636	UpdateFile	Update the file on disk
636	ValueExists	Check if a value exists
634	WriteBinaryStream	Write binary data
631	WriteBool	Write boolean value
633	WriteDate	Write date value
633	WriteDateTime	Write date/time value
634	WriteFloat	Write a floating-point value
631	WriteInt64	Write a Int64 value.
630	WriteInteger	Write an integer value
630	WriteString	Write a string value
634	WriteTime	Write time value

### 23.4.3 Property overview

Page	Properties	Access	Description
<a href="#">639</a>	BoolFalseStrings	rw	Strings to recognize as boolean <code>False</code> values.
<a href="#">638</a>	BoolTrueStrings	rw	Strings to recognize as boolean <code>True</code> values.
<a href="#">637</a>	CaseSensitive	rw	Are key and section names case sensitive
<a href="#">636</a>	Encoding	rw	Encoding of the ini file.
<a href="#">637</a>	EscapeLineFeeds	r	Should linefeeds be escaped ?
<a href="#">637</a>	FileName	r	Name of the .ini file
<a href="#">638</a>	FormatSettingsActive	rw	Is <code>FormatSettings</code> used or not
<a href="#">637</a>	Options	rw	Options currently in effect
<a href="#">639</a>	OwnsEncoding	r	Does the ini file instance own the encoding ?
<a href="#">638</a>	StripQuotes	rw	Should quotes be stripped from string values

### 23.4.4 TCustomIniFile.Create

**Synopsis:** Instantiate a new instance of `TCustomIniFile`.

**Declaration:**

```

constructor Create(const AFileName: string;
                  ADefaultEncoding: TEncoding;
                  AOptions: TIniFileOptions)
constructor Create(const AFileName: string;
                  ADefaultEncoding: TEncoding; AOwnsEncoding: Boolean;
                  AOptions: TIniFileOptions)
constructor Create(const AFileName: string; AOptions: TIniFileOptions)
; Virtual
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 `ifEscapeLineFeeds` option is present in `AOptions` or `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)

Default options for the `TCustomIniFile.Options` ([637](#)) property can be specified in `AOptions`.

A Default string encoding can be specified in `aEncoding`. If `aOwnsEncoding` is `True` the encoding will be freed when the ini file instance is destroyed.

**Errors:** If the file cannot be read, an exception may be raised.

See also: `Destroy` ([629](#)), `TCustomIniFile.Options` ([637](#))

### 23.4.5 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 ([627](#))

### 23.4.6 TCustomIniFile.SetBoolStringValue

Synopsis: Set the boolean string values to use when writing to file

Declaration: `procedure SetBoolStringValue(ABoolValue: Boolean;  
Values: Array of string)`

Visibility: `public`

Description: `SetBoolStringValue` Sets the strings to recognize as boolean variable `ABoolValue` to `Values`. When reading boolean values, these values are reported as `ABoolValue` in `TCustomIniFile.ReadBool` ([631](#)). The first value in the list is used to write the boolean `ABoolValue` value.

To ensure correct operation, this function should be called with values for both `True` and `False`.

The string values are ignored if `ifoWriteStringBoolean` is not in the ini file options ([637](#)).

See also: TCustomIniFile.ReadBool ([631](#)), TCustomIniFile.WriteBool ([631](#)), TCustomIniFile.BoolTrueStrings ([638](#)), TCustomIniFile.BoolFalseStrings ([639](#)), TCustomIniFile.Options ([637](#))

### 23.4.7 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 ([636](#))

### 23.4.8 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: `WriteString` ([630](#)), `ReadInteger` ([630](#)), `ReadBool` ([631](#)), `ReadDate` ([632](#)), `ReadDateTime` ([632](#)), `ReadTime` ([632](#)), `ReadFloat` ([632](#)), `ReadBinaryStream` ([633](#))

### 23.4.9 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.

Note that it is not possible to write strings with newline characters in them. Newlines can be read from a .ini file, but there is no support for writing them.

See also: [ReadString \(629\)](#), [WriteInteger \(630\)](#), [WriteBool \(631\)](#), [WriteDate \(633\)](#), [WriteDateTime \(633\)](#), [WriteTime \(634\)](#), [WriteFloat \(634\)](#), [WriteBinaryStream \(634\)](#)

### 23.4.10 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: [WriteInteger \(630\)](#), [ReadString \(629\)](#), [ReadBool \(631\)](#), [ReadDate \(632\)](#), [ReadDateTime \(632\)](#), [ReadTime \(632\)](#), [ReadFloat \(632\)](#), [ReadBinaryStream \(633\)](#)

### 23.4.11 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: [ReadInteger \(630\)](#), [WriteString \(630\)](#), [WriteBool \(631\)](#), [WriteDate \(633\)](#), [WriteDateTime \(633\)](#), [WriteTime \(634\)](#), [WriteFloat \(634\)](#), [WriteBinaryStream \(634\)](#)

### 23.4.12 TCustomIniFile.ReadInt64

Synopsis: Read Int64 value

Declaration: `function ReadInt64(const Section: string; const Ident: string;  
Default: Int64) : Int64; Virtual`

Visibility: public

**Description:** `ReadInt64` reads a signed 64-bit integer value from the ini file. The value is searched in the `Section` section, with key `Ident`.

If the value is not found at the specified `Section`, `Ident` pair, or the value is not a `Int64` value then the `Default` value is returned instead.

This function is needed because `ReadInteger` (625) only reads at most a 32-bit value.

See also: `ReadInteger` (625), `WriteInt64` (625)

### 23.4.13 TCustomIniFile.WriteInt64

**Synopsis:** Write a `Int64` value.

**Declaration:** `procedure WriteInt64(const Section: string; const Ident: string; Value: Int64); Virtual`

**Visibility:** `public`

**Description:** `WriteInt64` writes `Value` as a signed 64-bit integer value to section `Section`, key `Ident`.

See also: `WriteInteger` (625), `ReadInt64` (625)

### 23.4.14 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.

See also: `WriteBool` (631), `ReadInteger` (630), `ReadString` (629), `ReadDate` (632), `ReadDateTime` (632), `ReadTime` (632), `ReadFloat` (632), `ReadBinaryStream` (633)

### 23.4.15 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: `ReadBool` (631), `WriteInteger` (630), `WriteString` (630), `WriteDate` (633), `WriteDateTime` (633), `WriteTime` (634), `WriteFloat` (634), `WriteBinaryStream` (634)



### 23.4.16 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.

See also: `WriteDate` (633), `ReadInteger` (630), `ReadBool` (631), `ReadString` (629), `ReadDateTime` (632), `ReadTime` (632), `ReadFloat` (632), `ReadBinaryStream` (633)

### 23.4.17 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: `WriteDateTime` (633), `ReadInteger` (630), `ReadBool` (631), `ReadDate` (632), `ReadString` (629), `ReadTime` (632), `ReadFloat` (632), `ReadBinaryStream` (633)

### 23.4.18 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: `WriteFloat` (634), `ReadInteger` (630), `ReadBool` (631), `ReadDate` (632), `ReadDateTime` (632), `ReadTime` (632), `ReadString` (629), `ReadBinaryStream` (633)

### 23.4.19 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.

See also: `WriteTime` (634), `ReadInteger` (630), `ReadBool` (631), `ReadDate` (632), `ReadDateTime` (632), `ReadString` (629), `ReadFloat` (632), `ReadBinaryStream` (633)

### 23.4.20 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: `WriteBinaryStream` (634), `ReadInteger` (630), `ReadBool` (631), `ReadDate` (632), `ReadDateTime` (632), `ReadTime` (632), `ReadFloat` (632), `ReadString` (629)

### 23.4.21 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.

See also: `ReadDate` (632), `WriteInteger` (630), `WriteBool` (631), `WriteString` (630), `WriteDateTime` (633), `WriteTime` (634), `WriteFloat` (634), `WriteBinaryStream` (634)

### 23.4.22 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: `ReadDateTime` (632), `WriteInteger` (630), `WriteBool` (631), `WriteDate` (633), `WriteString` (630), `WriteTime` (634), `WriteFloat` (634), `WriteBinaryStream` (634)

### 23.4.23 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: `ReadFloat` (632), `WriteInteger` (630), `WriteBool` (631), `WriteDate` (633), `WriteDateTime` (633), `WriteTime` (634), `WriteString` (630), `WriteBinaryStream` (634)

### 23.4.24 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: `ReadTime` (632), `WriteInteger` (630), `WriteBool` (631), `WriteDate` (633), `WriteDateTime` (633), `WriteString` (630), `WriteFloat` (634), `WriteBinaryStream` (634)

### 23.4.25 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: `ReadBinaryStream` (633), `WriteInteger` (630), `WriteBool` (631), `WriteDate` (633), `WriteDateTime` (633), `WriteTime` (634), `WriteFloat` (634), `WriteString` (630)

### 23.4.26 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` (635) method should be used.

See also: `ReadSections` (635), `SectionExists` (629), `ReadSectionValues` (635)

### 23.4.27 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: `SectionExists` (629), `ReadSectionValues` (635), `ReadSection` (634)

### 23.4.28 TCustomIniFile.ReadSectionValues

Synopsis: Read names and values of a section

**Declaration:** `procedure ReadSectionValues(const Section: string; Strings: TStrings;  
Options: TSectionValuesOptions); Virtual  
; Overload  
procedure ReadSectionValues(const Section: string; Strings: TStrings)  
; Virtual; Overload`

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` (634) can be used.

See also: `SectionExists` (629), `ReadSections` (635), `ReadSection` (634)

### 23.4.29 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: `SectionExists` (629), `ReadSections` (635), `DeleteKey` (636)

### 23.4.30 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: `EraseSection` ([635](#))

### 23.4.31 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: `CacheUpdates` ([644](#))

### 23.4.32 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: `SectionExists` ([629](#))

### 23.4.33 TCustomIniFile.Encoding

Synopsis: Encoding of the ini file.

Declaration: `Property Encoding : TEncoding`

Visibility: public

Access: Read,Write

Description: `Encoding` is the encoding specified in the constructor. It cannot be changed during the lifetime of the instance.

See also: `TCustomIniFile.Create` ([628](#)), `TCustomIniFile.OwnsEncoding` ([639](#))

### 23.4.34 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: [Create \(628\)](#)

### 23.4.35 TCustomIniFile.Options

Synopsis: Options currently in effect

Declaration: `Property Options : TIniFileOptions`

Visibility: public

Access: Read,Write

Description: `Options` is the set of options currently in effect. See [TIniFileOption \(625\)](#) for a list of allowed options. The initial value of this property can be specified using the constructor of the class, `TCustomIniFile.Create (628)`. Not all options can be specified after the ini file object was created.

See also: [TIniFileOption \(625\)](#), [TIniFileOptions \(626\)](#), [TCustomIniFile.Create \(628\)](#)

### 23.4.36 TCustomIniFile.EscapeLineFeeds

Synopsis: Should linefeeds be escaped ?

Declaration: `Property EscapeLineFeeds : Boolean; deprecated;`

Visibility: public

Access: Read

Description: `EscapeLineFeeds` determines whether escaping of linefeeds is enabled: For a description of this feature, see [Create \(628\)](#), as the value of this property must be specified when the `TCustomIniFile` instance is created.

By default, `EscapeLineFeeds` is `False`.

See also: [Create \(628\)](#), [CaseSensitive \(637\)](#)

### 23.4.37 TCustomIniFile.CaseSensitive

Synopsis: Are key and section names case sensitive

Declaration: `Property CaseSensitive : Boolean; deprecated;`

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: `EscapeLineFeeds` ([637](#))

### 23.4.38 TCustomIniFile.StripQuotes

**Synopsis:** Should quotes be stripped from string values

**Declaration:** `Property StripQuotes : Boolean; deprecated;`

**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).

### 23.4.39 TCustomIniFile.FormatSettingsActive

**Synopsis:** Is `FormatSettings` used or not

**Declaration:** `Property FormatSettingsActive : Boolean; deprecated;`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `FormatSettingsActive` can be set to `True` to use the `TCustomIniFile.FormatSettings` (??) field when reading and/or writing values of type date/time or float. If the setting is set to `False` then the defaults specified in the `sysutils` unit are used.

### 23.4.40 TCustomIniFile.BoolTrueStrings

**Synopsis:** Strings to recognize as boolean `True` values.

**Declaration:** `Property BoolTrueStrings : TStringArray`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `BoolTrueStrings` is a list of strings that will be recognized as boolean `True` value in `TCustomIniFile.ReadBool` ([631](#)) The first string in the list will be used when writing a `True` boolean value in `TCustomIniFile.WriteBool` ([631](#)).

The string values are ignored if `ifoWriteStringBoolean` is not in the ini file options ([637](#)).

See also: `TCustomIniFile.SetBoolStringValue` ([629](#)), `TCustomIniFile.BoolFalseStrings` ([639](#)), `TCustomIniFile.ReadBool` ([631](#)), `TCustomIniFile.WriteBool` ([631](#)), `TCustomIniFile.Options` ([637](#))

### 23.4.41 TCustomIniFile.BoolFalseStrings

Synopsis: Strings to recognize as boolean `False` values.

Declaration: `Property BoolFalseStrings : TStringArray`

Visibility: `public`

Access: `Read, Write`

Description: `BoolFalseStrings` is a list of strings that will be recognized as boolean `False` value in `TCustomIniFile.ReadBool` (631) The first string in the list will be used when writing a `False` boolean value in `TCustomIniFile.WriteBool` (631).

The string values are ignored if `ifoWriteStringBoolean` is not in the ini file options (637).

See also: `TCustomIniFile.SetBoolStringValues` (629), `TCustomIniFile.BoolTrueStrings` (638), `TCustomIniFile.ReadBool` (631), `TCustomIniFile.WriteBool` (631), `TCustomIniFile.Options` (637)

### 23.4.42 TCustomIniFile.OwnsEncoding

Synopsis: Does the ini file instance own the encoding ?

Declaration: `Property OwnsEncoding : Boolean`

Visibility: `public`

Access: `Read`

Description: `OwnsEncoding` indicates whether the encoding is owned by the ini file instance or not. If it is owned, it will be freed on destroy. The value of this property is set in the constructor.

See also: `TCustomIniFile.Encoding` (636), `TCustomIniFile.Create` (628)

## 23.5 THashedStringList

### 23.5.1 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` (640) and `IndexOfName` (640) functions make use of these hash values to quickly locate a value.

See also: `IndexOf` (640), `IndexOfName` (640)

### 23.5.2 Method overview

Page	Method	Description
639	<code>Destroy</code>	Clean up instance
640	<code>IndexOf</code>	Returns the index of a string in the list of strings
640	<code>IndexOfName</code>	Return the index of a name in the list of name=value pairs

### 23.5.3 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` (639)

### 23.5.4 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 `#rtl.classes.TStringList.IndexOf` (??) method and uses the hash values to look for the location of `S`.

See also: `#rtl.classes.TStringList.IndexOf` (??), `THashedStringList.IndexOfName` (640)

### 23.5.5 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 `#rtl.classes.TStrings.IndexOfName` (??) method and uses the hash values of the names to look for the location of `Name`.

See also: `#rtl.classes.TStrings.IndexOfName` (??), `THashedStringList.IndexOf` (640)

## 23.6 TIniFile

### 23.6.1 Description

`TIniFile` is an implementation of `TCustomIniFile` (627) which does the same as `TMemIniFile` (650), 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`

See also: `TCustomIniFile` (627), `TMemIniFile` (650)

### 23.6.2 Method overview

Page	Method	Description
<a href="#">641</a>	Create	Create a new instance of <code>TIniFile</code>
<a href="#">643</a>	DeleteKey	Delete key
<a href="#">641</a>	Destroy	Remove the <code>TIniFile</code> instance from memory
<a href="#">643</a>	EraseSection	
<a href="#">642</a>	ReadSection	Read the key names in a section
<a href="#">642</a>	ReadSectionRaw	Read raw section
<a href="#">643</a>	ReadSections	Read section names
<a href="#">643</a>	ReadSectionValues	
<a href="#">642</a>	ReadString	Read a string
<a href="#">644</a>	UpdateFile	Update the file on disk
<a href="#">642</a>	WriteString	Write string to file

### 23.6.3 Property overview

Page	Properties	Access	Description
<a href="#">644</a>	CacheUpdates	rw	Should changes be kept in memory
<a href="#">644</a>	Stream	r	Stream from which ini file was read
<a href="#">645</a>	WriteBOM	rw	

### 23.6.4 TIniFile.Create

Synopsis: Create a new instance of `TIniFile`

Declaration: constructor `Create(const AFileName: string; AOptions: TIniFileOptions)`  
                   ; Override; Overload  
 constructor `Create(AStream: TStream; AOptions: TIniFileOptions)`  
                   ; Overload  
 constructor `Create(AStream: TStream; AEscapeLineFeeds: Boolean)`  
                   ; Overload  
 constructor `Create(AStream: TStream; ADefaultEncoding: TEncoding;`  
                   `AOptions: TIniFileOptions)`  
 constructor `Create(AStream: TStream; ADefaultEncoding: TEncoding;`  
                   `AOwnsEncoding: Boolean; AOptions: TIniFileOptions)`

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`.

Default options for the `TCustomIniFile.Options` ([637](#)) property can be specified in `AOptions`.

See also: `TCustomIniFile` ([627](#)), `TMemIniFile` ([650](#)), `TCustomIniFile.Options` ([637](#))

### 23.6.5 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:** `UpdateFile` ([636](#)), `CacheUpdates` ([644](#))

### 23.6.6 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` ([629](#)) abstract method by looking at the in-memory copy of the ini file and returning the string found there.

**See also:** `TCustomIniFile.ReadString` ([629](#))

### 23.6.7 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` ([630](#)) abstract method by writing the string to the in-memory copy of the ini file. If `CacheUpdates` ([644](#)) property is `False`, then the whole file is immediately written to disk as well.

**Errors:** If an error happens when the file is written to disk, an exception will be raised.

### 23.6.8 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` ([634](#))

**See also:** `TCustomIniFile.ReadSection` ([634](#)), `TIniFile.ReadSectionRaw` ([642](#))

### 23.6.9 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` (642), `TCustomIniFile.ReadSection` (634)

### 23.6.10 TIniFile.ReadSections

**Synopsis:** Read section names

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

**Visibility:** public

**Description:** `ReadSections` is the implementation of `TCustomIniFile.ReadSections` (635). It operates on the in-memory copy of the inifile, and places all section names in `Strings`.

See also: `TIniFile.ReadSection` (642), `TCustomIniFile.ReadSections` (635), `TIniFile.ReadSectionValues` (643)

### 23.6.11 TIniFile.ReadSectionValues

**Synopsis:**

**Declaration:** `procedure ReadSectionValues(const Section: string; Strings: TStrings;  
AOptions: TSectionValuesOptions); Override  
; Overload`

**Visibility:** public

**Description:** `ReadSectionValues` is the implementation of `TCustomIniFile.ReadSectionValues` (635). 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` (642), `TCustomIniFile.ReadSectionValues` (635), `TIniFile.ReadSections` (643)

### 23.6.12 TIniFile.EraseSection

**Synopsis:**

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

**Visibility:** public

**Description:** `EraseSection` deletes the section `Section` from memory, if `CacheUpdates` (644) is `False`, then the file is immediately updated on disk. This method is the implementation of the abstract `TCustomIniFile.EraseSection` (635) method.

See also: `TCustomIniFile.EraseSection` (635), `TIniFile.ReadSection` (642), `TIniFile.ReadSections` (643)

### 23.6.13 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` (644) is `False`, then the file is immediately updated on disk.

See also: `CacheUpdates` (644)

### 23.6.14 `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: `CacheUpdates` (644)

### 23.6.15 `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` (644) method will use this stream to write changes to.

See also: `Create` (641), `UpdateFile` (644)

### 23.6.16 `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` (644). 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 adversely affect performance.

See also: `UpdateFile` (644)

### 23.6.17 TIniFile.WriteBOM

Declaration: `Property WriteBOM : Boolean`

Visibility: `public`

Access: `Read,Write`

## 23.7 TIniFileKey

### 23.7.1 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` (640) class.

See also: `TIniFile` (640)

### 23.7.2 Method overview

Page	Method	Description
645	Create	Create a new instance of <code>TIniFileKey</code>

### 23.7.3 Property overview

Page	Properties	Access	Description
645	Ident	rw	Key name
646	Value	rw	Key value

### 23.7.4 TIniFileKey.Create

Synopsis: Create a new instance of `TIniFileKey`

Declaration: `constructor Create(const AIdent: string; const AValue: string)`

Visibility: `public`

Description: `Create` instantiates a new instance of `TIniFileKey` on the heap. It fills `Ident` (645) with `AIdent` and `Value` (646) with `AValue`.

See also: `Ident` (645), `Value` (646)

### 23.7.5 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: `Value` (646)

### 23.7.6 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: `Ident` ([645](#))

## 23.8 TIniFileKeyList

### 23.8.1 Description

`TIniFileKeyList` maintains a list of `TIniFileKey` ([645](#)) instances on behalf of the `TIniFileSection` ([647](#)) class. It stores they keys of one section of the .ini files.

See also: `TIniFileKey` ([645](#)), `TIniFileSection` ([647](#))

### 23.8.2 Method overview

Page	Method	Description
<a href="#">646</a>	<code>Clear</code>	Clear the list
<a href="#">646</a>	<code>Destroy</code>	Free the instance

### 23.8.3 Property overview

Page	Properties	Access	Description
<a href="#">647</a>	<code>Items</code>	<code>r</code>	Indexed access to <code>TIniFileKey</code> items in the list

### 23.8.4 TIniFileKeyList.Destroy

Synopsis: Free the instance

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` clears up the list using `Clear` ([646](#)) and then calls the inherited `destroy`.

See also: `Clear` ([646](#))

### 23.8.5 TIniFileKeyList.Clear

Synopsis: Clear the list

Declaration: `procedure Clear; Override`

Visibility: `public`

Description: `Clear` removes all `TIniFileKey` ([645](#)) instances from the list, and frees the instances.

See also: `TIniFileKey` ([645](#))

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

See also: TIniFileKey (645)

## 23.9 TIniFileSection

### 23.9.1 Description

TIniFileSection is a class which represents a section in the .ini, and is used internally by the TIniFile (640) class (one instance of TIniFileSection is created for each section in the file by the TIniFileSectionList (649) list). The name of the section is stored in the Name (648) property, and the key/value pairs in this section are available in the KeyList (648) property.

See also: TIniFileKeyList (646), TIniFile (640), TIniFileSectionList (649)

### 23.9.2 Method overview

Page	Method	Description
648	Create	Create a new section object
648	Destroy	Free the section object from memory
647	Empty	Is the section empty

### 23.9.3 Property overview

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

### 23.9.4 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.



### 23.9.5 TIniFileSection.Create

Synopsis: Create a new section object

Declaration: `constructor Create(const AName: string)`

Visibility: `public`

Description: `Create` instantiates a new `TIniFileSection` class, and sets the name to `AName`. It allocates a `TIniFileKeyList` (646) instance to keep all the key/value pairs for this section.

See also: `TIniFileKeyList` (646)

### 23.9.6 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: `Create` (648), `TIniFileKeyList` (646)

### 23.9.7 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` (648)

### 23.9.8 TIniFileSection.KeyList

Synopsis: List of key/value pairs in this section

Declaration: `Property KeyList : TIniFileKeyList`

Visibility: `public`

Access: `Read`

Description: `KeyList` is the `TIniFileKeyList` (646) instance that is used by the `TIniFileSection` to keep the key/value pairs of the section.

See also: `TIniFileSection.Name` (648), `TIniFileKeyList` (646)

## 23.10 TIniFileSectionList

### 23.10.1 Description

TIniFileSectionList maintains a list of TIniFileSection (647) instances, one for each section in an .ini file. TIniFileSectionList is used internally by the TIniFile (640) class to represent the sections in the file.

See also: TIniFileSection (647), TIniFile (640)

### 23.10.2 Method overview

Page	Method	Description
649	Clear	Clear the list
649	Destroy	Free the object from memory

### 23.10.3 Property overview

Page	Properties	Access	Description
649	Items	r	Indexed access to all the section objects in the list

### 23.10.4 TIniFileSectionList.Destroy

Synopsis: Free the object from memory

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy calls Clear (649) to clear the section list and then calls the inherited Destroy

See also: Clear (649)

### 23.10.5 TIniFileSectionList.Clear

Synopsis: Clear the list

Declaration: procedure Clear; Override

Visibility: public

Description: Clear removes all TIniFileSection (647) items from the list, and frees the items it removes from the list.

See also: TIniFileSection (647), TIniFileSectionList.Items (649)

### 23.10.6 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 \(647\)](#), [TIniFileSectionList.Clear \(649\)](#)

## 23.11 TMemIniFile

### 23.11.1 Description

TMemIniFile is a simple descendent of TIniFile (640) 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.

See also: [TIniFile \(640\)](#), [TCustomIniFile \(627\)](#), [CacheUpdates \(644\)](#)

### 23.11.2 Method overview

Page	Method	Description
650	Clear	Clear the data
650	Create	Create a new instance of <code>TMemIniFile</code>
651	GetStrings	Get contents of ini file as stringlist
651	Rename	Rename the ini file
651	SetStrings	Set data from a stringlist

### 23.11.3 TMemIniFile.Create

### Synopsis: Create a new instance of TMemIniFile

```
Declaration: constructor Create(const AFileName: string; AOptions: TIniFileOptions)
                ; Override; Overload
constructor Create(const AFileName: string; AEscapeLineFeeds: Boolean)
                ; Override; Overload
```

Visibility: public

**Description:** `Create` simply calls the inherited `Create` (641), and sets the `CacheUpdates` (644) to `True` so updates will be kept in memory till they are explicitly written to disk.

See also: [TIniFile.Create \(641\)](#), [CacheUpdates \(644\)](#)

### 23.11.4 TMemIniFile.Clear

### Synopsis: Clear the data

Declaration: procedure Clear

Visibility: public

**Description:** Clear removes all sections and key/value pairs from memory. If CacheUpdates (644) is set to `False` then the file on disk will immediately be emptied.

See also: [SetStrings \(651\)](#), [GetStrings \(651\)](#)

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

See also: `SetStrings` (651), `Clear` (650)

### 23.11.6 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: `UpdateFile` (644)

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

See also: `GetStrings` (651), `Clear` (650)

## 23.12 TStringHash

### 23.12.1 Description

`TStringHash` is a Delphi compatibility object. It is not used in the `TIniFile` implementation. It implements a bucket list for `Name=Value` pairs, where `Value` is an integer. This enables quick lookup of values based on a name.

See also: `TIniFile` (640), `TStringHash.Create` (652), `TStringHash.ValueOf` (653)

**23.12.2 Method overview**

Page	Method	Description
<a href="#">652</a>	Add	Add a new value to the hash
<a href="#">653</a>	Clear	Remove all values
<a href="#">652</a>	Create	Create a new instance of TStringHash
<a href="#">652</a>	Destroy	Free TStringHash instance
<a href="#">653</a>	Modify	Try to modify an existing value
<a href="#">653</a>	Remove	Remove a key from the hash
<a href="#">653</a>	ValueOf	Retrieve value of Key

**23.12.3 Property overview**

Page	Properties	Access	Description
<a href="#">654</a>	AddReplacesExisting	rw	Should Add replace existing values or not

**23.12.4 TStringHash.Create**

Synopsis: Create a new instance of TStringHash

Declaration: constructor Create(ACapacity: Cardinal)

Visibility: public

Description: Create instantiates a new instance of TStringHash. The ACapacity argument is present for Delphi compatibility, but is otherwise unused.

See also: TStringHash.ValueOf ([653](#)), TStringHash.Destroy ([652](#))

**23.12.5 TStringHash.Destroy**

Synopsis: Free TStringHash instance

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy clears the internal data structures and removes the TStringHash instance from memory.

See also: TStringHash.Create ([652](#)), TStringHash.Clear ([653](#))

**23.12.6 TStringHash.Add**

Synopsis: Add a new value to the hash

Declaration: procedure Add(const Key: string; Value: Integer)

Visibility: public

Description: Add adds the value Value with name AKey to the list. The behaviour of Add depends on AddReplacesExisting ([654](#)). If it is False, then the existing value is left unchanged, and an exception is raised. If AddReplacesExisting is True then an existing value is replaced.

Errors: An exception will be raised if the value already exists and AddReplacesExisting is False

See also: TStringHash.AddReplacesExisting ([654](#)), TStringHash.Modify ([653](#)), TStringHash.Remove ([653](#)), TStringHash.ValueOf ([653](#))

### 23.12.7 TStringHash.Clear

Synopsis: Remove all values

Declaration: `procedure Clear`

Visibility: `public`

Description: `Clear` removes all values from the hash.

See also: `TStringHash.Destroy` (652), `TStringHash.Add` (652), `TStringHash.Remove` (653), `TStringHash.ValueOf` (653)

### 23.12.8 TStringHash.Modify

Synopsis: Try to modify an existing value

Declaration: `function Modify(const Key: string; Value: Integer) : Boolean`

Visibility: `public`

Description: `Modify` will replace the value of `Key` with `Value`. `Key` must exist. It returns `True` if the operation was successful. If the value didn't exist, `False` is returned.

See also: `TStringHash.Add` (652), `TStringHash.Clear` (653), `TStringHash.Remove` (653), `TStringHash.ValueOf` (653)

### 23.12.9 TStringHash.Remove

Synopsis: Remove a key from the hash

Declaration: `procedure Remove(const Key: string)`

Visibility: `public`

Description: `Remove` removes the key `Key` from the hash, if it was present.

Errors: None.

See also: `TStringHash.Add` (652), `TStringHash.Clear` (653), `TStringHash.Modify` (653), `TStringHash.ValueOf` (653)

### 23.12.10 TStringHash.ValueOf

Synopsis: Retrieve value of `Key`

Declaration: `function ValueOf(const Key: string) : Integer`

Visibility: `public`

Description: `ValueOf` returns the value of `Key`, if it is present. if the key is not present, `-1` is returned.

Errors: None.

See also: `TStringHash.Add` (652), `TStringHash.Clear` (653), `TStringHash.Modify` (653), `TStringHash.Remove` (653)

### 23.12.11 TStringHash.AddReplacesExisting

Synopsis: Should Add replace existing values or not

Declaration: Property `AddReplacesExisting` : Boolean

Visibility: public

Access: Read, Write

Description: `AddReplacesExisting` indicates whether `TStringHash.Add` ([652](#)) will replace an existing value (`True`) or will raise an exception when an existing value is added (`False`).

See also: `TStringHash.Add` ([652](#))

## Chapter 24

# Reference for unit 'iostream'

### 24.1 Used units

Table 24.1: Used units by unit 'iostream'

Name	Page
Classes	??

### 24.2 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`).

### 24.3 Constants, types and variables

#### 24.3.1 Types

`TIOSType = (iosInput, iosOutPut, iosError)`

Table 24.2: 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` (656) constructor of `TIOStream` (656), it determines what kind of stream is created.



## 24.4 EIOStreamError

### 24.4.1 Description

Error thrown in case of an invalid operation on a TIOStream ([656](#)).

## 24.5 TIOStream

### 24.5.1 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 ([655](#)) 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 ([657](#)) behaviour based on this position.

See also: TIOSType ([655](#)), THandleStream (??)

### 24.5.2 Method overview

Page	Method	Description
<a href="#">656</a>	Create	Construct a new instance of TIOStream ( <a href="#">656</a> )
<a href="#">656</a>	Read	Read data from the stream.
<a href="#">657</a>	Seek	Set the stream position
<a href="#">657</a>	Write	Write data to the stream

### 24.5.3 TIOStream.Create

Synopsis: Construct a new instance of TIOStream ([656](#))

Declaration: `constructor Create(aIOSType: TIOSType)`

Visibility: public

Description: Create creates a new instance of TIOStream ([656](#)), 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 ([656](#)), TIOStream.Write ([657](#))

### 24.5.4 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 ([656](#)) 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` (655), `TIOStream.Write` (657)

### 24.5.5 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` (656) exception. If the stream can be written to, it calls the inherited `Write` to actually read the data.

Errors: An `EIOStreamError` exception is raised if the stream does not allow writing.

See also: `TIOSType` (655), `TIOStream.Read` (656)

### 24.5.6 TIOStream.Seek

Synopsis: Set the stream position

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; 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` (656) exception is raised if the stream does not allow the requested seek operation.

See also: `EIOStreamError` (656)

# Chapter 25

## Reference for unit 'libtar'

### 25.1 Used units

Table 25.1: Used units by unit 'libtar'

Name	Page
Classes	??
sysutils	??

### 25.2 Overview

The libtar units provides 2 classes to read and write .tar archives: TTarArchive (662) class can be used to read a tar file, and the TTarWriter (664) class can be used to write a tar file. The unit was implemented originally by Stefan Heymann.

### 25.3 Constants, types and variables

#### 25.3.1 Constants

```
ALL_PERMISSIONS = [tpReadByOwner, tpWriteByOwner, tpExecuteByOwner  
    , tpReadByGroup, tpWriteByGroup, tpExecuteByGroup, tpReadByOther,  
    tpWriteByOther, tpExecuteByOther]
```

ALL\_PERMISSIONS is a set constant containing all possible permissions (read/write/execute, for all groups of users) for an archive entry.

```
EXECUTE_PERMISSIONS = [tpExecuteByOwner, tpExecuteByGroup, tpExecuteByOther  
    ]
```

WRITE\_PERMISSIONS is a set constant containing all possible execute permissions set for an archive entry.

```
FILETYPE_NAME : Array[TFileType] of string = ('Regular', 'Link', 'Symbolic Link'  
    , 'Char File', 'Block File', 'Directory', 'FIFO File', 'Contiguous'  
    , 'Dir Dump', 'Multivol', 'Volume Header')
```

`FILETYPE_NAME` can be used to get a textual description for each of the possible entry file types.

```
READ_PERMISSIONS = [tpReadByOwner, tpReadByGroup, tpReadByOther]
```

`READ_PERMISSIONS` is a set constant containing all possible read permissions set for an archive entry.

```
WRITE_PERMISSIONS = [tpWriteByOwner, tpWriteByGroup, tpWriteByOther
]
```

`WRITE_PERMISSIONS` is a set constant containing all possible write permissions set for an archive entry.

### 25.3.2 Types

```
TFileType = (ftNormal, ftLink, ftSymbolicLink, ftCharacter, ftBlock,
ftDirectory, ftFifo, ftContiguous, ftDumpDir, ftMultiVolume
,
ftVolumeHeader)
```

Table 25.2: Enumeration values for type `TFileType`

Value	Explanation
<code>ftBlock</code>	Block device file
<code>ftCharacter</code>	Character device file
<code>ftContiguous</code>	Contiguous file
<code>ftDirectory</code>	Directory
<code>ftDumpDir</code>	List of files
<code>ftFifo</code>	FIFO file
<code>ftLink</code>	Hard link
<code>ftMultiVolume</code>	Multi-volume file part
<code>ftNormal</code>	Normal file
<code>ftSymbolicLink</code>	Symbolic link
<code>ftVolumeHeader</code>	Volume header, can appear only as first entry in the archive

`TFileType` describes the file type of a file in the archive. It is used in the `FileType` field of the `TTarDirRec` (661) record.

```
TTarMode = (tmSetUid, tmSetGid, tmSaveText)
```

Table 25.3: Enumeration values for type `TTarMode`

Value	Explanation
<code>tmSaveText</code>	Bit \$200 is set
<code>tmSetGid</code>	File has SetGID bit set
<code>tmSetUid</code>	File has SetUID bit set.

`TTarMode` describes extra file modes. It is used in the `Mode` field of the `TTarDirRec` (661) record.

`TTarModes = Set of TTarMode`

`TTarModes` denotes the full set of permission bits for the file in the field `Mode` field of the `TTarDirRec` (661) record.

`TTarPermission = (tpReadByOwner, tpWriteByOwner, tpExecuteByOwner, tpReadByGroup, tpWriteByGroup, tpExecuteByGroup, tpReadByOther, tpWriteByOther, tpExecuteByOther)`

Table 25.4: Enumeration values for type `TTarPermission`

Value	Explanation
<code>tpExecuteByGroup</code>	Group can execute the file
<code>tpExecuteByOther</code>	Other people can execute the file
<code>tpExecuteByOwner</code>	Owner can execute the file
<code>tpReadByGroup</code>	Group can read the file
<code>tpReadByOther</code>	Other people can read the file.
<code>tpReadByOwner</code>	Owner can read the file
<code>tpWriteByGroup</code>	Group can write the file
<code>tpWriteByOther</code>	Other people can write the file
<code>tpWriteByOwner</code>	Owner can write the file

`TTarPermission` denotes part of a files permission as it stored in the `.tar` archive. Each of these enumerated constants correspond with one of the permission bits from a UNIX file permission.

`TTarPermissions = Set of TTarPermission`

`TTarPermissions` describes the complete set of permissions that a file has. It is used in the `Permissions` field of the `TTarDirRec` (661) record.

## 25.4 Procedures and functions

### 25.4.1 ClearDirRec

Synopsis: Initialize tar archive entry

Declaration: `procedure ClearDirRec(var DirRec: TTarDirRec)`

Visibility: default

Description: `ClearDirRec` clears the `DirRec` entry, it basically zeroes out all fields.

See also: `TTarDirRec` (661)

### 25.4.2 ConvertFilename

Synopsis: Convert filename to archive format

Declaration: `function ConvertFilename(Filename: string) : string`

Visibility: default

Description: `ConvertFileName` converts the file name `FileName` to a format allowed by the tar archive. Basically, it converts directory specifiers to forward slashes.

### 25.4.3 FileTimeGMT

Synopsis: Extract filetype

Declaration: `function FileTimeGMT (FileName: string) : TDateTime; Overload`  
`function FileTimeGMT (SearchRec: TSearchRec) : TDateTime; Overload`

Visibility: default

Description: `FileTimeGMT` returns the timestamp of a filename (`FileName` must exist) or a search rec (`TSearchRec`) to a GMT representation that can be used in a tar entry.

See also: `TTarDirRec` ([661](#))

### 25.4.4 PermissionString

Synopsis: Convert a set of permissions to a string

Declaration: `function PermissionString (Permissions: TTarPermissions) : string`

Visibility: default

Description: `PermissionString` can be used to convert a set of `Permissions` to a string in the same format as used by the UNIX `'ls'` command.

See also: `TTarPermissions` ([660](#))

## 25.5 TTarDirRec

```
TTarDirRec = record
  Name : AnsiString;
  Size : Int64;
  DateTime
    : TDateTime;
  Permissions : TTarPermissions;
  FileType : TFileType
;
  LinkName : AnsiString;
  UID : Integer;
  GID : Integer;
  UserName
    : AnsiString;
  GroupName : AnsiString;
  ChecksumOK : Boolean;
  Mode : TTarModes;
  Magic : AnsiString;
  MajorDevNo : Integer;
  MinorDevNo : Integer;
  FilePos : Int64;
end
```

`TTarDirRec` describes an entry in the tar archive. It is similar to a directory entry as in `TSearchRec` ([??](#)), and is returned by the `TTarArchive.FindNext` ([663](#)) call.

## 25.6 TTarArchive

### 25.6.1 Description

`TTarArchive` is the class used to read and examine `.tar` archives. It can be constructed from a stream or from a filename. Creating an instance will not perform any operation on the stream yet.

See also: `TTarWriter` ([664](#)), `FindNext` ([663](#))

### 25.6.2 Method overview

Page	Method	Description
<a href="#">662</a>	<code>Create</code>	Create a new instance of the archive
<a href="#">662</a>	<code>Destroy</code>	Destroy <code>TTarArchive</code> instance
<a href="#">663</a>	<code>FindNext</code>	Find next archive entry
<a href="#">663</a>	<code>GetFilePos</code>	Return current archive position
<a href="#">663</a>	<code>ReadFile</code>	Read a file from the archive
<a href="#">662</a>	<code>Reset</code>	Reset archive
<a href="#">664</a>	<code>SetFilePos</code>	Set position in archive

### 25.6.3 TTarArchive.Create

Synopsis: Create a new instance of the archive

Declaration: `constructor Create(Stream: TStream); Overload`  
`constructor Create(Filename: string; FileMode: Word); Overload`

Visibility: `public`

Description: `Create` can be used to create a new instance of `TTarArchive` using either a `StreamTStream` ([??](#)) descendent or using a name of a file to open: `FileName`. In case of the filename, an open mode can be specified.

Errors: In case a filename is specified and the file cannot be opened, an exception will occur.

See also: `FindNext` ([663](#))

### 25.6.4 TTarArchive.Destroy

Synopsis: Destroy `TTarArchive` instance

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` closes the archive stream (if it created a stream) and cleans up the `TTarArchive` instance.

See also: `TTarArchive.Create` ([662](#))

### 25.6.5 TTarArchive.Reset

Synopsis: Reset archive

Declaration: `procedure Reset`

Visibility: public

Description: `Reset` sets the archive file position on the beginning of the archive.

See also: `TTarArchive.Create` ([662](#))

### 25.6.6 `TTarArchive.FindNext`

Synopsis: Find next archive entry

Declaration: `function FindNext (var DirRec: TTarDirRec) : Boolean`

Visibility: public

Description: `FindNext` positions the file pointer on the next archive entry, and returns all information about the entry in `DirRec`. It returns `True` if the operation was successful, or `False` if not (for instance, when the end of the archive was reached).

Errors: In case there are no more entries, `False` is returned.

See also: `TTarArchive.ReadFile` ([663](#))

### 25.6.7 `TTarArchive.ReadFile`

Synopsis: Read a file from the archive

Declaration: `procedure ReadFile (Buffer: POINTER); Overload`  
`procedure ReadFile (Stream: TStream); Overload`  
`procedure ReadFile (Filename: string); Overload`  
`function ReadFile : string; Overload`

Visibility: public

Description: `ReadFile` can be used to read the current file in the archive. It can be called after the archive was successfully positioned on an entry in the archive. The file can be read in various ways:

- directly in a memory buffer. No checks are performed to see whether the buffer points to enough memory.
- It can be copied to a `Stream`.
- It can be copied to a file with name `FileName`.
- The file content can be copied to a string

Errors: An exception may occur if the buffer is not large enough, or when the file specified in `filename` cannot be opened.

### 25.6.8 `TTarArchive.GetFilesPos`

Synopsis: Return current archive position

Declaration: `procedure GetFilesPos (var Current: Int64; var Size: Int64)`

Visibility: public

Description: `GetFilesPos` returns the position in the tar archive in `Current` and the complete archive size in `Size`.

See also: `TTarArchive.SetFilesPos` ([664](#)), `TTarArchive.Reset` ([662](#))



### 25.6.9 TTarArchive.SetFilePos

Synopsis: Set position in archive

Declaration: `procedure SetFilePos(NewPos: Int64)`

Visibility: public

Description: `SetFilePos` can be used to set the absolute position in the tar archive.

See also: `TTarArchive.Reset` (662), `TTarArchive.GetFilePos` (663)

## 25.7 TTarWriter

### 25.7.1 Description

`TTarWriter` can be used to create `.tar` archives. It can be created using a filename, in which case the archive will be written to the filename, or it can be created using a stream, in which case the archive will be written to the stream - for instance a compression stream.

See also: `TTarArchive` (662)

### 25.7.2 Method overview

Page	Method	Description
<a href="#">666</a>	<code>AddDir</code>	Add directory to archive
<a href="#">665</a>	<code>AddFile</code>	Add a file to the archive
<a href="#">667</a>	<code>AddLink</code>	Add hard link to archive
<a href="#">665</a>	<code>AddStream</code>	Add stream contents to archive.
<a href="#">666</a>	<code>AddString</code>	Add string as file data
<a href="#">666</a>	<code>AddSymbolicLink</code>	Add a symbolic link to the archive
<a href="#">667</a>	<code>AddVolumeHeader</code>	Add volume header entry
<a href="#">664</a>	<code>Create</code>	Create a new archive
<a href="#">665</a>	<code>Destroy</code>	Close archive and clean up <code>TTarWriter</code>
<a href="#">667</a>	<code>Finalize</code>	Finalize the archive

### 25.7.3 Property overview

Page	Properties	Access	Description
<a href="#">668</a>	<code>GID</code>	rw	Archive entry group ID
<a href="#">668</a>	<code>GroupName</code>	rw	Archive entry group name
<a href="#">669</a>	<code>Magic</code>	rw	Archive entry Magic constant
<a href="#">669</a>	<code>Mode</code>	rw	Archive entry mode
<a href="#">667</a>	<code>Permissions</code>	rw	Archive entry permissions
<a href="#">668</a>	<code>UID</code>	rw	Archive entry user ID
<a href="#">668</a>	<code>UserName</code>	rw	Archive entry user name

### 25.7.4 TTarWriter.Create

Synopsis: Create a new archive

Declaration: `constructor Create(TargetStream: TStream); Overload`  
`constructor Create(TargetFilename: string; Mode: Integer); Overload`

Visibility: public

**Description:** `Create` creates a new `TTarWriter` instance. This will start a new `.tar` archive. The archive will be written to the `TargetStream` stream or to a file with name `TargetFileName`, which will be opened with filemode `Mode`.

**Errors:** In case `TargetFileName` cannot be opened, an exception will be raised.

See also: `TTarWriter.Destroy` (665)

### 25.7.5 `TTarWriter.Destroy`

**Synopsis:** Close archive and clean up `TTarWriter`

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

Visibility: public

**Description:** `Destroy` will close the archive (i.e. it writes the end-of-archive marker, if it was not yet written), and then frees the `TTarWriter` instance.

See also: `TTarWriter.Finalize` (667)

### 25.7.6 `TTarWriter.AddFile`

**Synopsis:** Add a file to the archive

**Declaration:** `procedure AddFile(Filename: string; TarFilename: AnsiString)`

Visibility: public

**Description:** `AddFile` adds a file to the archive: the contents is read from `FileName`. Optionally, an alternative filename can be specified in `TarFileName`. This name should contain only forward slash path separators. If it is not specified, the name will be computed from `FileName`.

The archive entry is written with the current owner data and permissions.

**Errors:** If `FileName` cannot be opened, an exception will be raised.

See also: `TTarWriter.AddStream` (665), `TTarWriter.AddString` (666), `TTarWriter.AddLink` (667), `TTarWriter.AddSymbolicLink` (666), `TTarWriter.AddDir` (666), `TTarWriter.AddVolumeHeader` (667)

### 25.7.7 `TTarWriter.AddStream`

**Synopsis:** Add stream contents to archive.

**Declaration:** `procedure AddStream(Stream: TStream; TarFilename: AnsiString;  
FileDateGmt: TDateTime)`

Visibility: public

**Description:** `AddStream` will add the contents of `Stream` to the archive. The `Stream` will not be reset: only the contents of the stream from the current position will be written to the archive. The entry will be written with file name `TarFileName`. This name should contain only forward slash path separators. The entry will be written with timestamp `FileDateGmt`.

The archive entry is written with the current owner data and permissions.

See also: `TTarWriter.AddFile` (665), `TTarWriter.AddString` (666), `TTarWriter.AddLink` (667), `TTarWriter.AddSymbolicLink` (666), `TTarWriter.AddDir` (666), `TTarWriter.AddVolumeHeader` (667)

### 25.7.8 TTarWriter.AddString

Synopsis: Add string as file data

Declaration: `procedure AddString(Contents: AnsiString; TarFilename: AnsiString;  
FileDateGmt: TDateTime)`

Visibility: public

Description: `AddString` adds the string `Contents` as the data of an entry with file name `TarFileName`. This name should contain only forward slash path separators. The entry will be written with timestamp `FileDateGmt`.

The archive entry is written with the current owner data and permissions.

See also: `TTarWriter.AddFile` (665), `TTarWriter.AddStream` (665), `TTarWriter.AddLink` (667), `TTarWriter.AddSymbolicLink` (666), `TTarWriter.AddDir` (666), `TTarWriter.AddVolumeHeader` (667)

### 25.7.9 TTarWriter.AddDir

Synopsis: Add directory to archive

Declaration: `procedure AddDir(Dirname: AnsiString; DateGmt: TDateTime;  
MaxDirSize: Int64)`

Visibility: public

Description: `AddDir` adds a directory entry to the archive. The entry is written with name `DirName`, maximum directory size `MaxDirSize` (0 means unlimited) and timestamp `DateGmt`.

Note that this call only adds an entry for a directory to the archive: if `DirName` is an existing directory, it does not write all files in the directory to the archive.

The directory entry is written with the current owner data and permissions.

See also: `TTarWriter.AddFile` (665), `TTarWriter.AddStream` (665), `TTarWriter.AddLink` (667), `TTarWriter.AddSymbolicLink` (666), `TTarWriter.AddString` (666), `TTarWriter.AddVolumeHeader` (667)

### 25.7.10 TTarWriter.AddSymbolicLink

Synopsis: Add a symbolic link to the archive

Declaration: `procedure AddSymbolicLink(Filename: AnsiString; Linkname: AnsiString;  
DateGmt: TDateTime)`

Visibility: public

Description: `AddSymbolicLink` adds a symbolic link entry to the archive, with name `FileName`, pointing to `LinkName`. The entry is written with timestamp `DateGmt`.

The link entry is written with the current owner data and permissions.

See also: `TTarWriter.AddFile` (665), `TTarWriter.AddStream` (665), `TTarWriter.AddLink` (667), `TTarWriter.AddDir` (666), `TTarWriter.AddString` (666), `TTarWriter.AddVolumeHeader` (667)

### 25.7.11 **TTarWriter.AddLink**

Synopsis: Add hard link to archive

Declaration: `procedure AddLink (Filename: AnsiString; Linkname: AnsiString;  
DateGmt: TDateTime)`

Visibility: public

Description: `AddLink` adds a hard link entry to the archive. The entry has name `FileName`, timestamp `DateGmt` and points to `LinkName`.

The link entry is written with the current owner data and permissions.

See also: `TTarWriter.AddFile` (665), `TTarWriter.AddStream` (665), `TTarWriter.AddSymbolicLink` (666), `TTarWriter.AddDir` (666), `TTarWriter.AddString` (666), `TTarWriter.AddVolumeHeader` (667)

### 25.7.12 **TTarWriter.AddVolumeHeader**

Synopsis: Add volume header entry

Declaration: `procedure AddVolumeHeader (VolumeId: AnsiString; DateGmt: TDateTime)`

Visibility: public

Description: `AddVolumeHeader` adds a volume header entry to the archive. The entry is written with name `VolumeID` and timestamp `DateGmt`.

The volume header entry is written with the current owner data and permissions.

See also: `TTarWriter.AddFile` (665), `TTarWriter.AddStream` (665), `TTarWriter.AddSymbolicLink` (666), `TTarWriter.AddDir` (666), `TTarWriter.AddString` (666), `TTarWriter.AddLink` (667)

### 25.7.13 **TTarWriter.Finalize**

Synopsis: Finalize the archive

Declaration: `procedure Finalize`

Visibility: public

Description: `Finalize` writes the end-of-archive marker to the archive. No more entries can be added after `Finalize` was called.

If the `TTarWriter` instance is destroyed, it will automatically call `finalize` if `finalize` was not yet called.

See also: `TTarWriter.Destroy` (665)

### 25.7.14 **TTarWriter.Permissions**

Synopsis: Archive entry permissions

Declaration: `Property Permissions : TTarPermissions`

Visibility: public

Access: Read, Write

Description: `Permissions` is used for the permissions field of the archive entries.

See also: `TTarDirRec` (661)

### 25.7.15 TTarWriter.UID

Synopsis: Archive entry user ID

Declaration: `Property UID : Integer`

Visibility: `public`

Access: `Read,Write`

Description: `UID` is used for the `UID` field of the archive entries.

See also: `TTarDirRec` ([661](#))

### 25.7.16 TTarWriter.GID

Synopsis: Archive entry group ID

Declaration: `Property GID : Integer`

Visibility: `public`

Access: `Read,Write`

Description: `GID` is used for the `GID` field of the archive entries.

See also: `TTarDirRec` ([661](#))

### 25.7.17 TTarWriter.UserName

Synopsis: Archive entry user name

Declaration: `Property UserName : AnsiString`

Visibility: `public`

Access: `Read,Write`

Description: `UserName` is used for the `UserName` field of the archive entries.

See also: `TTarDirRec` ([661](#))

### 25.7.18 TTarWriter.GroupName

Synopsis: Archive entry group name

Declaration: `Property GroupName : AnsiString`

Visibility: `public`

Access: `Read,Write`

Description: `GroupName` is used for the `GroupName` field of the archive entries.

See also: `TTarDirRec` ([661](#))

### 25.7.19 TTarWriter.Mode

Synopsis: Archive entry mode

Declaration: `Property Mode : TTarModes`

Visibility: `public`

Access: `Read,Write`

Description: `Mode` is used for the `Mode` field of the archive entries.

See also: `TTarDirRec` ([661](#))

### 25.7.20 TTarWriter.Magic

Synopsis: Archive entry Magic constant

Declaration: `Property Magic : AnsiString`

Visibility: `public`

Access: `Read,Write`

Description: `Magic` is used for the `Magic` field of the archive entries.

See also: `TTarDirRec` ([661](#))

## Chapter 26

# Reference for unit 'memds'

### 26.1 Used units

Table 26.1: Used units by unit 'memds'

Name	Page
Classes	??
DB	<a href="#">316</a>
sysutils	??
Types	??

### 26.2 Overview

memds.pp contains classes, types, and routines needed to implement TMemDataset, an in-memory dataset. Ideas implemented in TMemDataset were taken from the THKMemTab component by Harri Kasulke. (Hamburg/Germany)

### 26.3 Constants, types and variables

#### 26.3.1 Constants

```
MarkerSize = SizeOf(Integer)
```

MarkerSize is a constant that indicates the size for markers used in TMemDataset. Markers are read from and written to the internal TMemoryStream for the in-memory dataset, and separates field definitions from the record data in the stream. A marker is also used to indicate the end of the stream.

MarkerSize is defined as the size for the Integer data type.

```
smData = 2
```

smData is an Integer constant that contains the marker used to signify the start of record data for an in-memory dataset. smData is used in TMemDataset methods which read or write record values using the stream for the in-memory dataset. The value for smData is 2.

```
smEOF = 0
```

smEOF is an Integer constant that contains the marker used as the End-of-File marker for an in-memory dataset. smEOF is used in TMemDataset methods which read or write data using a file or a stream. The value for smEOF is 0 (zero).

```
smFieldDefs = 1
```

smFieldDefs is an Integer constant that contains the marker used to signify the start of field definitions for an in-memory dataset. smFieldDefs is used in TMemDataset methods which read or write field definitions for the in-memory dataset. The value for smFieldDefs is 1.

## 26.4 MDSError

### 26.4.1 Description

MDSError is an Exception type raised when an error occurs while reading or writing values for an in-memory dataset. MDSError is raised in the TMemDataset.RaiseError method and uses messages defined in resource strings in the implementation for the unit, including:

- Fieldtype of Field "%s" not supported
- Bookmark %d not found
- Error in data stream at position %d
- Wrong data stream marker at position %d. Got %d, expected %d'
- Filename must not be empty

An MDSError exception will be raised when a field definition uses a data type not supported in TMemDataset. The exception will be raised for the following field types:

- ftADT
- ftCursor
- ftDataSet
- ftDBaseOle
- ftFmtMemo
- ftGraphic
- ftIDispatch
- ftInterface
- ftOraBlob
- ftOraClob
- ftParadoxOle
- ftReference
- ftTimeStamp



- `ftTypedBinary`
- `ftVariant`
- `ftUnknown`

See also: `TMemDataset.RaiseError` ([672](#)), `TMemDataset.FieldDefs` ([681](#)), `TField.FieldType` ([421](#))

## 26.5 TMemDataset

### 26.5.1 Description

`TMemDataset` is a `TDataset` descendant which implements an in-memory dataset. `TMemDataset` is a performant, single user dataset for non-mission critical use cases that do not require transactions. All record and field processing is done in memory; no data is read from or written to disk unless explicitly requested.

`TMemDataset` implements common facilities defined in the `TDataset` ancestor class. This includes using the `FieldDefs` property to define the structure for the dataset. Most (but not all) field types are supported in `TMemDataset`, including:

- `ftString`
- `ftGuid`
- `ftFixedChar`
- `ftBoolean`
- `ftCurrency`
- `ftFloat`
- `ftBCD`
- `ftLargeInt`
- `ftSmallInt`
- `ftWord`
- `ftInteger`
- `ftAutoInc` (behave like `ftInteger` i.e. no auto-increment functionality)
- `ftDateTime`
- `ftDate`
- `ftTime`
- `ftFmtBCD`
- `ftWideString`
- `ftFixedWideChar`
- `ftBytes`
- `ftVarBytes`

- `ftBlob`
- `ftMemo`
- `ftWideMemo`

TMemDataset implements common data manipulation methods such as: Append, AppendRecord, Insert, InsertRecord, Delete, Clear, and Refresh. TMemDataset implements Bookmarks and common navigation methods like: First, Next, Prior, Last, Locate, BOF, and EOF. Methods are provided that allow loading and saving both structure and data from a file, a stream, or another TDataset descendent.

TMemDataset provides methods to filter records, but they are implemented in a different manner than in TDataset. The Filter property is ignored; use the OnFilterRecord method and the Filtered property for this functionality.

One notable missing feature is Indexes. Index definitions are not implemented in TMemDataset.

TMemDataset uses ideas taken from the THKMemTab component by Harri Kasulke. (Hamburg/Germany)

See also: TDataset ([371](#))

## 26.5.2 Method overview

Page	Method	Description
<a href="#">675</a>	BookmarkValid	Determines if the specified Bookmark is valid
<a href="#">677</a>	Clear	Clears the content in the in-memory dataset
<a href="#">675</a>	CompareBookmarks	Gets the relative order for the specified Bookmarks
<a href="#">679</a>	CopyFromDataset	Loads field definitions and optional data from the specified TDataset
<a href="#">674</a>	Create	Constructor for the class instance
<a href="#">675</a>	CreateBlobStream	Creates a stream used to read or write Blob field data in the in-memory dataset
<a href="#">676</a>	CreateTable	Creates the internal storage for records in the in-memory dataset
<a href="#">677</a>	DataSize	Size of the internal TMemoryStream used in the in-memory dataset
<a href="#">674</a>	Destroy	Destructor for the class instance
<a href="#">678</a>	LoadFromFile	Loads the content for the dataset from the specified file name
<a href="#">678</a>	LoadFromStream	Loads the content for the dataset from the specified stream
<a href="#">676</a>	Locate	Locates a record with the specified values in the in-memory dataset
<a href="#">676</a>	Lookup	Searches for a record with the specified values, and returns a list of values
<a href="#">677</a>	SaveToFile	Saves field definitions and optional record data to the specified file name
<a href="#">678</a>	SaveToStream	Saves field definitions and optional record data to the specified stream

### 26.5.3 Property overview

Page	Properties	Access	Description
<a href="#">681</a>	Active		Indicates if the in-memory dataset is Active
<a href="#">683</a>	AfterCancel		
<a href="#">682</a>	AfterClose		
<a href="#">683</a>	AfterDelete		
<a href="#">682</a>	AfterEdit		
<a href="#">682</a>	AfterInsert		
<a href="#">681</a>	AfterOpen		
<a href="#">683</a>	AfterPost		
<a href="#">684</a>	AfterScroll		
<a href="#">683</a>	BeforeCancel		
<a href="#">682</a>	BeforeClose		
<a href="#">683</a>	BeforeDelete		
<a href="#">682</a>	BeforeEdit		
<a href="#">682</a>	BeforeInsert		
<a href="#">681</a>	BeforeOpen		
<a href="#">683</a>	BeforePost		
<a href="#">684</a>	BeforeScroll		
<a href="#">681</a>	FieldDefs		Field definitions for the in-memory dataset
<a href="#">680</a>	FileModified	r	Indicates if the in-memory dataset has been modified
<a href="#">680</a>	FileName	rw	File name used to read or write field definitions and optional data
<a href="#">680</a>	Filter		Filter for the dataset
<a href="#">681</a>	Filtered		Indicates if records in the dataset are filtered using OnFilter-Record
<a href="#">684</a>	OnDeleteError		
<a href="#">684</a>	OnEditError		
<a href="#">685</a>	OnFilterRecord		Event handler signalled to include or exclude records in the in-memory dataset
<a href="#">684</a>	OnNewRecord		
<a href="#">685</a>	OnPostError		

### 26.5.4 TMemDataset.Create

Synopsis: Constructor for the class instance

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: Create is the overridden constructor for the class instance. Create calls the inherited constructor.

Create allocates resources required for internal member variables in the class, such as the TMemoryStream that contains the field definitions and record data and the list used to store Blob data. Other internal member variables are set to their default values. Creates sets the default values for the following published properties:

**BookmarkSize** `SizeOf(LongInt)`

### 26.5.5 TMemDataset.Destroy

Synopsis: Destructor for the class instance

Declaration: `destructor Destroy; Override`

Visibility: public

Description: Destroy is the overridden destructor for the class instance. Destroy frees resources allocated to internal member variables in the class. The list used for Blob data is cleared and freed. Destroy calls the inherited destructor, and frees the internal memory stream for the class.

### 26.5.6 TMemDataset.BookmarkValid

Synopsis: Determines if the specified Bookmark is valid

Declaration: `function BookmarkValid(ABookmark: TBookMark) : Boolean; Override`

Visibility: public

Description: BookmarkValid is an overridden Boolean function used to determine if the specified Bookmark is valid. BookmarkValid implements the virtual method defined in the ancestor class.

In TMemDataset, a Bookmark is considered to be valid when it contains an Integer value that represents a record in the in-memory dataset. Bookmarks are zero-based and must be less than the record count for dataset.

The return value is False when ABookmark is unassigned (contains Nil), or True when the preceding conditions are satisfied.

### 26.5.7 TMemDataset.CompareBookmarks

Synopsis: Gets the relative order for the specified Bookmarks

Declaration: `function CompareBookmarks(Bookmark1: TBookMark; Bookmark2: TBookMark) : LongInt; Override`

Visibility: public

Description: CompareBookmarks is an overridden LongInt function which determines the relative order for the specified Bookmarks.

### 26.5.8 TMemDataset.CreateBlobStream

Synopsis: Creates a stream used to read or write Blob field data in the in-memory dataset

Declaration: `function CreateBlobStream(Field: TField; Mode: TBlobStreamMode) : TStream; Override`

Visibility: public

Description: CreateBlobStream is an overridden TStream function which creates a TMDSBlobStream for the specified field with the read/write permissions in Mode. CreateBlobStream is called when the specified Field needs to read or write its value (for TBlobField or descendent field types).

Mode indicates the permissions need for the blob stream. When Mode contains bmWrite, the value in the State property must indicate that the editing operation is enabled. An exception is raised using DatabaseErrorFmt if State contains a value other than: dsEdit, dsInsert, dsFilter, or dsCalcFields.

Similarly, the Field must allow editing when not in dsSetKey or dsFilter state. An exception is raised using DatabaseErrorFmt if Field has its ReadOnly property set.

The return value contains the TMDSBlobStream instance created using Field and Mode as arguments. Please note that the Blob stream is not saved as part of the data in the in-memory dataset; in its dynamically created and freed as needed.

See also: TFieldDef.DataType ([451](#)), TDataset.State ([398](#)), TField.ReadOnly ([445](#))

### 26.5.9 TMemDataset.Locate

Synopsis: Locates a record with the specified values in the in-memory dataset

Declaration: `function Locate(const KeyFields: string; const KeyValues: Variant;  
Options: TLocateOptions) : Boolean; Override`

Visibility: public

Description: Locate is an overridden Boolean function used to locate a record with the specified values in the specified fields. LocateOptions indicates if case-insensitivity or partial keys searches are used in the method. Locate calls the inherited method to ensure that the dataset is bi-directional. Locate calls CheckActive to ensure that the dataset has been opened prior to searching for values in record data. Locate calls MDSLlocateRecord to get the return value for the method. When the return value is True, the current record for the dataset is updated and Resync is called to update the active record buffer.

See also: TDataset.Locate ([387](#)), TLocateOption ([327](#)), TLocateOptions ([327](#)), TDataset.Resync ([390](#))

### 26.5.10 TMemDataset.Lookup

Synopsis: Searches for a record with the specified values, and returns a list of values

Declaration: `function Lookup(const KeyFields: string; const KeyValues: Variant;  
const ResultFields: string) : Variant; Override`

Visibility: public

Description: Lookup is an overridden Variant function used to search for the first record that matches the specified values. KeyFields is a comma-delimited list of field names to examine in the method. KeyValues is a variant array with values for the specified field names. ResultFields is a comma-delimited list of field names to include in the return values for the method.

Lookup calls MDSLlocateRecord to search for the specified values in the record data for the in-memory dataset. If a record is located that matches the search criteria, calculated or lookup fields in the dataset are recalculated. The return value is a variant array with values for the fields specified in ResultFields. The return value is set to Null if a record with the specified search values is not found.

Please note that Lookup does not change the active record in the dataset.

For example:

```
var AResultVals: Variant;  
    AResultVals := AMemDS.Lookup('lastname, firstname', VarArrayCreate('Franks', 'Pete'))
```

See also: TMemDataset.Locate ([676](#))

### 26.5.11 TMemDataset.CreateTable

Synopsis: Creates the internal storage for records in the in-memory dataset

Declaration: `procedure CreateTable`

Visibility: public

**Description:** CreateTable is used to create the internal storage for records in the in-memory dataset. CreateTable calls CheckInactive to ensure that the dataset is not already opened or Active. CreateTable calls Clear to remove any existing record data in the in-memory dataset. Field definitions are retained. CreateTable calls CalcRecordLayout to determine the record size including Bookmark and BookmarkFlag values. Sets the internal TableIsCreated member to True.

See also: TDataset.Active ([400](#)), TMemDataset.Clear ([677](#)), TDataset.FieldDefs ([395](#)), TMemDataset.CalcRecordLayout ([672](#))

### 26.5.12 TMemDataset.DataSize

**Synopsis:** Size of the internal TMemoryStream used in the in-memory dataset

**Declaration:** `function DataSize : Integer`

**Visibility:** public

**Description:** DataSize is an Integer function used to get the size of the internal stream in the in-memory dataset.

### 26.5.13 TMemDataset.Clear

**Synopsis:** Clears the content in the in-memory dataset

**Declaration:** `procedure Clear(ClearDefs: Boolean)`  
`procedure Clear`

**Visibility:** public

**Description:** Clear is an overloaded procedure used to clear record data, Blob streams, and optionally Field definitions in the in-memory dataset. Clear removes any Blob streams allocated for memo fields in the dataset. Clear removes any memory allocated to the internal TMemoryStream used for record data in the dataset. If the dataset is Active, the Resync method is called to refresh values in the active record buffer.

ClearDefs indicates if the FieldDefs for the dataset are also cleared. When ClearDefs is True, the Close method is called to deactivate the dataset. All field definitions in FieldDefs are removed. The internal member TableIsCreated is set to False.

See also: TDataset.Active ([400](#)), TDataset.Close ([379](#)), TDataset.FieldDefs ([395](#))

### 26.5.14 TMemDataset.SaveToFile

**Synopsis:** Saves field definitions and optional record data to the specified file name

**Declaration:** `procedure SaveToFile(AFileName: string)`  
`procedure SaveToFile(AFileName: string; SaveData: Boolean)`

**Visibility:** public

**Description:** SaveToFile is an overloaded procedure used to store field definitions and optional record data in the dataset to the specified file name. AFileName is the file name on the local file system used to store values from the dataset. SaveData indicates if record data is included in the values stored to the file. When SaveData contains False, only the field definitions for the dataset are stored in the file.

AFileName must contain a file name for the local file system. SaveToFile calls RaiseError to raise an exception if the value in AFileName is an empty string ("). SaveToFile creates a TFileStream for the specified file name, and calls SaveToStream to store the content from the dataset.

SaveToFile reimplements the method defined in the ancestor class.

### 26.5.15 TMemDataset.SaveToStream

**Synopsis:** Saves field definitions and optional record data to the specified stream

**Declaration:** `procedure SaveToStream(F: TStream)`  
`procedure SaveToStream(F: TStream; SaveData: Boolean)`

**Visibility:** public

**Description:** `SaveToStream` is used to save field definitions and optional record data for the in-memory dataset to the specified stream. `SaveToStream` calls `SaveFieldDefsToStream` to save the field definitions in `FieldDefs` to the stream specified in `F`.

`SaveData` indicates if record data is included in the values written to the stream. When `SaveData` contains `True`, the `SaveDataToStream` method is called to save record data to the stream. No record data is written when `SaveData` is `False`. `SaveToStream` calls `WriteMarker` to write the `smEOF` marker value that signifies the end of record data in the stream.

Use `LoadFromStream` to load field definitions and record data for the in-memory dataset.

See also: `TMemDataset.LoadFromStream` ([678](#))

### 26.5.16 TMemDataset.LoadFromStream

**Synopsis:** Loads the content for the dataset from the specified stream

**Declaration:** `procedure LoadFromStream(F: TStream)`

**Visibility:** public

**Description:** `LoadFromStream` is used to load the content for the dataset from the specified stream. `F` is a `TStream` descendent that is used to load the field definitions and record data for the in-memory dataset. `LoadFromStream` calls `Close` to ensure that the dataset saves its existing content (when `FileName` has been assigned) and clears any default `Fields` created when the dataset was opened.

`LoadFromStream` calls `ReadFieldDefsFromStream` to load field definitions from the stream in `F`. `CreateTable` is called to initialize storage for record data in the dataset. `LoadDataFromStream` is called to load any record data present in the stream. `CheckMarker` is called to ensure that the stream is positioned on the `smEOF` marker that signals the end of record data in the stream. An exception is raised if the stream was truncated or does not contain the value `smEOF` at the current position in the stream. `LoadFromStream` sets the value in the `FileModified` property to `False`.

Use `SaveToStream` to write the field definitions and record data in the dataset to a stream.

See also: `TMemDataset.SaveToStream` ([678](#))

### 26.5.17 TMemDataset.LoadFromFile

**Synopsis:** Loads the content for the dataset from the specified file name

**Declaration:** `procedure LoadFromFile(AFileName: string)`

**Visibility:** public

**Description:** `LoadFromFile` is used to load the content for the dataset from the specified file name. `LoadFromFile` creates a `TFileStream` for the file name specified in `AFileName`. The file stream is passed to `LoadFromStream` to load the contents of the file into the in-memory dataset. The file stream is freed prior to exiting from the method.

Use `SaveToFile` to save the contents of an in-memory dataset to a file on the local file system.

See also: `TMemDataset.LoadFromStream` ([678](#)), `TMemDataset.SaveToFile` ([677](#))

### 26.5.18 TMemDataset.CopyFromDataset

**Synopsis:** Loads field definitions and optional data from the specified TDataset

**Declaration:** `procedure CopyFromDataset (DataSet: TDataSet)`  
`procedure CopyFromDataset (DataSet: TDataSet; CopyData: Boolean)`

**Visibility:** public

**Description:** CopyFromDataset is used to load field definitions and optional record data from the specified TDataset descendent. Dataset contains the TDataset used as the source for the structure and optional record data loaded in the method. CopyData indicates if record data is loaded in the method. When CopyData contains False, only the structure from Dataset is loaded in method.

CopyFromDataset removes any existing field definitions in FieldDefs, and any record data stored in the in-memory dataset. CopyFromDataset uses the Fields in the DataSet argument to determine the new structure for the in-memory dataset. This is done because the visible Fields in the dataset may differ from the actual field definitions. CopyFromDataset creates and adds a TFieldDef instance to FieldsDefs for each of the Fields in DataSet.

CopyFromDataset calls CreateTable to allocated record storage for the new field definitions in FieldDefs.

When CopyData contains True, record data from the DataSet argument is added to the in-memory dataset. When CopyData contains False, record data in the DataSet argument is ignored.

The Open method is called to activate both datasets. DisableControls is called for both TDatasets to prevent updates during record navigation. All records in DataSet are loaded into the in-memory dataset by calling Append and setting the value for each of the field definitions in the target. Field definitions with the following data types are loaded using the native type for the field:

- ftFixedChar
- ftString
- ftBoolean
- ftFloat
- ftLargeInt
- ftSmallInt
- ftInteger
- ftDate
- ftTime
- ftDateTime

All other field values are loaded using their AsString representation.

CopyFromDataset calls Post after adding each record in the dataset. If an exception occurs, the Cancel method is called and the exception is re-raised.

CopyFromDataset calls the EnableControls method in both datasets when record data has been loaded in the method. Please note that the record position in the DataSet argument is restored after loading record data.

See also: TDataset.Fields ([398](#)), TDataset.FieldDefs ([395](#)), TMemDataset.CreateTable ([676](#)), TDataset.Append ([377](#)), TDataset.Post ([389](#))



### 26.5.19 TMemDataset.FileModified

Synopsis: Indicates if the in-memory dataset has been modified

Declaration: `Property FileModified : Boolean`

Visibility: public

Access: Read

Description: FileModified is a read-only Boolean property which indicates if the in-memory dataset has been modified. The value in FileModified is updated in methods that write record buffers to the internal memory stream for the dataset, such as:

- MDSWriteRecord
- MDSAppendRecord
- InternalDelete

The value in FileModified is also updated in methods called when opening or closing the in-memory dataset, such as:

- LoadFromStream
- SaveDataToStream
- InternalClose

### 26.5.20 TMemDataset.Filter

Synopsis: Filter for the dataset

Declaration: `Property Filter : ; unimplemented;`

Visibility: public

Access:

Description: **Remark** Filter is not implemented in TMemDataset. Values assigned to the Filter property are silently discarded. Use OnFilterRecord and Filtered instead.

See also: TMemDataset.Filtered ([681](#)), TMemDataset.OnFilterRecord ([685](#))

### 26.5.21 TMemDataset.FileName

Synopsis: File name used to read or write field definitions and optional data

Declaration: `Property FileName : string`

Visibility: published

Access: Read,Write

Description: FileName is a String property that specifies the file used to load field definitions and optional data when the dataset is opened. When FileName is assigned, and the dataset has been modified, it indicates the file name used to store field definitions and data in the local file system.

See also: (??)

### 26.5.22 TMemDataset.Filtered

Synopsis: Indicates if records in the dataset are filtered using OnFilterRecord

Declaration: `Property Filtered :`

Visibility: published

Access:

Description: Filtered is a published Boolean property that indicates if records in the dataset are filtered using the OnFilterRecord event handler. Filtered is used methods that retrieve record buffers or perform record searches, and determines if records are visible in the dataset.

When Filtered contains True, the MDSFilterRecord method is called to perform filtering for records in the dataset. Unlike the ancestor class, the Filter property is not used in TMemDataset. Values assigned to the Filter property are silently discarded. Use the OnFilterRecord event handler to implement comparisons need to determine record visibility.

See also: TMemDataset.OnFilterRecord ([685](#)), TDataset.Filter ([399](#))

### 26.5.23 TMemDataset.Active

Synopsis: Indicates if the in-memory dataset is Active

Declaration: `Property Active :`

Visibility: published

Access:

### 26.5.24 TMemDataset.FieldDefs

Synopsis: Field definitions for the in-memory dataset

Declaration: `Property FieldDefs :`

Visibility: published

Access:

### 26.5.25 TMemDataset.BeforeOpen

Synopsis:

Declaration: `Property BeforeOpen :`

Visibility: published

Access:

### 26.5.26 TMemDataset.AfterOpen

Synopsis:

Declaration: `Property AfterOpen :`

Visibility: published

Access:

### **26.5.27 TMemDataset.BeforeClose**

Synopsis:

Declaration: `Property BeforeClose :`

Visibility: published

Access:

### **26.5.28 TMemDataset.AfterClose**

Synopsis:

Declaration: `Property AfterClose :`

Visibility: published

Access:

### **26.5.29 TMemDataset.BeforeInsert**

Synopsis:

Declaration: `Property BeforeInsert :`

Visibility: published

Access:

### **26.5.30 TMemDataset.AfterInsert**

Synopsis:

Declaration: `Property AfterInsert :`

Visibility: published

Access:

### **26.5.31 TMemDataset.BeforeEdit**

Synopsis:

Declaration: `Property BeforeEdit :`

Visibility: published

Access:

### **26.5.32 TMemDataset.AfterEdit**

Synopsis:

Declaration: `Property AfterEdit :`

Visibility: published

Access:

### **26.5.33 TMemDataset.BeforePost**

Synopsis:

Declaration: `Property BeforePost` :

Visibility: published

Access:

### **26.5.34 TMemDataset.AfterPost**

Synopsis:

Declaration: `Property AfterPost` :

Visibility: published

Access:

### **26.5.35 TMemDataset.BeforeCancel**

Synopsis:

Declaration: `Property BeforeCancel` :

Visibility: published

Access:

### **26.5.36 TMemDataset.AfterCancel**

Synopsis:

Declaration: `Property AfterCancel` :

Visibility: published

Access:

### **26.5.37 TMemDataset.BeforeDelete**

Synopsis:

Declaration: `Property BeforeDelete` :

Visibility: published

Access:

### **26.5.38 TMemDataset.AfterDelete**

Synopsis:

Declaration: `Property AfterDelete` :

Visibility: published

Access:

### **26.5.39 TMemDataset.BeforeScroll**

Synopsis:

Declaration: `Property BeforeScroll :`

Visibility: published

Access:

### **26.5.40 TMemDataset.AfterScroll**

Synopsis:

Declaration: `Property AfterScroll :`

Visibility: published

Access:

### **26.5.41 TMemDataset.OnDeleteError**

Synopsis:

Declaration: `Property OnDeleteError :`

Visibility: published

Access:

### **26.5.42 TMemDataset.OnEditError**

Synopsis:

Declaration: `Property OnEditError :`

Visibility: published

Access:

Description:

See also: (??)

### **26.5.43 TMemDataset.OnNewRecord**

Synopsis:

Declaration: `Property OnNewRecord :`

Visibility: published

Access:

### 26.5.44 TMemDataset.OnPostError

Synopsis:

Declaration: Property OnPostError :

Visibility: published

Access:

### 26.5.45 TMemDataset.OnFilterRecord

Synopsis: Event handler signalled to include or exclude records in the in-memory dataset

Declaration: Property OnFilterRecord :

Visibility: published

Access:

Description: OnFilterRecord is a published TFilterRecordEvent property which provides the event handler signalled to include or exclude records in the in-memory dataset. OnFilterRecord provides a way for the application to decide whether a record is visible in the dataset on a record-by-record basis. Applications must assign a procedure to the event handler that performs any comparison needed to determine record visibility. The procedure must set the value in Accept to True to make the record visible in the in-memory dataset.

Set the Filtered property to True to enable the OnFilterRecord event handler during record navigation.

OnFilterRecord is used as an alternative filtering mechanism; TMemDataset does not implement the Filter property. Values assigned to the Filter property are silently discarded.

See also: TDataSet.OnFilterRecord ([407](#)), TFilterRecordEvent ([326](#)), TDataSet.Filtered ([399](#)), TDataSet.Resync ([390](#))

## Chapter 27

# Reference for unit 'MSSQLConn'

### 27.1 Used units

Table 27.1: Used units by unit 'MSSQLConn'

Name	Page
BufDataset	<a href="#">130</a>
Classes	??
DB	<a href="#">316</a>
dblib	??
SQLDB	<a href="#">769</a>
sysutils	??

### 27.2 Overview

Connector to Microsoft SQL Server databases. Needs FreeTDS dblib library.

### 27.3 Constants, types and variables

#### 27.3.1 Variables

`DBLibLibraryName` : `string` = `DBLIBDLL`

### 27.4 EMSSQLDatabaseError

#### 27.4.1 Description

Sybase/MS SQL Server specific error

#### 27.4.2 Property overview

Page	Properties	Access	Description
<a href="#">687</a>	DBErrorCode	r	Sybase/MS SQL Server error code

### 27.4.3 EMSSQLDatabaseError.DBErrorCode

Synopsis: Sybase/MS SQL Server error code

Declaration: `Property DBErrorCode : Integer; deprecated;`

Visibility: public

Access: Read

Description: Error code as generated by the database server.

## 27.5 TMSSQLConnection

### 27.5.1 Description

Connector to Microsoft SQL Server databases.

Requirements:

MS SQL Server Client Library is required (ntwdblib.dll)

- or -

FreeTDS (dblib.dll)

Older FreeTDS libraries may require freetds.conf: (<http://www.freetds.org/userguide/freetdsconf.htm>)

[global]

tds version = 7.1

client charset = UTF-8

port = 1433 or instance = ... (optional)

dump file = freetds.log (optional)

text size = 2147483647 (optional)

Known problems:

- CHAR/VARCHAR data truncated to column length when encoding to UTF-8 (use NCHAR/NVARCHAR instead or CAST char/varchar to nchar/nvarchar)

- Multiple result sets (MARS) are not supported (for example when SP returns more than 1 result set only 1st is processed)

- DB-Library error 10038 "Results Pending": set `TSQLQuery.PacketRecords=-1` to fetch all pending rows

- BLOB data (IMAGE/TEXT columns) larger than 16MB are truncated to 16MB: (set `TMSSQL-Connection.Params: 'TEXTSIZE=2147483647'` or execute `'SET TEXTSIZE 2147483647'`)

### 27.5.2 Method overview

Page	Method	Description
<a href="#">688</a>	Create	
<a href="#">688</a>	CreateDB	
<a href="#">688</a>	DropDB	
<a href="#">688</a>	GetConnectionInfo	



### 27.5.3 Property overview

Page	Properties	Access	Description
<a href="#">689</a>	CharSet		
<a href="#">689</a>	Connected		
<a href="#">690</a>	DatabaseName		
<a href="#">689</a>	HostName		Host and optionally port or instance
<a href="#">690</a>	KeepConnection		
<a href="#">690</a>	LoginPrompt		
<a href="#">690</a>	OnLogin		
<a href="#">690</a>	Params		
<a href="#">688</a>	Password		
<a href="#">690</a>	Role		
<a href="#">689</a>	Transaction		
<a href="#">689</a>	UserName		

### 27.5.4 TMSSQLConnection.Create

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

### 27.5.5 TMSSQLConnection.GetConnectionInfo

Declaration: `function GetConnectionInfo(InfoType: TConnInfoType) : string; Override`

Visibility: `public`

### 27.5.6 TMSSQLConnection.CreateDB

Declaration: `procedure CreateDB; Override`

Visibility: `public`

### 27.5.7 TMSSQLConnection.DropDB

Declaration: `procedure DropDB; Override`

Visibility: `public`

### 27.5.8 TMSSQLConnection.Password

Declaration: `Property Password :`

Visibility: `published`

Access:

Description: `TMSSQLConnection` specific: if you don't enter a `UserName` and `Password`, the connector will try to use Trusted Authentication/SSPI (on Windows only).

### 27.5.9 TMSSQLConnection.Transaction

Declaration: Property Transaction :

Visibility: published

Access:

### 27.5.10 TMSSQLConnection.UserName

Declaration: Property UserName :

Visibility: published

Access:

Description: TMSSQLConnection specific: if you don't enter a UserName and Password, the connector will try to use Trusted Authentication/SSPI (on Windows only).

### 27.5.11 TMSSQLConnection.CharSet

Declaration: Property CharSet :

Visibility: published

Access:

Description: Character Set - if you use Microsoft DB-Lib and set to 'UTF-8' then char/varchar fields will be UTF8Encoded/Decoded.

If you use FreeTDS DB-Lib, then you must compile with iconv support (requires libiconv2.dll) or cast char/varchar to nchar/nvarchar in SELECTs.

### 27.5.12 TMSSQLConnection.HostName

Synopsis: Host and optionally port or instance

Declaration: Property HostName :

Visibility: published

Access:

Description: TMSSQLConnection specific: you can specify an instance or a port after the host name itself.

Instance should be specified with a backslash e.g.: 127.0.0.1\SQLEXPRESS. Port should be specified with a colon, e.g. BIGBADSERVER:1433

See <http://www.freetds.org/userguide/portoverride.htm>

### 27.5.13 TMSSQLConnection.Connected

Declaration: Property Connected :

Visibility: published

Access:

#### 27.5.14 TMSSQLConnection.Role

Declaration: Property Role :

Visibility: published

Access:

#### 27.5.15 TMSSQLConnection.DatabaseName

Declaration: Property DatabaseName :

Visibility: published

Access:

Description: TMSSQLConnection specific: the master database should always exist on a server.

#### 27.5.16 TMSSQLConnection.KeepConnection

Declaration: Property KeepConnection :

Visibility: published

Access:

#### 27.5.17 TMSSQLConnection.LoginPrompt

Declaration: Property LoginPrompt :

Visibility: published

Access:

#### 27.5.18 TMSSQLConnection.Params

Declaration: Property Params :

Visibility: published

Access:

Description: TMSSQLConnection specific:

set "AutoCommit=true" if you don't want to explicitly commit/rollback transactions

set "TextSize=16777216" - to set maximum size of blob/text/image data returned. Otherwise, these large fields may be cut off when retrieving/setting data.

#### 27.5.19 TMSSQLConnection.OnLogin

Declaration: Property OnLogin :

Visibility: published

Access:

## 27.6 TMSSQLConnectionDef

### 27.6.1 Method overview

Page	Method	Description
<a href="#">691</a>	ConnectionClass	
<a href="#">691</a>	DefaultLibraryName	
<a href="#">691</a>	Description	
<a href="#">691</a>	LoadedLibraryName	
<a href="#">691</a>	LoadFunction	
<a href="#">691</a>	TypeName	
<a href="#">691</a>	UnLoadFunction	

### 27.6.2 TMSSQLConnectionDef.TypeName

Declaration: `class function TypeName : string; Override`

Visibility: default

### 27.6.3 TMSSQLConnectionDef.ConnectionClass

Declaration: `class function ConnectionClass : TSQLConnectionClass; Override`

Visibility: default

### 27.6.4 TMSSQLConnectionDef.Description

Declaration: `class function Description : string; Override`

Visibility: default

### 27.6.5 TMSSQLConnectionDef.DefaultLibraryName

Declaration: `class function DefaultLibraryName : string; Override`

Visibility: default

### 27.6.6 TMSSQLConnectionDef.LoadFunction

Declaration: `class function LoadFunction : TLibraryLoadFunction; Override`

Visibility: default

### 27.6.7 TMSSQLConnectionDef.UnLoadFunction

Declaration: `class function UnLoadFunction : TLibraryUnLoadFunction; Override`

Visibility: default

### 27.6.8 TMSSQLConnectionDef.LoadedLibraryName

Declaration: `class function LoadedLibraryName : string; Override`

Visibility: default

## 27.7 TSybaseConnection

### 27.7.1 Description

Connector to Sybase Adaptive Server Enterprise (ASE) database servers.

Requirements:

FreeTDS (dblib.dll)

Older FreeTDS libraries may require freetds.conf: (<http://www.freetds.org/userguide/freetdsconf.htm>)

[global]

tds version = 7.1

client charset = UTF-8

port = 5000 (optional)

dump file = freetds.log (optional)

text size = 2147483647 (optional)

### 27.7.2 Method overview

Page	Method	Description
<a href="#">692</a>	Create	

### 27.7.3 TSybaseConnection.Create

Declaration: constructor Create(AOwner: TComponent); Override

Visibility: public

## 27.8 TSybaseConnectionDef

### 27.8.1 Method overview

Page	Method	Description
<a href="#">692</a>	ConnectionClass	
<a href="#">693</a>	Description	
<a href="#">692</a>	TypeName	

### 27.8.2 TSybaseConnectionDef.TypeName

Declaration: class function TypeName : string; Override

Visibility: default

### 27.8.3 TSybaseConnectionDef.ConnectionClass

Declaration: class function ConnectionClass : TSQLConnectionClass; Override

Visibility: default

#### 27.8.4 TSybaseConnectionDef.Description

Declaration: `class function Description : string; Override`

Visibility: `default`

# Chapter 28

## Reference for unit 'nullstream'

### 28.1 Used units

Table 28.1: Used units by unit 'nullstream'

Name	Page
Classes	??

### 28.2 Overview

The `nullstream` unit implements `TNullStream` (694), a stream which acts more or less as the `/dev/null` device on unix: all read and write operations will succeed, but the data is discarded on write, or null bytes are read.

### 28.3 ENullStreamError

#### 28.3.1 Description

`ENullStreamError` is the exception raised when `TNullStream.Seek` (695) results in an invalid position.

See also: `TNullStream.Seek` (695)

### 28.4 TNullStream

#### 28.4.1 Description

`TNullStream` discards any data written to it (but keeps a virtual size) and returns 0 bytes when read from. It emulates a `#rtl.Classes.TMemoryStream` (??): When writing to the stream, the size is increased as needed. When reading, the maximum number of returned bytes is limited to the size of the stream.

See also: `TNullStream.Read` (695), `TNullStream.Write` (695), `#rtl.Classes.TStream.Size` (??)

### 28.4.2 Method overview

Page	Method	Description
<a href="#">696</a>	Create	Create a new instance
<a href="#">695</a>	Read	Read null bytes from the stream
<a href="#">695</a>	Seek	Set current position in the stream
<a href="#">695</a>	Write	Write to stream

### 28.4.3 TNullStream.Read

Synopsis: Read null bytes from the stream

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

Visibility: public

Description: `Read` reads `Count` null bytes from the stream. `Count` can be at most `Size`. The `Buffer` will be filled with null bytes, effectively zeroing out the memory. The size can be increased using `Write` or by explicitly setting `Size`.

See also: `TNullStream.Write` ([695](#)), `#rtl.Classes.TStream.Size` (??)

### 28.4.4 TNullStream.Write

Synopsis: Write to stream

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

Visibility: public

Description: `Write` simulates a write operation: no data is actually written from `Buffer`, but the size of the stream is enlarged if the amount of bytes `Count` and current position in the stream make this necessary.

See also: `TNullStream.Read` ([695](#)), `#rtl.Classes.TStream.Size` (??)

### 28.4.5 TNullStream.Seek

Synopsis: Set current position in the stream

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; Override`

Visibility: public

Description: `Seek` sets the current position in the stream. It simulates this operation by keeping a "virtual" position. See `#rtl.Classes.TStream.Seek` (??) for more info about the arguments.

Errors: If the requested operation would cause the position to fall outside of the allowed range (0 to `Size`) then a `ENullStreamError` ([694](#)) exception is raised.

See also: `TNullStream.Read` ([695](#)), `TNullStream.Write` ([695](#)), `#rtl.Classes.TStream.Seek` (??)



### 28.4.6 TNullStream.Create

Synopsis: Create a new instance

Declaration: `constructor Create`

Visibility: `public`

Description: `Create` initializes the size and position of the stream to zero.

See also: `#rtl.Classes.TStream.Position (??)`, `#rtl.Classes.TStream.Size (??)`

## Chapter 29

# Reference for unit 'Pipes'

### 29.1 Used units

Table 29.1: Used units by unit 'Pipes'

Name	Page
Classes	??
sysutils	??

### 29.2 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.

### 29.3 Constants, types and variables

#### 29.3.1 Constants

`ENoSeekMsg = 'Cannot seek on pipes'`

Constant used in `EPipeSeek` (698) exception.

`EPipeMsg = 'Failed to create pipe.'`

Constant used in `EPipeCreation` (698) exception.

### 29.4 Procedures and functions

#### 29.4.1 CreatePipeHandles

Synopsis: Function to create a set of pipe handles

Declaration: `function CreatePipeHandles(var Inhandle: THandle;`

```
var OutHandle: THandle;
APipeBufferSize: Cardinal) : 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` (698)

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

On some systems (notably: windows) the size of the buffer to be used for communication between 2 ends of the buffer can be specified in the `APipeBufferSize` (697) parameter. This parameter is ignored on systems that do not support setting the buffer size.

**Errors:** If no pipe handles could be created, an `EPipeCreation` (698) exception is raised.

See also: `CreatePipeHandles` (697), `TInputPipeStream` (699), `TOutputPipeStream` (701)

## 29.5 EPipeCreation

### 29.5.1 Description

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

## 29.6 EPipeError

### 29.6.1 Description

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

## 29.7 EPipeSeek

### 29.7.1 Description

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

## 29.8 TInputPipeStream

### 29.8.1 Description

`TInputPipeStream` is created by the `CreatePipeStreams` (698) 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.

See also: `TStream` (??), `CreatePipeStreams` (698), `TOutputPipeStream` (701)

### 29.8.2 Method overview

Page	Method	Description
<a href="#">699</a>	<code>Destroy</code>	Destroy this instance of the input pipe stream
<a href="#">700</a>	<code>Read</code>	Read data from the stream to a buffer.
<a href="#">700</a>	<code>Seek</code>	Set the current position of the stream
<a href="#">699</a>	<code>Write</code>	Write data to the stream.

### 29.8.3 Property overview

Page	Properties	Access	Description
<a href="#">700</a>	<code>NumBytesAvailable</code>	<code>r</code>	Number of bytes available for reading.

### 29.8.4 TInputPipeStream.Destroy

Synopsis: Destroy this instance of the input pipe stream

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` overrides the destructor to close the pipe handle, prior to calling the inherited destructor.

Errors: None

See also: `TInputPipeStream.Create` (699)

### 29.8.5 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 `EStreamError` (??) exception is raised when this function is called.

See also: `Read` (700), `Seek` (700)

### 29.8.6 TInputPipeStream.Seek

Synopsis: Set the current position of the stream

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; 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` (698) exception is raised if the stream does not allow the requested seek operation.

See also: `EPipeSeek` (698), `Seek` (??)

### 29.8.7 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: `Write` (699), `Seek` (700)

### 29.8.8 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` (700)

## 29.9 TOutputPipeStream

### 29.9.1 Description

TOutputPipeStream is created by the CreatePipeStreams (698) call to represent the writing end of a pipe. It is a TStream (??) descendent which does not allow reading.

See also: TStream (??), CreatePipeStreams (698), TInputPipeStream (699)

### 29.9.2 Method overview

Page	Method	Description
<a href="#">701</a>	Destroy	Destroy this instance of the output pipe stream
<a href="#">701</a>	Read	Read data from the stream.
<a href="#">701</a>	Seek	Sets the position in the stream

### 29.9.3 TOutputPipeStream.Destroy

Synopsis: Destroy this instance of the output pipe stream

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: Destroy overrides the destructor to close the pipe handle, prior to calling the inherited destructor.

Errors: None

See also: TOutputPipeStream.Create (701)

### 29.9.4 TOutputPipeStream.Seek

Synopsis: Sets the position in the stream

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; 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 (698) exception is raised if this method is called.

### 29.9.5 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 EStreamError (??) exception is raised when this function is called.

See also: Seek (701)

## Chapter 30

# Reference for unit 'pooledmm'

### 30.1 Used units

Table 30.1: Used units by unit 'pooledmm'

Name	Page
Classes	??

### 30.2 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.

### 30.3 Constants, types and variables

#### 30.3.1 Types

`PPooledMemManagerItem` = `^TPooledMemManagerItem`

`PPooledMemManagerItem` is a pointer type, pointing to a `TPooledMemManagerItem` (703) item, used in a linked list.

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

`TEnumItemsMethod` is a prototype for the callback used in the `TNonFreePooledMemManager.EnumerateItems` (704) call. The parameter `Item` will be set to each of the pointers in the item list of `TNonFreePooledMemManager` (703).

#### 30.4 TPooledMemManagerItem

`TPooledMemManagerItem` = `record`

```

    Next : PPooledMemManagerItem;
end

```

`TPooledMemManagerItem` is used internally by the `TPooledMemManager` (705) class to maintain the free list block. It simply points to the next free block.

## 30.5 TNonFreePooledMemManager

### 30.5.1 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.

### 30.5.2 Method overview

Page	Method	Description
703	<code>Clear</code>	Clears the memory
703	<code>Create</code>	Creates a new instance of <code>TNonFreePooledMemManager</code>
704	<code>Destroy</code>	Removes the <code>TNonFreePooledMemManager</code> instance from memory
704	<code>EnumerateItems</code>	Enumerate all items in the list
704	<code>NewItem</code>	Return a pointer to a new memory block

### 30.5.3 Property overview

Page	Properties	Access	Description
704	<code>ItemSize</code>	<code>r</code>	Size of an item in the list

### 30.5.4 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` (704) is valid after a call to `Clear`

See also: `NewItem` (704)

### 30.5.5 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` (704)

### 30.5.6 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` (703), `TNonFreePooledMemManager.Clear` (703)

### 30.5.7 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` (704). 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` (703)

### 30.5.8 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)

### 30.5.9 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` (703)

## 30.6 TPooledMemManager

### 30.6.1 Description

`TPooledMemManager` is a class which maintains a linked list of blocks, represented by the `TPooledMemManagerItem` (703) record. It should not be used directly, but should be descended from and the descendent should implement the actual memory manager.

See also: `TPooledMemManagerItem` (703)

### 30.6.2 Method overview

Page	Method	Description
<a href="#">705</a>	<code>Clear</code>	Clears the list
<a href="#">705</a>	<code>Create</code>	Creates a new instance of the <code>TPooledMemManager</code> class
<a href="#">706</a>	<code>Destroy</code>	Removes an instance of <code>TPooledMemManager</code> class from memory

### 30.6.3 Property overview

Page	Properties	Access	Description
<a href="#">707</a>	<code>AllocatedCount</code>	r	Total number of allocated items in the list
<a href="#">706</a>	<code>Count</code>	r	Number of items in the list
<a href="#">707</a>	<code>FreeCount</code>	r	Number of free items in the list
<a href="#">707</a>	<code>FreedCount</code>	r	Total number of freed items in the list.
<a href="#">706</a>	<code>MaximumFreeCountRatio</code>	rw	Maximum ratio of free items over total items
<a href="#">706</a>	<code>MinimumFreeCount</code>	rw	Minimum count of free items in the list

### 30.6.4 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` (707)

### 30.6.5 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` (706)

### 30.6.6 TPooledMemManager.Destroy

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

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` calls `Clear` (705) and then calls the inherited `destroy`.

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

See also: `TPooledMemManager.Create` (705)

### 30.6.7 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` (706)

### 30.6.8 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` (706)

### 30.6.9 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` (707), `TPooledMemManager.AllocatedCount` (707), `TPooledMemManager.FreedCount` (707)

### 30.6.10 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` (706), `TPooledMemManager.AllocatedCount` (707), `TPooledMemManager.FreedCount` (707)

### 30.6.11 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` (706), `TPooledMemManager.FreeCount` (707), `TPooledMemManager.FreedCount` (707)

### 30.6.12 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` (706), `TPooledMemManager.FreeCount` (707), `TPooledMemManager.AllocatedCount` (707)

# Chapter 31

## Reference for unit 'process'

### 31.1 Used units

Table 31.1: Used units by unit 'process'

Name	Page
Classes	??
Math	??
Pipes	<a href="#">697</a>
sysutils	??

### 31.2 Overview

The `Process` unit contains the code for the `TProcess` ([713](#)) 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.

### 31.3 Constants, types and variables

#### 31.3.1 Types

```
TOnRunCommandEvent = procedure(Sender: TObject; Context: TObject;  
    Status: TRunCommandEventCode;  
    const Message: string) of object
```

`TOnRunCommandEvent` is the event handler prototype for the various events emitted by the `TProcess` ([713](#)) class during the `RunCommandLoop` ([719](#)) call.

```
TprocessChar = Char
```

TProcessChar is a single-byte character in the single-byte version of TProcess, but is a 2-byte character in the unicode version of TProcess.

TProcessClass = Class of TPROCESS

Class of TProcess

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

Table 31.2: 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.
poDetached	
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)
poPassInput	Pass standard input handle on to new process
poRunIdle	
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.

When a new process is started using TProcess.Execute (716), these options control the way the process is started. Note that not all options are supported on all platforms.

TProcessOptions = Set of TProcessOption

Set of TProcessOption (709).

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

Table 31.3: Enumeration values for type TProcessPriority

Value	Explanation
ppAboveNormal	Above normal priority
ppBelowNormal	Below normal priority
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.

`TProcessString = String`

`TProcessString` is a single-byte string in the single-byte version of `TProcess`, but is a 2-byte (unicode) string in the unicode version of `TProcess`.

`TProcessStringList = TStringList`

`TProcessStringList` is an alias for `TProcessStrings` (710) in unicode code, or an alias for the `#classes.TStringList` (??) class in single-byte string mode.

`TProcessStrings = TString`

`TProcessStrings` is a simple string list class which, depending on the version (unicode or not) contains unicode strings or single-byte strings: in the latter case it is an alias for the `#classes.TStrings` (??) class.

`TRunCommandEventCode = (RunCommandIdle, RunCommandReadOutputString, RunCommandReadOutputStream, RunCommandFinished, RunCommandException)`

Table 31.4: Enumeration values for type `TRunCommandEventCode`

Value	Explanation
<code>RunCommandException</code>	An error happened during reading of the command
<code>RunCommandFinished</code>	The command finished
<code>RunCommandIdle</code>	No data was available for reading
<code>RunCommandReadOutputStream</code>	Output from the command was read
<code>RunCommandReadOutputString</code>	Output from the command was read as a string

`TRunCommandEventCode` is an enumerated type indicating the stage at which a process is during the `RunCommandLoop` (719) call, reported through the `TProcess.OnRunCommandEvent` (723) event handler.

`TRunCommandEventCodeSet = Set of TRunCommandEventCode`

`TRunCommandEventCodeSet` is a set of `TRunCommandEventCode` (710) values.

`TShowWindowOptions = (swoNone, swoHIDE, swoMaximize, swoMinimize, swoRestore, swoShow, swoShowDefault, swoShowMaximized, swoShowMinimized, swoshowMinNOActive, swoShowNA, swoShowNoActivate, swoShowNormal)`

Table 31.5: 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

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)
```

Table 31.6: Enumeration values for type TStartupOption

Value	Explanation
suoUseCountChars	Use the console character width as specified in TProcess (713).
suoUseFillAttribute	Use the console fill attribute as specified in TProcess (713).
suoUsePosition	Use the window sizes as specified in TProcess (713).
suoUseShowWindow	Use the Show Window options specified in TShowWindowOption (711)
suoUseSize	Use the window sizes as specified in TProcess (713)

These options are mainly for Win32, and determine what should be done with the application once it's started.

```
TStartupOptions = Set of TStartupOption
```

Set of TStartupOption (711).

### 31.3.2 Variables

```
DefaultTProcess : TProcessClass = TPROCESS
```

DefaultTProcess is the process class used by the RunCommand (712) and RunCommandInDir (712) calls. You can set it to customize the process class to use during these calls. By default the TProcess class is used.



## 31.4 Procedures and functions

### 31.4.1 CommandToList

**Synopsis:** Convert a command-line to a list of command options

**Declaration:** `procedure CommandToList(S: TProcessString; List: TProcessStrings)`

**Visibility:** default

**Description:** `CommandToList` splits the string `S` in command-line arguments that are returned, one per item, in the `List` stringlist. Command-line arguments are separated by whitespace (space, tab, CR and LF characters). If an argument needs to contain a space character, it can be surrounded in quote characters (single or double quotes).

**Errors:** There is currently no way to specify a quote character inside a quoted argument.

**See also:** `TProcess.CommandLine` ([724](#))

### 31.4.2 RunCommand

**Synopsis:** Execute a command in the current working directory

**Declaration:** `function RunCommand(const exename: TProcessString;  
const commands: Array of TProcessString;  
out outputstring: string; Options: TProcessOptions;  
SWOptions: TShowWindowOptions) : Boolean`  
`function RunCommand(const cmdline: TProcessString;  
out outputstring: string) : Boolean`

**Visibility:** default

**Description:** `RunCommand` runs `RunCommandInDir` ([712](#)) with an empty current working directory.

The version using `CmdLine` attempts to split the command line in a binary and separate command-line arguments. This version of the function is deprecated.

**See also:** `RunCommandInDir` ([712](#))

### 31.4.3 RunCommandInDir

**Synopsis:** Run a command in a specific directory.

**Declaration:** `function RunCommandInDir(const curdir: TProcessString;  
const exename: TProcessString;  
const commands: Array of TProcessString;  
out outputstring: string;  
out exitstatus: Integer;  
Options: TProcessOptions;  
SWOptions: TShowWindowOptions) : Integer`  
`function RunCommandInDir(const curdir: TProcessString;  
const exename: TProcessString;  
const commands: Array of TProcessString;  
out outputstring: string;  
Options: TProcessOptions;  
SWOptions: TShowWindowOptions) : Boolean`  
`function RunCommandInDir(const curdir: TProcessString;`

```
const cmdline: TProcessString;
out outputstring: string) : Boolean
```

Visibility: default

**Description:** `RunCommandInDir` will execute binary `exename` with command-line options `commands`, setting `curdir` as the current working directory for the command. The Options (709) are taken into consideration (`poRunSuspended`, `poWaitOnExit` are removed from the set). The output of the command is captured, and returned in the string `OutputString`. The function waits for the command to finish, and returns `True` if the command was started successfully, `False` otherwise. In the case where the return value is an integer, it is zero for success, and -1 on error.

If a `ExitStatus` parameter is specified the exit status of the command is returned in this parameter.

The version using `cmdline` attempts to split the command line in a binary and separate command-line arguments. This version of the function is deprecated.

**Errors:** On error, `False` is returned.

See also: `TProcess` (713), `RunCommand` (712), `TProcessOptions` (709)

## 31.5 EProcess

### 31.5.1 Description

Exception raised when an error occurs in a `TProcess` routine.

See also: `TProcess` (713)

## 31.6 TPROCESS

### 31.6.1 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` (724) property to the full pathname of the program that should be executed, and call `Execute` (716). To determine whether the process is still running (i.e. has not stopped executing), the `Running` (729) property can be checked.

More advanced techniques can be used with the Options (727) settings.

See also: `Create` (715), `Execute` (716), `Running` (729), `CommandLine` (724), `Options` (727)

**31.6.2 Method overview**

Page	Method	Description
<a href="#">716</a>	CloseInput	Close the input stream of the process
<a href="#">717</a>	CloseOutput	Close the output stream of the process
<a href="#">717</a>	CloseStderr	Close the error stream of the process
<a href="#">715</a>	Create	Create a new instance of the <code>TProcess</code> class.
<a href="#">716</a>	Destroy	Destroy this instance of <code>TProcess</code>
<a href="#">716</a>	Execute	Execute the program with the given options
<a href="#">718</a>	ReadInputStream	Read available data from input stream
<a href="#">717</a>	Resume	Resume execution of a suspended process
<a href="#">719</a>	RunCommandLoop	Execute command and collect output in strings
<a href="#">717</a>	Suspend	Suspend a running process
<a href="#">718</a>	Terminate	Terminate a running process
<a href="#">718</a>	WaitOnExit	Wait for the program to stop executing.

### 31.6.3 Property overview

Page	Properties	Access	Description
<a href="#">723</a>	Active	rw	Start or stop the process.
<a href="#">724</a>	ApplicationName	rw	Name of the application to start (deprecated)
<a href="#">724</a>	CommandLine	rw	Command-line to execute (deprecated)
<a href="#">726</a>	ConsoleTitle	rw	Title of the console window
<a href="#">726</a>	CurrentDirectory	rw	Working directory of the process.
<a href="#">726</a>	Desktop	rw	Desktop on which to start the process.
<a href="#">726</a>	Environment	rw	Environment variables for the new process
<a href="#">724</a>	Executable	rw	Executable name. Supersedes <code>CommandLine</code> and <code>ApplicationName</code> .
<a href="#">722</a>	ExitCode	r	Exit code of the process
<a href="#">722</a>	ExitStatus	r	Exit status of the process.
<a href="#">731</a>	FillAttribute	rw	Color attributes of the characters in the console window (Windows only)
<a href="#">719</a>	Handle	r	Handle of the process
<a href="#">722</a>	InheritHandles	rw	Should the created process inherit the open handles of the current process.
<a href="#">721</a>	Input	r	Stream connected to standard input of the process.
<a href="#">723</a>	OnRunCommandEvent	rw	Event handler, called when <code>RunCommandLoop</code> is executing
<a href="#">727</a>	Options	rw	Options to be used when starting the process.
<a href="#">721</a>	Output	r	Stream connected to standard output of the process.
<a href="#">725</a>	Parameters	rw	Command-line arguments. Supersedes <code>CommandLine</code> .
<a href="#">723</a>	PipeBufferSize	rw	Buffer size to be used when using pipes
<a href="#">727</a>	Priority	rw	Priority at which the process is running.
<a href="#">720</a>	ProcessHandle	r	Alias for <code>Handle</code> ( <a href="#">719</a> )
<a href="#">720</a>	ProcessID	r	ID of the process.
<a href="#">723</a>	RunCommandSleepTime	rw	Sleep time between attempts to collect data
<a href="#">729</a>	Running	r	Determines whether the process is still running.
<a href="#">729</a>	ShowWindow	rw	Determines how the process main window is shown (Windows only)
<a href="#">728</a>	StartupOptions	rw	Additional (Windows) startup options
<a href="#">721</a>	Stderr	r	Stream connected to standard diagnostic output of the process.
<a href="#">720</a>	ThreadHandle	r	Main process thread handle
<a href="#">720</a>	ThreadID	r	ID of the main process thread
<a href="#">729</a>	WindowColumns	rw	Number of columns in console window (windows only)
<a href="#">730</a>	WindowHeight	rw	Height of the process main window
<a href="#">730</a>	WindowLeft	rw	X-coordinate of the initial window (Windows only)
<a href="#">719</a>	WindowRect	rw	Positions for the main program window.
<a href="#">730</a>	WindowRows	rw	Number of rows in console window (Windows only)
<a href="#">731</a>	WindowTop	rw	Y-coordinate of the initial window (Windows only)
<a href="#">731</a>	WindowWidth	rw	Height of the process main window (Windows only)
<a href="#">732</a>	XTermProgram	rw	XTerm program to use (UNIX only)

### 31.6.4 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.

### 31.6.5 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:** `Create` (715)

### 31.6.6 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` (724), applying as much as of the specified options as supported on the current platform.

If the `poWaitOnExit` option is specified in `Options` (727), 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` (718) call can be used to wait for it to close, or the `Running` (729) call can be used to check whether it is still running.

The `TProcess.Terminate` (718) call can be used to terminate the program if it is still running, or the `Suspend` (717) call can be used to temporarily stop the program's execution.

The `ExitStatus` (722) function can be used to check the program's exit status, after it has stopped executing.

**Errors:** On error a `EProcess` (713) exception is raised.

**See also:** `TProcess.Running` (729), `TProcess.WaitOnExit` (718), `TProcess.Terminate` (718), `TProcess.Suspend` (717), `TProcess.Resume` (717), `TProcess.ExitStatus` (722), `TProcess.ExitCode` (722)

### 31.6.7 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: [Input \(721\)](#), [StdErr \(721\)](#), [Output \(721\)](#), [CloseOutput \(717\)](#), [CloseStdErr \(717\)](#)

### 31.6.8 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: [Output \(721\)](#), [Input \(721\)](#), [StdErr \(721\)](#), [CloseInput \(716\)](#), [CloseStdErr \(717\)](#)

### 31.6.9 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: [Output \(721\)](#), [Input \(721\)](#), [StdErr \(721\)](#), [CloseInput \(716\)](#), [CloseStdErr \(717\)](#)

### 31.6.10 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 \(727\)](#).

**Errors:** None.

See also: [TProcess.Suspend \(717\)](#), [TProcess.Options \(727\)](#), [TProcess.Execute \(716\)](#), [TProcess.Terminate \(718\)](#)

### 31.6.11 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 \(717\)](#) call.

`Suspend` is fundamentally different from [TProcess.Terminate \(718\)](#) which actually stops the process.

Errors: On error, a nonzero result is returned.

See also: [TProcess.Options \(727\)](#), [TProcess.Resume \(717\)](#), [TProcess.Terminate \(718\)](#), [TProcess.Execute \(716\)](#)

### 31.6.12 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 \(722\)](#), [TProcess.Suspend \(717\)](#), [TProcess.Execute \(716\)](#), [TProcess.WaitForExit \(718\)](#), [TProcess.ExitCode \(722\)](#)

### 31.6.13 TPROCESS.WaitForExit

Synopsis: Wait for the program to stop executing.

Declaration: `function WaitForExit : Boolean`  
`function WaitForExit(Timeout: DWord) : Boolean`

Visibility: public

Description: `WaitForExit` 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.

The `TimeOut` argument can be used to specify a timeout in milliseconds. If omitted, the call will wait indefinitely.

Errors: On error, `False` is returned. No extended error information is available, as it is highly system dependent.

See also: [TProcess.ExitStatus \(722\)](#), [TProcess.Terminate \(718\)](#), [TProcess.Running \(729\)](#), [TProcess.ExitCode \(722\)](#)

### 31.6.14 TPROCESS.ReadInputStream

Synopsis: Read available data from input stream

Declaration: `function ReadInputStream(p: TInputPipeStream; var BytesRead: Integer;`  
`var DataLength: Integer; var Data: string;`  
`MaxLoops: Integer) : Boolean; Virtual`  
`function ReadInputStream(p: TInputPipeStream; data: TStream;`  
`MaxLoops: Integer) : Boolean; Virtual`

Visibility: public

**Description:** `ReadInputStream` reads data from the given input pipe stream `p` after checking that data is available. It returns `True` if data was successfully read from the file handle. In the variant with a string data, the data is placed in the string `Data`, and `DataLength` is updated with the new length, `BytesRead` is updated with the amount of bytes read. `MaxLoop` determines how often an attempt at reading data is made.

In the variant with a stream, the available data is simply written to the stream.

**Errors:** None.

### 31.6.15 TPROCESS.RunCommandLoop

**Synopsis:** Execute command and collect output in strings

**Declaration:** `function RunCommandLoop(out outputstring: string;  
out stderrstring: string;  
out anexitstatus: Integer) : Integer; Virtual`

**Visibility:** public

**Description:** `RunCommandLoop` executes the command, and runs a loop to read output of the command: the output of the command is returned in the `outputstring` parameter, and the error output is returned in the `stderrstring` string.

During collection of data or on error, the `TProcess.OnRunCommandEvent` (723) event handler is called during the various stages of the call. If it is not explicitly set, a sleep period specified by `TPROCESS.RunCommandSleepTime` (723) is interjected between the various read calls.

The return value of this call is 1 for error, zero for success.

See also: `TProcess.OnRunCommandEvent` (723), `TPROCESS.RunCommandSleepTime` (723)

### 31.6.16 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

### 31.6.17 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` (716) has been called. It is not reset after the process stopped.

See also: `TProcess.ThreadHandle` (720), `TProcess.ProcessID` (720), `TProcess.ThreadID` (720)



### 31.6.18 TPROCESS.ProcessHandle

Synopsis: Alias for Handle (719)

Declaration: `Property ProcessHandle : THandle`

Visibility: public

Access: Read

Description: `ProcessHandle` equals `Handle` (719) and is provided for completeness only.

See also: `TProcess.Handle` (719), `TProcess.ThreadHandle` (720), `TProcess.ProcessID` (720), `TProcess.ThreadID` (720)

### 31.6.19 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` (716) has been called. It is not reset after the process stopped.

See also: `TProcess.Handle` (719), `TProcess.ProcessID` (720), `TProcess.ThreadID` (720)

### 31.6.20 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` (716) has been called. It is not reset after the process stopped.

See also: `TProcess.Handle` (719), `TProcess.ThreadHandle` (720), `TProcess.ThreadID` (720)

### 31.6.21 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` (716) has been called. It is not reset after the process stopped.

See also: `TProcess.ProcessID` (720), `TProcess.Handle` (719), `TProcess.ThreadHandle` (720)

### 31.6.22 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` (727).

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` (721), `TProcess.StdErr` (721), `TProcess.Options` (727), `TProcessOption` (709)

### 31.6.23 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` (727).

The `Output` stream also contains any data written to standard diagnostic output (`stderr`) when the `poStdErrToOutPut` flag is used in `Options` (727).

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` (721), `TProcess.StdErr` (721), `TProcess.Options` (727), `TProcessOption` (709)

### 31.6.24 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` (727).

The `Output` stream equals the `Output` (721) when the `poStdErrToOutPut` flag is used in `Options` (727).

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` (721), `TProcess.Output` (721), `TProcess.Options` (727), `TProcessOption` (709)

### 31.6.25 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 OS for the process when it stopped executing: Normally, this is the exit code of the process.

The value of this property is only meaningful when the process has finished executing. If it is not yet running then the value is -1. (it was zero in earlier versions of FPC)

See also: `TProcess.Running` (729), `TProcess.Terminate` (718), `TProcess.ExitCode` (722)

### 31.6.26 TPROCESS.ExitCode

**Synopsis:** Exit code of the process

**Declaration:** `Property ExitCode : Integer`

**Visibility:** public

**Access:** Read

**Description:** `ExitCode` is the actual exit code of the process. On UNIX, this may differ from the `ExitStatus` (708) value if the process was terminated by a signal: in that case `ExitStatus` is the raw exit status as reported by one of the UNIX `Wait` command, and `ExitCode` is the exit code reported by the program.

See also: `TProcess.ExitStatus` (722), `TProcess.Running` (729), `TProcess.WaitOnExit` (718), `TProcess.Terminate` (718)

### 31.6.27 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` (721), `TProcess.Output` (721), `TProcess.StdErr` (721)

### 31.6.28 TPROCESS.OnRunCommandEvent

**Synopsis:** Event handler, called when `RunCommandLoop` is executing

**Declaration:** `Property OnRunCommandEvent : TOnRunCommandEvent`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** `OnRunCommandEvent` is a progress report callback, called at various stages of the `TProcess.RunCommandLoop` (719) call and when an exception occurs.

See also: `TProcess.RunCommandLoop` (719)

### 31.6.29 TPROCESS.RunCommandSleepTime

**Synopsis:** Sleep time between attempts to collect data

**Declaration:** `Property RunCommandSleepTime : Integer`

**Visibility:** `public`

**Access:** `Read,Write`

**Description:** Sleep time between attempts to collect data

### 31.6.30 TPROCESS.PipeBufferSize

**Synopsis:** Buffer size to be used when using pipes

**Declaration:** `Property PipeBufferSize : Cardinal`

**Visibility:** `published`

**Access:** `Read,Write`

**Description:** `PipeBufferSize` indicates the buffer size used when creating pipes (when `soUsePipes` is specified in `Options`). This option is not respected on all platforms (currently only Windows uses this).

See also: `#fcl.pipes.CreatePipeHandles` (697)

### 31.6.31 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` (716), `TProcess.Terminate` (718)

### 31.6.32 TPROCESS.ApplicationName

**Synopsis:** Name of the application to start (deprecated)

**Declaration:** `Property ApplicationName : TProcessString; deprecated;`

**Visibility:** published

**Access:** Read,Write

**Description:** `ApplicationName` is an alias for `TProcess.CommandLine` (724). It's mostly for use in the Windows `CreateProcess` call. If `CommandLine` is not set, then `ApplicationName` will be used instead.

`ApplicationName` is deprecated. New code should use `Executable` (724) instead, and leave `ApplicationName` empty.

See also: `TProcess.CommandLine` (724), `TProcess.Executable` (724), `TProcess.Parameters` (725)

### 31.6.33 TPROCESS.CommandLine

**Synopsis:** Command-line to execute (deprecated)

**Declaration:** `Property CommandLine : TProcessString; deprecated;`

**Visibility:** published

**Access:** Read,Write

**Description:** `CommandLine` is deprecated. To avoid problems with command-line options with spaces in them and the quoting problems that this entails, it has been superseded by the properties `TProcess.Executable` (724) and `TProcess.Parameters` (725), which should be used instead of `CommandLine`. New code should leave `CommandLine` empty.

`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` (724), `TProcess.Executable` (724), `TProcess.Parameters` (725)

### 31.6.34 TPROCESS.Executable

**Synopsis:** Executable name. Supersedes `CommandLine` and `ApplicationName`.

**Declaration:** `Property Executable : TProcessString`

**Visibility:** published

Access: Read,Write

**Description:** `Executable` is the name of the executable to start. It should not contain any command-line arguments. If no path is given, it will be searched in the `PATH` environment variable.

The extension must be given, none will be added by the component itself. It may be that the OS adds the extension, but this behaviour is not guaranteed.

Arguments should be passed in `TProcess.Parameters` (725).

`Executable` supersedes the `TProcess.CommandLine` (724) and `TProcess.ApplicationName` (724) properties, which have been deprecated. However, if either of `CommandLine` or `ApplicationName` is specified, they will be used instead of `Executable`.

See also: `CommandLine` (724), `ApplicationName` (724), `Parameters` (725)

### 31.6.35 TPROCESS.Parameters

**Synopsis:** Command-line arguments. Supersedes `CommandLine`.

**Declaration:** `Property Parameters : TProcessStrings`

**Visibility:** published

Access: Read,Write

**Description:** `Parameters` contains the command-line arguments that should be passed to the program specified in `Executable` (724).

Commandline arguments should be specified one per item in `Parameters`: each item in `Parameters` will be passed as a separate command-line item. It is therefor not necessary to quote whitespace in the items. As a consequence, it is not allowed to specify multiple command-line parameters in 1 item in the stringlist. If a command needs 2 options `-t` and `-s`, the following is not correct:

```
With Parameters do
begin
  add(' -t -s' );
end;
```

Instead, the code should read:

```
With Parameters do
begin
  add(' -t' );
  Add(' -s' );
end;
```

**Remark** Note that `Parameters` is ignored if either of `CommandLine` or `ApplicationName` is specified. It can only be used with `Executable`.

**Remark** The idea of using `Parameters` is that they are passed unmodified to the operating system. On Windows, a single command-line string must be constructed, and each parameter is surrounded by double quote characters if it contains a space. The programmer must not quote parameters with spaces.

See also: `Executable` (724), `CommandLine` (724), `ApplicationName` (724)

### 31.6.36 TPROCESS.ConsoleTitle

Synopsis: Title of the console window

Declaration: `Property ConsoleTitle : TProcessString`

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` ([729](#)), `TProcess.WindowRows` ([730](#))

### 31.6.37 TPROCESS.CurrentDirectory

Synopsis: Working directory of the process.

Declaration: `Property CurrentDirectory : TProcessString`

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` ([726](#))

### 31.6.38 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` ([721](#)), `TProcess.Output` ([721](#)), `TProcess.StdErr` ([721](#))

### 31.6.39 TPROCESS.Environment

Synopsis: Environment variables for the new process

Declaration: `Property Environment : TProcessStrings`

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` (727)

### 31.6.40 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` (716) call is made.

Table 31.7:

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` (709), `TProcessOptions` (709), `TProcess.Priority` (727), `TProcess.StartupOptions` (728)

### 31.6.41 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 31.8:

Priority	Meaning
ppHigh	The process runs at higher than normal priority.
ppIdle	The process only runs when the system is idle (i.e. has nothing else to do)
ppNormal	The process runs at normal priority.
ppRealTime	The process runs at real-time priority.

Note that not all priorities can be set by any user. Usually, only users 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 31.9:

Priority	Nice value
ppHigh	20
ppIdle	20
ppNormal	0
ppRealTime	-20

See also: `TProcessPriority` ([709](#))

### 31.6.42 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.

Table 31.10:

Priority	Meaning
<code>suoUseShowWindow</code>	Use the Show Window options specified in <code>ShowWindow</code> ( <a href="#">729</a> )
<code>suoUseSize</code>	Use the specified window sizes
<code>suoUsePosition</code>	Use the specified window sizes.
<code>suoUseCountChars</code>	Use the specified console character width.
<code>suoUseFillAttribute</code>	Use the console fill attribute specified in <code>FillAttribute</code> ( <a href="#">731</a> ).

See also: `TProcess.ShowWindow` ([729](#)), `TProcess.WindowHeight` ([730](#)), `TProcess.WindowWidth` ([731](#)), `TProcess.WindowLeft` ([730](#)), `TProcess.WindowTop` ([731](#)), `TProcess.WindowColumns` ([729](#)), `TProcess.WindowRows` ([730](#)), `TProcess.FillAttribute` ([731](#))

**31.6.43 TPROCESS.Running**

Synopsis: Determines whether 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 (718), TProcess.Active (723), TProcess.ExitStatus (722), TProcess.ExitCode (722)

**31.6.44 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 31.11:

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

**31.6.45 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 (728)

See also: `TProcess.WindowHeight` (730), `TProcess.WindowWidth` (731), `TProcess.WindowLeft` (730), `TProcess.WindowTop` (731), `TProcess.WindowRows` (730), `TProcess.FillAttribute` (731), `TProcess.StartupOptions` (728)

### 31.6.46 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` (728)

See also: `TProcess.WindowWidth` (731), `TProcess.WindowLeft` (730), `TProcess.WindowTop` (731), `TProcess.WindowColumns` (729), `TProcess.WindowRows` (730), `TProcess.FillAttribute` (731), `TProcess.StartupOptions` (728)

### 31.6.47 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` (728)

See also: `TProcess.WindowHeight` (730), `TProcess.WindowWidth` (731), `TProcess.WindowTop` (731), `TProcess.WindowColumns` (729), `TProcess.WindowRows` (730), `TProcess.FillAttribute` (731), `TProcess.StartupOptions` (728)

### 31.6.48 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` (728)

See also: `TProcess.WindowHeight` (730), `TProcess.WindowWidth` (731), `TProcess.WindowLeft` (730), `TProcess.WindowTop` (731), `TProcess.WindowColumns` (729), `TProcess.FillAttribute` (731), `TProcess.StartupOptions` (728)

### 31.6.49 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` (728)

See also: `TProcess.WindowHeight` (730), `TProcess.WindowWidth` (731), `TProcess.WindowLeft` (730), `TProcess.WindowColumns` (729), `TProcess.WindowRows` (730), `TProcess.FillAttribute` (731), `TProcess.StartupOptions` (728)

### 31.6.50 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` (728)

See also: `TProcess.WindowHeight` (730), `TProcess.WindowLeft` (730), `TProcess.WindowTop` (731), `TProcess.WindowColumns` (729), `TProcess.WindowRows` (730), `TProcess.FillAttribute` (731), `TProcess.StartupOptions` (728)

### 31.6.51 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` (730), `TProcess.WindowWidth` (731), `TProcess.WindowLeft` (730), `TProcess.WindowTop` (731), `TProcess.WindowColumns` (729), `TProcess.WindowRows` (730), `TProcess.StartupOptions` (728)

### 31.6.52 TPROCESS.XTermProgram

Synopsis: XTerm program to use (UNIX only)

Declaration: `Property XTermProgram : string`

Visibility: published

Access: Read, Write

Description: `XTermProgram` can be used to specify the console program to use when `poConsole` is specified in `TProcess.Options` (727).

If none is specified, `DetectXTerm` (708) is used to detect the terminal program to use. the list specified in `TryTerminals` is tried. If none is found, then the `DESKTOP_SESSION` environment variable is examined:

**kdekonsole** is used if it is found.

**gnome**gnome-terminal is used if it is found

**windowmaker**aterm or xterm are used if found.

If after all this, no terminal is found, then a list of default programs is tested: 'x-terminal-emulator', 'xterm', 'aterm', 'wterm', 'rxvt'

See also: `TProcess.Options` (727), `DetectXTerm` (708)

## Chapter 32

# Reference for unit 'RttiUtils'

### 32.1 Used units

Table 32.1: Used units by unit 'RttiUtils'

Name	Page
Classes	??
StrUtils	??
sysutils	??
TypeInfo	??

### 32.2 Overview

The `rttiutils` unit is a unit providing simplified access to the RTTI information from published properties using the `TPropInfoList` (736) 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` (738) class can be used for this. The implementation is based on the `apputils` unit from `RXLib` by *AO ROSNO* and *Master-Bank*

### 32.3 Constants, types and variables

#### 32.3.1 Constants

```
sPropNameDelimiter : string = '_'
```

Separator used when constructing section/key names

#### 32.3.2 Types

```
TEraseSectEvent = procedure(const ASection: string) of object
```

`TEraseSectEvent` is used by `TPropsStorage` (738) 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.

```
TPropStorageOption = (psoAlwaysStoreStringsCount)
```

Table 32.2: Enumeration values for type TPropStorageOption

Value	Explanation
psoAlwaysStoreStringsCount	Always store the count of strings. Default is not to store the count

TPropStorageOption is the enumeration type used in the TPropsStorage.Options (741) property of TPropsStorage (738)

```
TPropStorageOptions = Set of TPropStorageOption
```

TPropStorageOptions is the set of TPropStorageOption used in TPropsStorage.Options (741).

```
TReadStrEvent = function(const ASection: string; const Item: string
;
const Default: string) : string of
object
```

TReadStrEvent is used by TPropsStorage (738) 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 (738) 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.

### 32.3.3 Variables

```
FindGlobalComponentCallBack : TFindComponentEvent
```

FindGlobalComponentCallBack is called by UpdateStoredList (735) 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.

## 32.4 Procedures and functions

### 32.4.1 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` (735), `TPropsStorage.StoreObjectsProps` (740) or `TPropsStorage.LoadObjectsProps` (740).

**See also:** `ParseStoredItem` (735), `UpdateStoredList` (735), `TPropsStorage.StoreObjectsProps` (740), `TPropsStorage.LoadObjectsProps` (740)

### 32.4.2 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` (735). 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` (735), `UpdateStoredList` (735), `TPropsStorage.StoreObjectsProps` (740), `TPropsStorage.LoadObjectsProps` (740)

### 32.4.3 UpdateStoredList

**Synopsis:** Update a stringlist with object references

**Declaration:** `procedure UpdateStoredList (AComponent: TComponent;  
AStoredList: TStringList; FromForm: Boolean)`

**Visibility:** default

**Description:** `UpdateStoredList` will parse the strings in `AStoredList` using `ParseStoredItem` (735) 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` (734) 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 \(735\)](#), [TPropsStorage.StoreObjectsProps \(740\)](#), [TPropsStorage.LoadObjectsProps \(740\)](#), [FindGlobalComponentCallBack \(734\)](#)

## 32.5 TPropInfoList

### 32.5.1 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 \(738\)](#)

See also: [TPropsStorage \(738\)](#)

### 32.5.2 Method overview

Page	Method	Description
<a href="#">737</a>	Contains	Check whether a certain property is included
<a href="#">736</a>	Create	Create a new instance of <code>TPropInfoList</code>
<a href="#">737</a>	Delete	Delete property information from the list
<a href="#">736</a>	Destroy	Remove the <code>TPropInfoList</code> instance from memory
<a href="#">737</a>	Find	Retrieve property information based on name
<a href="#">737</a>	Intersect	Intersect 2 property lists

### 32.5.3 Property overview

Page	Properties	Access	Description
<a href="#">738</a>	Count	r	Number of items in the list
<a href="#">738</a>	Items	r	Indexed access to the property type pointers

### 32.5.4 TPropInfoList.Create

Synopsis: Create a new instance of `TPropInfoList`

Declaration: `constructor Create(AObject: TObject; Filter: TTypeKinds;  
Sorted: Boolean)`

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: [Destroy \(736\)](#)

### 32.5.5 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: [Create \(736\)](#)

### 32.5.6 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: [Find \(737\)](#), [Intersect \(737\)](#)

### 32.5.7 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: [Intersect \(737\)](#), [Contains \(737\)](#)

### 32.5.8 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 \(737\)](#) call.

Errors: No checking on the validity of `Index` is performed.

See also: [Intersect \(737\)](#)

### 32.5.9 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: [Delete \(737\)](#), [Contains \(737\)](#)

### 32.5.10 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: [Items \(738\)](#)

### 32.5.11 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: [Count \(738\)](#)

## 32.6 TPropsStorage

### 32.6.1 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.

See also: [TPropInfoList \(736\)](#)

### 32.6.2 Method overview

Page	Method	Description
<a href="#">739</a>	<code>LoadAnyProperty</code>	Load a property value
<a href="#">740</a>	<code>LoadObjectsProps</code>	Load a list of component properties
<a href="#">740</a>	<code>LoadProperties</code>	Load a list of properties
<a href="#">739</a>	<code>StoreAnyProperty</code>	Store a property value
<a href="#">740</a>	<code>StoreObjectsProps</code>	Store a list of component properties
<a href="#">739</a>	<code>StoreProperties</code>	Store a list of properties

### 32.6.3 Property overview

Page	Properties	Access	Description
<a href="#">741</a>	AObject	rw	Object to load or store properties from
<a href="#">743</a>	OnEraseSection	rw	Erase a section in storage
<a href="#">742</a>	OnReadString	rw	Read a string value from storage
<a href="#">742</a>	OnWriteString	rw	Write a string value to storage
<a href="#">741</a>	Options	rw	Options to take into account when saving or loading properties from the storage
<a href="#">742</a>	Prefix	rw	Prefix to use in storage
<a href="#">742</a>	Section	rw	Section name for storage

### 32.6.4 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` ([741](#)) property of `TPropsStorage`.

Errors: If the property pointer is invalid or `AObject` is invalid, an exception will be raised.

See also: `AObject` ([741](#)), `LoadAnyProperty` ([739](#)), `LoadProperties` ([740](#)), `StoreProperties` ([739](#))

### 32.6.5 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` ([741](#)) property of `TPropsStorage`.

Errors: If the property pointer is invalid or `AObject` is invalid, an exception will be raised.

See also: `AObject` ([741](#)), `StoreAnyProperty` ([739](#)), `LoadProperties` ([740](#)), `StoreProperties` ([739](#))

### 32.6.6 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` ([741](#)) object.

Errors: If an invalid property name is specified, an exception will be raised.

See also: `AObject` ([741](#)), `StoreAnyProperty` ([739](#)), `LoadProperties` ([740](#)), `LoadAnyProperty` ([739](#))

### 32.6.7 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` (741) object.

Errors: If an invalid property name is specified, an exception will be raised.

See also: `AObject` (741), `StoreAnyProperty` (739), `StoreProperties` (739), `LoadAnyProperty` (739)

### 32.6.8 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` (735) 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` (741) property.

Errors: If an invalid component is specified, an exception will be raised.

See also: `UpdateStoredList` (735), `StoreObjectsProps` (740), `LoadProperties` (740), `LoadAnyProperty` (739)

### 32.6.9 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` (735) 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` (741) property.

See also: `UpdateStoredList` (735), `LoadObjectsProps` (740), `LoadProperties` (740), `LoadAnyProperty` (739)

### 32.6.10 TPropsStorage.Options

**Synopsis:** Options to take into account when saving or loading properties from the storage

**Declaration:** `Property Options : TPropStorageOptions`

**Visibility:** public

**Access:** Read,Write

**Description:** `Options` can be used to tweak the behaviour of `TPropsStorage` when it loads or saves data to the storage. Currently the following options are available

**psoAlwaysStoreStringsCount** Always store the count of strings. Default is not to store the count

See also: `TPropStorageOptions` (734), `TPropStorageOption` (734)

### 32.6.11 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` (740) or `LoadObjectProps` (740) will destroy any value that this property might have.

See also: `LoadProperties` (740), `LoadAnyProperty` (739), `StoreProperties` (739), `StoreAnyProperty` (739), `StoreObjectProps` (740), `LoadObjectProps` (740)

### 32.6.12 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` ([742](#))

### 32.6.13 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` ([742](#))

### 32.6.14 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: `OnWriteString` ([742](#)), `OnEraseSection` ([743](#)), `TReadStrEvent` ([734](#))

### 32.6.15 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: `OnReadString` ([742](#)), `OnEraseSection` ([743](#)), `TWriteStrEvent` ([734](#))

### 32.6.16 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: `OnReadString` ([742](#)), `OnWriteString` ([742](#)), `TEraseSectEvent` ([733](#))



## Chapter 33

# Reference for unit 'simpleipc'

### 33.1 Used units

Table 33.1: Used units by unit 'simpleipc'

Name	Page
Classes	??
Contnrs	<a href="#">200</a>
syncobjs	<a href="#">876</a>
sysutils	??

### 33.2 Overview

The SimpleIPC unit provides classes to implement a simple, one-way IPC mechanism using string messages. It provides a TSimpleIPCServer ([761](#)) component for the server, and a TSimpleIPCClient ([758](#)) component for the client. The components are cross-platform, and should work both on Windows and UNIX-like systems.

The Unix implementation of the SimpleIPC unit uses file-based sockets. It will attempt to clean up any registered server socket files that were not removed cleanly.

It does this in the unit finalization code. It does not install a signal handler by itself, that is the task of the programmer. But program crashes (access violations and such) that are handled by the RTL will be handled gracefully.

This also means that if the process is killed with the KILL signal, it has no chance of removing the files (KILL signals cannot be caught), in which case socket files may remain in the file system. However, the client code attempts to cater for this and will remove the stale sockets if it detects them.

Under Windows, the communication is done through WM\_COPYDATA messages. Starting from Windows Vista it is forbidden to send messages between service applications and desktop applications, so a SimpleIPC client in a desktop application cannot connect to a SimpleIPC server in a service application and vice versa.

### 33.3 Constants, types and variables

#### 33.3.1 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.

```
SErrMessageQueueOverflow = 'Message queue overflow (limit %s)'
```

Too many messages in the message queue

```
SErrServerNotActive = 'Server with ID %s is not active.'
```

Error message if server is not active

```
SErrThreadContext =
    'This operation is illegal outside of IPC thread context.'
```

Thread context error message.

```
SErrThreadFailure = 'IPC thread failure.'
```

Thread failure message

#### 33.3.2 Constants

```
MsgVersion = 1
```

Current version of the messaging protocol

```
mtString = 1
```

String message type

```
mtUnknown = 0
```

Unknown message type

#### 33.3.3 Types

```
TIPCCClientCommClass = Class of TIPCCClientComm
```

TIPCCClientCommClass is used by TSimpleIPCCClient ([758](#)) to decide which kind of communication channel to set up.

```
TIPCMessageOverflowAction = (ipcmoaNone, ipcmoaDiscardOld,
    ipcmoaDiscardNew, ipcmoaError)
```

Table 33.2: Enumeration values for type TIPCMessageOverflowAction

Value	Explanation
ipcmoaDiscardNew	Discard the new message
ipcmoaDiscardOld	Discard the oldest message
ipcmoaError	Raise an error
ipcmoaNone	Do nothing, just add the message

TIPCMessageOverflowAction describes what will happen if the message queue hits the size limit for the queue.

Do nothing, just add the message

Discard the oldest message

Discard the new message

```
TIPCServerCommClass = Class of TIPCServerComm
```

TIPCServerCommClass is used by TSimpleIPCServer (761) to decide which kind of communication channel to set up.

```
TMessageQueueEvent = procedure(Sender: TObject; Msg: TIPCServerMsg
    )
    of object
```

TMessageQueueEvent is the signature of the event handler that is executed when a new message arrives on the server and the queue is full, and maxaction is ipcmoaError.

```
TMessageType = LongInt
```

TMessageType is provided for backward compatibility with earlier versions of the simpleipc unit.

### 33.3.4 Variables

```
DefaultIPCClientClass : TIPCCClientCommClass = Nil
```

DefaultIPCCClientClass is filled with a class pointer indicating which kind of communication protocol class should be instantiated by the TSimpleIPCCClient (758) 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).

```
DefaultIPCMessageOverflowAction : TIPCMessageOverflowAction = TSimpleIPCServer
    .DefaultMaxAction
```

DefaultIPCMessageOverflowAction is the default for the message queue overflow action when a new queue is made.

```
DefaultIPCMessageQueueLimit : Integer = TSimpleIPCServer
    .DefaultMaxQueue
```

DefaultIPCMessageOverflowAction is the default for the maximum message queue size when a new queue is made. A zero size means no limit.

```
DefaultIPCServerClass : TIPCServerCommClass = Nil
```

DefaultIPCServerClass is filled with a class pointer indicating which kind of communication protocol class should be instantiated by the TSimpleIPCServer (761) 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.

## 33.4 TMsgHeader

```
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.

## 33.5 EIPCErrors

### 33.5.1 Description

EIPCErrors is the exception used by the various classes in the SimpleIPC unit to report errors.

## 33.6 TIPCCClientComm

### 33.6.1 Description

TIPCCClientComm 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 TSimpleIPCCClient (758) class does not implement the messaging protocol by itself. Instead, it creates an instance of a (platform dependent) descendent of TIPCCClientComm which handles the internals of the communication protocol.

The server side of the messaging protocol is handled by the TIPCServerComm (750) component. The descendent components must always be implemented in pairs.

See also: TSimpleIPCCClient (758), TIPCServerComm (750), TSimpleIPCServer (761)

### 33.6.2 Method overview

Page	Method	Description
<a href="#">748</a>	Connect	Connect to the server
<a href="#">748</a>	Create	Create a new instance of the <code>TIPCCClientComm</code>
<a href="#">748</a>	Disconnect	Disconnect from the server
<a href="#">749</a>	SendMessage	Send a message
<a href="#">749</a>	ServerRunning	Check if the server is running.

### 33.6.3 Property overview

Page	Properties	Access	Description
<a href="#">749</a>	Owner	r	<code>TSimpleIPCCClient</code> instance for which communication must be handled.

### 33.6.4 TIPCCClientComm.Create

Synopsis: Create a new instance of the `TIPCCClientComm`

Declaration: `constructor Create(AOwner: TSimpleIPCCClient); Virtual`

Visibility: public

Description: `Create` instantiates a new instance of the `TIPCCClientComm` class, and stores the `AOwner` reference to the `TSimpleIPCCClient` ([758](#)) instance for which it will handle communication. It can be retrieved later using the `Owner` ([749](#)) property.

See also: `Owner` ([749](#)), `TSimpleIPCCClient` ([758](#))

### 33.6.5 TIPCCClientComm.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` ([757](#)) and `ServerInstance` ([760](#)) properties of the owning `TSimpleIPCCClient` ([758](#)) instance.

`Connect` is called by the `TSimpleIPCCClient.Connect` ([759](#)) call or when the `Active` ([757](#)) 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: `TSimpleIPCCClient.Connect` ([759](#)), `Active` ([757](#)), `Disconnect` ([748](#))

### 33.6.6 TIPCCClientComm.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 `TSimpleIPCCClient.Disconnect` (759) call or when the `Active` (757) 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: `TSimpleIPCCClient.Disconnect` (759), `Active` (757), `Connect` (748)

### 33.6.7 TIPCCClientComm.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 `TSimpleIPCCClient` (758) component.

See also: `TSimpleIPCCClient.InstanceID` (758), `TSimpleIPCCClient.ServerID` (758)

### 33.6.8 TIPCCClientComm.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.

### 33.6.9 TIPCCClientComm.Owner

**Synopsis:** `TSimpleIPCCClient` instance for which communication must be handled.

**Declaration:** `Property Owner : TSimpleIPCCClient`

**Visibility:** `public`

**Access:** `Read`

**Description:** `Owner` is the `TSimpleIPCCClient` (758) instance for which the communication must be handled. It cannot be changed, and must be specified when the `TIPCCClientComm` instance is created.

See also: `TSimpleIPCCClient` (758), `TIPCCClientComm.Create` (748)

## 33.7 TIPServerComm

### 33.7.1 Description

`TIPServerComm` 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` (761) class does not implement the messaging protocol by itself. Instead, it creates an instance of a (platform dependent) descendent of `TIPServerComm` which handles the internals of the communication protocol.

The client side of the messaging protocol is handled by the `TIPCClientComm` (747) component. The descendent components must always be implemented in pairs.

See also: `TSimpleIPCTServer` (761), `TIPCClientComm` (747)

### 33.7.2 Method overview

Page	Method	Description
750	Create	Create a new instance of the communication handler
751	PeekMessage	See if a message is available.
751	ReadMessage	Read message from the channel.
750	StartServer	Start the server-side of the communication channel
751	StopServer	Stop the server side of the communication channel.

### 33.7.3 Property overview

Page	Properties	Access	Description
752	InstanceID	r	Unique identifier for the communication channel.
752	Owner	r	<code>TSimpleIPCTServer</code> instance for which to handle transport

### 33.7.4 TIPServerComm.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` (752) property.

See also: `Owner` (752)

### 33.7.5 TIPServerComm.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` (757) property of the `TSimpleIPCServer` (761) instance is set to `True`.

If `Threaded` is `True` then a background thread is started which will check for new messages periodically (see also `TSimpleIPCServer.ThreadTimeOut` (768)). The arrival of new messages can be acted upon with `TSimpleIPCServer.OnMessageQueued` (766).

Errors: In case of an error, an `EIPCError` (747) exception is raised.

See also: `TSimpleIPCServer` (761), `TSimpleIPC.Active` (757), `TSimpleIPCServer.OnMessageQueued` (766), `TSimpleIPCServer.ThreadTimeOut` (768)

### 33.7.6 TIPCServerComm.StopServer

Synopsis: Stop the server side of the communication channel.

Declaration: `procedure StopServer; Virtual; Abstract`

Visibility: `public`

Description: `StopServer` 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` (757) property of the `TSimpleIPCServer` (761) instance is set to `False`.

Errors: In case of an error, an `EIPCError` (747) exception is raised.

See also: `TSimpleIPCServer` (761), `TSimpleIPC.Active` (757)

### 33.7.7 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` (751) call.

See also: `ReadMessage` (751)

### 33.7.8 TIPCServerComm.ReadMessage

Synopsis: Read message from the channel.

Declaration: `procedure ReadMessage; Virtual; Abstract`

Visibility: `public`

Description: `ReadMessage` reads the message for the channel, and stores the information in the data structures in the `Owner` class.

`ReadMessage` is a blocking call: if no message is available, the program will wait till a message arrives. Use `PeekMessage` (751) to see if a message is available.

See also: `TSimpleIPCServer` (761)



### 33.7.9 TIPCTServerComm.Owner

Synopsis: TSimpleIPCServer instance for which to handle transport

Declaration: Property Owner : TSimpleIPCServer

Visibility: public

Access: Read

Description: Owner refers to the TSimpleIPCServer (761) instance for which this instance of TSimpleIPCServer handles the transport. It is specified when the TIPCTServerComm is created.

See also: TSimpleIPCServer (761)

### 33.7.10 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.

## 33.8 TIPCTServerMsg

### 33.8.1 Description

TIPCTServerMsg is an auxiliary class used in the IPC server class TSimpleIPCServer (761). It keeps the data for 1 message. The set of messages is managed in TIPCTServerMsgQueue (754). There should normally be no need to use this class directly.

See also: TIPCTServerMsgQueue (754)

### 33.8.2 Method overview

Page	Method	Description
<a href="#">753</a>	Create	Create a new instance of a server message
<a href="#">753</a>	Destroy	Destroy an instance of a server message

### 33.8.3 Property overview

Page	Properties	Access	Description
<a href="#">753</a>	MsgType	rw	Message type
<a href="#">754</a>	OwnsStream	rw	Does the message own the stream
<a href="#">753</a>	Stream	r	Stream to store message data
<a href="#">754</a>	StringMessage	r	String message sent by client

### 33.8.4 TIPCTServerMsg.Create

Synopsis: Create a new instance of a server message

Declaration: `constructor Create`  
`constructor Create (AStream: TStream; AOwnsStream: Boolean)`

Visibility: `public`

Description: `Create` initializes the stream used to hold the message data.

See also: `TIPCTServerMsg.Destroy` ([753](#))

### 33.8.5 TIPCTServerMsg.Destroy

Synopsis: Destroy an instance of a server message

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` frees the stream used to hold the message data.

See also: `TIPCTServerMsg.Create` ([753](#))

### 33.8.6 TIPCTServerMsg.Stream

Synopsis: Stream to store message data

Declaration: `Property Stream : TStream`

Visibility: `public`

Access: `Read`

Description: `Stream` contains the message data as binary data.

See also: `TIPCTServerMsg.MsgType` ([753](#))

### 33.8.7 TIPCTServerMsg.MsgType

Synopsis: Message type

Declaration: `Property MsgType : TMessageType`

Visibility: `public`

Access: `Read, Write`

Description: `MsgType` simply contains the message type. The possible message types are application defined.

See also: `TIPCTServerMsg.Stream` ([753](#))

### 33.8.8 TIPCTServerMsg.OwnsStream

Synopsis: Does the message own the stream

Declaration: `Property OwnsStream : Boolean`

Visibility: public

Access: Read,Write

Description: `OwnsStream` can be set to true to signal that the message should release the stream when the message is destroyed. The initial value can be specified in the constructor.

See also: `TIPCTServerMsg.Create` (753)

### 33.8.9 TIPCTServerMsg.StringMessage

Synopsis: String message sent by client

Declaration: `Property StringMessage : string`

Visibility: public

Access: Read

Description: `StringMessage` is the message sent by the client as a string.

## 33.9 TIPCTServerMsgQueue

### 33.9.1 Description

`TIPCTServerMsgQueue` implements a message queue with FIFO characteristics. It has support for a maximum queue length (`TIPCTServerMsgQueue.MaxCount` (756)) and various ways of dealing with overflowing queue (`TIPCTServerMsgQueue.MaxAction` (756))

See also: `TIPCTServerMsgQueue.MaxCount` (756), `TIPCTServerMsgQueue.MaxAction` (756)

### 33.9.2 Method overview

Page	Method	Description
<a href="#">755</a>	<code>Clear</code>	Clear the message queue
<a href="#">755</a>	<code>Create</code>	Create a new message queue instance
<a href="#">755</a>	<code>Destroy</code>	Destroy server message queue instance
<a href="#">756</a>	<code>Pop</code>	Remove the oldest message from the queue
<a href="#">755</a>	<code>Push</code>	Add a new message to the queue

### 33.9.3 Property overview

Page	Properties	Access	Description
<a href="#">756</a>	<code>Count</code>	r	Number of messages in the queue
<a href="#">756</a>	<code>MaxAction</code>	rw	Action to take when the number of messages will exceed <code>MaxCount</code> .
<a href="#">756</a>	<code>MaxCount</code>	rw	Maximum number of messages in the queue, 0 for unlimited

### 33.9.4 TIPCTServerMsgQueue.Create

Synopsis: Create a new message queue instance

Declaration: `constructor Create`

Visibility: `public`

Description: `Create` creates a list to contain the messages, and initializes `TIPCTServerMsgQueue.MaxCount` (756) and `TIPCTServerMsgQueue.MaxAction` (756) with their default values (`DefaultIPCMessageQueueLimit` (747) and `DefaultIPCMessageOverflowAction` (746), respectively)

Note that the messages are owned by the queue till they are popped of the queue.

See also: `TIPCTServerMsgQueue.MaxCount` (756), `TIPCTServerMsgQueue.MaxAction` (756), `DefaultIPCMessageQueueLimit` (747), `DefaultIPCMessageOverflowAction` (746)

### 33.9.5 TIPCTServerMsgQueue.Destroy

Synopsis: Destroy server message queue instance

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` discards the remaining messages in the list and removes the message queue from memory.

See also: `TIPCTServerMsgQueue.Create` (755), `TIPCTServerMsgQueue.Clear` (755)

### 33.9.6 TIPCTServerMsgQueue.Clear

Synopsis: Clear the message queue

Declaration: `procedure Clear`

Visibility: `public`

Description: `Clear` discards the remaining messages in the list.

### 33.9.7 TIPCTServerMsgQueue.Push

Synopsis: Add a new message to the queue

Declaration: `procedure Push(AItem: TIPCTServerMsg)`

Visibility: `public`

Description: `Push` verifies if the message can be added to the queue (discarding old messages depending on the setting of `TIPCTServerMsgQueue.MaxAction` (756)) and adds the message `AItem` to the queue.

The message `AItem` is owned by the queue until it is popped off the queue.

Errors: If the maximum queue length is reached, and the `MaxAction` (756) is set to `ipcmoaError`, an exception will be raised.

See also: `TIPCTServerMsgQueue.MaxCount` (756), `TIPCTServerMsgQueue.MaxAction` (756), `TIPCTServerMsgQueue.Pop` (756)

### 33.9.8 TIPCTServerMsgQueue.Pop

Synopsis: Remove the oldest message from the queue

Declaration: `function Pop : TIPCTServerMsg`

Visibility: public

Description: `Pop` removes the oldest message from the queue if there is one, and returns it. If none exists, `Nil` is returned. The caller is responsible for freeing the message instance.

Errors: None.

See also: `TIPCTServerMsgQueue.Push` (755)

### 33.9.9 TIPCTServerMsgQueue.Count

Synopsis: Number of messages in the queue

Declaration: `Property Count : Integer`

Visibility: public

Access: Read

Description: `Count` is the current number of messages in the queue.

See also: `MaxCount` (756)

### 33.9.10 TIPCTServerMsgQueue.MaxCount

Synopsis: Maximum number of messages in the queue, 0 for unlimited

Declaration: `Property MaxCount : Integer`

Visibility: public

Access: Read,Write

Description: `MaxCount` is the maximum number of messages in the queue. When this amount is zero, the amount of messages is unlimited.

When a new message is pushed, and the `Count` (756) is equal to `MaxCount`, the `MaxAction` (756) property is examined to know what to do.

See also: `Count` (756), `MaxAction` (756)

### 33.9.11 TIPCTServerMsgQueue.MaxAction

Synopsis: Action to take when the number of messages will exceed `MaxCount`.

Declaration: `Property MaxAction : TIPCTMessageOverflowAction`

Visibility: public

Access: Read,Write

Description: `MaxAction` determines what will happen if the current `Count` (756) equals `MaxCount` (756) and a new message is put in the queue using `Push` (755):

Do nothing, just add the message

Discard the oldest message

Discard the new message

See also: `TIPCServerMsgQueue.Count` (756), `TIPCServerMsgQueue.MaxCount` (756), `TIPCServerMsgQueue.Push` (755), `TIPCMessageOverflowAction` (746)

## 33.10 TSimpleIPC

### 33.10.1 Description

`TSimpleIPC` is the common ancestor for the `TSimpleIPCServer` (761) and `TSimpleIPCClient` (758) classes. It implements some common properties between client and server.

See also: `TSimpleIPCServer` (761), `TSimpleIPCClient` (758)

### 33.10.2 Property overview

Page	Properties	Access	Description
<a href="#">757</a>	Active	rw	Communication channel active
<a href="#">757</a>	ServerID	rw	Unique server identification

### 33.10.3 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` (757) identification.

See also: `ServerID` (757)

### 33.10.4 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: `Active` (757)

## 33.11 TSimpleIPCClient

### 33.11.1 Description

`TSimpleIPCClient` is the client side of the simple IPC communication protocol. The client program should create a `TSimpleIPCClient` instance, set its `ServerID` (757) property to the unique name for the server it wants to send messages to, and then set the `Active` (757) property to `True` (or call `Connect` (758)).

After the connection with the server was established, messages can be sent to the server with the `SendMessage` (760) or `SendStringMessage` (760) calls.

See also: `TSimpleIPCServer` (761), `TSimpleIPC` (757), `TIPCClientComm` (747)

### 33.11.2 Method overview

Page	Method	Description
759	<code>Connect</code>	Connect to the server
758	<code>Create</code>	Create a new instance of <code>TSimpleIPCClient</code>
758	<code>Destroy</code>	Remove the <code>TSimpleIPCClient</code> instance from memory
759	<code>Disconnect</code>	Disconnect from the server
760	<code>SendMessage</code>	Send a message to the server
760	<code>SendStringMessage</code>	Send a string message to the server
760	<code>SendStringMessageFmt</code>	Send a formatted string message
759	<code>ServerRunning</code>	Check if the server is running.

### 33.11.3 Property overview

Page	Properties	Access	Description
760	<code>ServerInstance</code>	rw	Server instance identification

### 33.11.4 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: `Destroy` (758)

### 33.11.5 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: `Create` (758)

### 33.11.6 TSimpleIPCClient.Connect

Synopsis: Connect to the server

Declaration: `procedure Connect`

Visibility: `public`

Description: `Connect` connects to the server indicated in the `ServerID` (757) and `InstanceID` (758) properties. `Connect` is called automatically if the `Active` (757) property is set to `True`.

After a successful call to `Connect`, messages can be sent to the server using `SendMessage` (760) or `SendStringMessage` (760).

Calling `Connect` if the connection is already open has no effect.

Errors: If creating the connection fails, an `EIPCErrors` (747) exception may be raised.

See also: `ServerID` (757), `InstanceID` (758), `Active` (757), `SendMessage` (760), `SendStringMessage` (760), `Disconnect` (759)

### 33.11.7 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` (759). `Disconnect` is called automatically if the `Active` (757) 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 `EIPCErrors` (747) exception may be raised.

See also: `Active` (757), `Connect` (759)

### 33.11.8 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` (757) and `InstanceID` (758) 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: `Connect` (759)



### 33.11.9 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` (747) exception is raised.

See also: `Connect` (759), `SendStringMessage` (760)

### 33.11.10 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 `MsgType` and data `Msg` to the server. This is a convenience function: a small wrapper around the `SendMessage` (760) method

Errors: Same as for `SendMessage`.

See also: `SendMessage` (760), `Connect` (759), `SendStringMessageFmt` (760)

### 33.11.11 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 `MsgType` and message formatted from `Msg` and `Args` to the server. This is a convenience function: a small wrapper around the `SendStringMessage` (760) method

Errors: Same as for `SendMessage`.

See also: `SendMessage` (760), `Connect` (759), `SendStringMessage` (760)

### 33.11.12 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` (765) property set to `False`, and should match the server's `InstanceID` (765) property.

See also: `ServerID` (757), `GLocal` (765), `InstanceID` (765)

## 33.12 TSimpleIPCServer

### 33.12.1 Description

`TSimpleIPCServer` is the server side of the simple IPC communication protocol. The server program should create a `TSimpleIPCServer` instance, set its `ServerID` (757) property to a unique name for the system, and then set the `Active` (757) property to `True` (or call `StartServer` (762)).

After the server was started, it can check for availability of messages with the `PeekMessage` (763) call, and read the message with `ReadMessage` (763).

See also: `TSimpleIPCClient` (758), `TSimpleIPC` (757), `TIPCServerComm` (750)

### 33.12.2 Method overview

Page	Method	Description
762	Create	Create a new instance of <code>TSimpleIPCServer</code>
762	Destroy	Remove the <code>TSimpleIPCServer</code> instance from memory
763	GetMessageData	Read the data of the last message in a stream
763	PeekMessage	Check if a client message is available.
763	ReadMessage	Read message from the queue
762	StartServer	Start the server
763	StopServer	Stop the server

### 33.12.3 Property overview

Page	Properties	Access	Description
765	Global	rw	Is the server reachable to all users or not
765	InstanceID	r	Instance ID
767	MaxAction	rw	Action to take when the number of messages will exceed <code>MaxQueue</code> .
767	MaxQueue	rw	Maximum number of messages in the queue, 0 for unlimited
764	Message	r	Last read message
764	MsgData	r	Last message data
764	MsgType	r	Last message type
766	OnMessage	rw	Event triggered when a message arrives
766	OnMessageError	rw	Event called when a new message has arrived, and the queue is full
766	OnMessageQueued	rw	Event called when a new message has arrived
766	OnThreadError	rw	Triggered when a thread reports an error
764	StringMessage	r	Last message as a string.
768	SynchronizeEvents	rw	Should events be run in the main thread ?
767	Threaded	rw	Is the server running threaded or not ?
765	ThreadError	r	Last thread error
765	ThreadExecuting	r	Is the message thread currently executing ?
768	ThreadTimeout	rw	Timeout waiting for message

### 33.12.4 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: Destroy ([762](#))

### 33.12.5 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: Create ([762](#))

### 33.12.6 TSimpleIPCServer.StartServer

Synopsis: Start the server

Declaration: procedure StartServer  
 procedure StartServer(AThreaded: Boolean)

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 ([750](#)) descendent) and activates the communication channel.

The aThreaded property can be used to force or disable threaded mode: in threaded mode, a thread is started that automatically checks for new messages and puts them on a queue. If the argument is not specified, then the property TSimpleIPCServer.Threaded ([767](#)) is examined to know whether to start in threaded mode or not.

After this method was called, clients can connect and send messages.

Prior to calling this method, the ServerID ([757](#)) property must be set.

Errors: If an error occurs a EIPCErrror ([747](#)) exception may be raised.

See also: TIPCServerComm ([750](#)), Active ([757](#)), ServerID ([757](#)), StopServer ([763](#)), TSimpleIPCServer.Threaded ([767](#))

### 33.12.7 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` (750) descendent).

See also: `TIPCServerComm` (750), `Active` (757), `ServerID` (757), `StartServer` (762)

### 33.12.8 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` (763).

See also: `ReadMessage` (763)

### 33.12.9 TSimpleIPCServer.ReadMessage

Synopsis: Read message from the queue

Declaration: `function ReadMessage : Boolean`

Visibility: `public`

Description: `ReadMessage` will read the oldest message from the queue, and make it available in `TSimpleIPCServer.MsgType` (764) and `TSimpleIPCServer.MsgData` (764)

It is safe to call this even if a watch thread is started.

See also: `TSimpleIPCServer.MsgType` (764), `TSimpleIPCServer.MsgData` (764)

### 33.12.10 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` (764) 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 successful call to `ReadMessage` (763). It will also not clear the data buffer.

See also: `StringMessage` (764), `MsgData` (764), `MsgType` (764)

### 33.12.11 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 successful call to `ReadMessage` (763).

See also: `GetMessageData` (763)

### 33.12.12 TSimpleIPCServer.Message

Synopsis: Last read message

Declaration: `Property Message : TIPCServerMsg`

Visibility: `public`

Access: `Read`

Description: `Message` is the last read message (using `TSimpleIPCServer.ReadMessage` (763)) from the message queue.

See also: `TSimpleIPCServer.ReadMessage` (763)

### 33.12.13 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 successful call to `ReadMessage` (763).

See also: `ReadMessage` (763)

### 33.12.14 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` (764) is better suited to read the data.

This property will contain valid data only after a successful call to `ReadMessage` (763).

See also: `StringMessage` (764), `ReadMessage` (763)

### 33.12.15 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` (761) property to form the unique server endpoint which a client should use.

See also: `ServerID` (761), `Global` (761)

### 33.12.16 TSimpleIPCServer.ThreadExecuting

Synopsis: Is the message thread currently executing ?

Declaration: `Property ThreadExecuting : Boolean`

Visibility: public

Access: Read

Description: `ThreadExecuting` is true if the server is currently running a message loop in a thread and the thread is in an executing state.

See also: `TSimpleIPCServer.ThreadError` (765), `TSimpleIPCServer.StartServer` (762), `TSimpleIPCServer.OnThreadError` (766), `TSimpleIPCServer.Threaded` (767)

### 33.12.17 TSimpleIPCServer.ThreadError

Synopsis: Last thread error

Declaration: `Property ThreadError : string`

Visibility: public

Access: Read

Description: `ThreadError` is the last error reported by the thread (or none if no error was caught).

See also: `TSimpleIPCServer.ThreadExecuting` (765), `TSimpleIPCServer.StartServer` (762), `TSimpleIPCServer.OnThreadError` (766), `TSimpleIPCServer.Threaded` (767)

### 33.12.18 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: `ServerID` (761), `InstanceID` (765)

### 33.12.19 TSimpleIPCServer.OnMessage

Synopsis: Event triggered when a message arrives

Declaration: Property OnMessage : TNotifyEvent

Visibility: published

Access: Read,Write

Description: OnMessage is called by ReadMessage (763) when a message has been read. The actual message data can be retrieved with one of the StringMessage (764), MsgData (764) or MsgType (764) properties.

See also: StringMessage (764), MsgData (764), MsgType (764)

### 33.12.20 TSimpleIPCServer.OnMessageQueued

Synopsis: Event called when a new message has arrived

Declaration: Property OnMessageQueued : TNotifyEvent

Visibility: published

Access: Read,Write

Description: OnMessageQueued is an event handler that is called whenever a new message is pushed on the queue.

See also: TSimpleIPCServer.PeekMessage (763), TSimpleIPCServer.OnMessageError (766)

### 33.12.21 TSimpleIPCServer.OnMessageError

Synopsis: Event called when a new message has arrived, and the queue is full

Declaration: Property OnMessageError : TMessageQueueEvent

Visibility: published

Access: Read,Write

Description: OnMessageError is called whenever the message queue is full and a new message arrives on the server, and maxaction (744) is ipcmoaError.

### 33.12.22 TSimpleIPCServer.OnThreadError

Synopsis: Triggered when a thread reports an error

Declaration: Property OnThreadError : TNotifyEvent

Visibility: published

Access: Read,Write

Description: OnThreadError is triggered when the server thread reports an error. The actual error message can be examined in TSimpleIPCServer.ThreadError (765)

See also: TSimpleIPCServer.ThreadExecuting (765), TSimpleIPCServer.StartServer (762), TSimpleIPCServer.OnThreadError (766), TSimpleIPCServer.ThreadError (765), TSimpleIPCServer.Threaded (767)

### 33.12.23 TSimpleIPCServer.MaxQueue

Synopsis: Maximum number of messages in the queue, 0 for unlimited

Declaration: `Property MaxQueue : Integer`

Visibility: published

Access: Read,Write

Description: `MaxQueue` is the maximum number of messages in the queue. When this amount is zero, the amount of messages is unlimited.

When a new message is pushed, and the `Count` (756) is equal to `MaxQueue`, the `MaxAction` (767) property is examined to know what to do.

See also: `MaxAction` (767)

### 33.12.24 TSimpleIPCServer.MaxAction

Synopsis: Action to take when the number of messages will exceed `MaxQueue`.

Declaration: `Property MaxAction : TIPCMessagesOverflowAction`

Visibility: published

Access: Read,Write

Description: `MaxAction` determines what will happen if the number of messages on the queue equals `MaxQueue` (767) and a new message is put in the queue during `PeekMessage` (763):

Do nothing, just add the message

Discard the oldest message

Discard the new message

See also: `TSimpleIPCServer.MaxQueue` (767), `TSimpleIPCServer.PeekMessage` (763), `TIPCMessagesOverflowAction` (746)

### 33.12.25 TSimpleIPCServer.Threaded

Synopsis: Is the server running threaded or not ?

Declaration: `Property Threaded : Boolean`

Visibility: published

Access: Read,Write

Description: `Threaded` indicates whether the server was started in threaded mode or not. It can be set before calling `StartServer` (762). Trying to set it when the server is started will result in an error.

See also: `StartServer` (762)



### 33.12.26 TSimpleIPCServer.ThreadTimeout

Synopsis: Timeout waiting for message

Declaration: `Property ThreadTimeout : Integer`

Visibility: published

Access: Read,Write

Description: `ThreadTimeOut` is the time the thread will wait for messages between loop iterations, if the server is started with threading enabled.

When stopping the server, this is also the maximum time the server will be blocked when stopping, because it needs to wait for the thread to stop.

See also: `TSimpleIPCServer.StartServer` ([762](#)), `TSimpleIPCServer.StopServer` ([763](#))

### 33.12.27 TSimpleIPCServer.SynchronizeEvents

Synopsis: Should events be run in the main thread ?

Declaration: `Property SynchronizeEvents : Boolean`

Visibility: published

Access: Read,Write

Description: `SynchronizeEvents` can be set to `True` to force execution of events in the main thread, when the server is running in threaded mode. If set to `False`, the events will be triggered in the thread responsible for checking messages. It is ignored when the server is not running threaded. It cannot be set when the server is already started.

See also: `TSimpleIPCServer.Threaded` ([767](#)), `StartServer` ([762](#))

## Chapter 34

# Reference for unit 'SQLDB'

### 34.1 Used units

Table 34.1: Used units by unit 'SQLDB'

Name	Page
BufDataset	<a href="#">130</a>
Classes	??
DB	<a href="#">316</a>
sqlscript	??
SQLTypes	<a href="#">842</a>
sysutils	??

### 34.2 Overview

The SQLDB unit defines four main classes to handle data in SQL based databases.

1. TSQLConnection ([796](#)) represents the connection to the database. Here, properties pertaining to the connection (machine, database, user password) must be set. This is an abstract class, which should not be used directly. Per database type (mysql, firebird, postgres, oracle, sqlite) a descendent should be made and used.
2. TSQLQuery ([810](#)) is a `#fcl.db.TDataset` ([371](#)) descendent which can be used to view and manipulate the result of an SQL select query. It can also be used to execute all kinds of SQL statements.
3. TSQLTransaction ([837](#)) represents the transaction in which an SQL command is running. SQLDB supports multiple simultaneous transactions in a database connection. For databases that do not support this functionality natively, it is simulated by maintaining multiple connections to the database.
4. TSQLScript ([825](#)) can be used when many SQL commands must be executed on a database, for example when creating a database.

There is also a unified way to retrieve schema information, and a registration for connector types. More information on how to use these components can be found in UsingSQLDB ([770](#)).

### 34.3 Using SQLDB to access databases

SQLDB can be used to connect to any SQL capable database. It allows to execute SQL statements on any supported database type in a uniform way, and allows to fetch and manipulate result sets (such as returned by a `SELECT` statement) using a standard `TDataset` (371) interface. SQLDB takes care that updates to the database are posted automatically to the database, in a cached manner.

When using SQLDB, 3 components are always needed:

1. A `TSQLConnection` (796) descendent. This represents the connection to the database: the location of the database, and the username and password to authenticate the connection must be specified here. For each supported database type (Firebird, PostgreSQL, MySQL) there is a separate connection component. They all descend from `TSQLConnection`.
2. A `TSQLTransaction` (837) component. SQLDB allows you to have multiple active but independent transactions in your application. (useful for instance in middle-tier applications). If the native database client library does not support this directly, it is emulated using multiple connections to the database.
3. A `TSQLQuery` (810) component. This encapsulates an SQL statement. Any kind of SQL statement can be executed. The `TSQLQuery` component is a `TDataset` descendent: If the statement returns a result set, then it can be manipulated using the usual `TDataset` mechanisms.

The 3 components must be linked together: the connection must point to a default transaction (it is used to execute certain queries for metadata), the transaction component must point to a connection component. The `TSQLQuery` component must point to both a transaction and a database.

So in order to view the contents of a table, typically the procedure goes like this:

```
{ $mode objfpc } { $h+ }
uses sqlldb, ibconnection;

Var
  C : TSQLConnection;
  T : TSQLTransaction;
  Q : TSQLQuery;

begin
  // Create a connection.
  C:=TIBConnection.Create( Nil );
  try
    // Set credentials.
    C.UserName:='MyUSER';
    C.Password:='Secret';
    C.DatabaseName:='/home/firebird/events.fb';
    // Create a transaction.
    T:=TSQLTransaction.Create( C );
    // Point to the database instance
    T.Database:=C;
    // Now we can open the database.
    C.Connected:=True;
    // Create a query to return data
    Q:=TSQLQuery.Create( C );
    // Point to database and transaction.
```

```

Q.Database:=C;
Q.Transaction:=T;
// Set the SQL select statement
Q.SQL.Text:='SELECT * FROM USERS';
// And now use the standard TDataset methods.
Q.Open;
While not Q.EOF do
begin
  Writeln(Q.FieldName('U_NAME').AsString);
  Q.Next;
end;
Q.Close;
finally
  C.Free;
end;
end.

```

The above code is quite simple. The connection type is `TIBConnection`, which is used for Firebird/Interbase databases. To connect to another database (for instance PostgreSQL), the exact same code could be used, but instead of a `TIBConnection`, a `TPQConnection` component must be used:

```

{$mode objfpc}{$h+}
uses sqldb, pqconnection;

Var
  C : TSQLConnection;
  T : TSQLTransaction;
  Q : TSQLQuery;

begin
  // Create a connection.
  C:=TPQConnection.Create( Nil );

```

The rest of the code remains identical.

The above code used an SQL `SELECT` statement and the `Open` method to fetch data from the database. Almost the same method applies when trying to execute other kinds of queries, such as DDL queries:

```

{$mode objfpc}{$h+}
uses sqldb, ibconnection;

Var
  C : TSQLConnection;
  T : TSQLTransaction;
  Q : TSQLQuery;

begin
  C:=TIBConnection.Create( Nil );
  try
    C.UserName:='MyUSER';
    C.Password:='Secret';
    C.DatabaseName:='/home/firebird/events.fb';

```

```

T:=TSQLTransaction.Create(C);
T.Database:=C;
C.Connected:=True;
Q:=TSQLQuery.Create(C);
Q.Database:=C;
Q.Transaction:=T;
// Set the SQL statement. SQL is a tstrings instance.
With Q.SQL do
begin
Add('CREATE TABLE USERS ( ');
Add(' U_NAME VARCHAR(50), ');
Add(' U_PASSWORD VARCHAR(50) ');
Add(' ) ');
end;
// And now execute the query using ExecSQL
// There is no result, so Open cannot be used.
Q.ExecSQL;
// Commit the transaction.
T.Commit;
finally
C.Free;
end;
end.

```

As can be seen from the above example, the setup is the same as in the case of fetching data. Note that `TSQLQuery` (810) can only execute 1 SQL statement during `ExecSQL`. If many SQL statements must be executed, `TSQLScript` (825) must be used.

There is much more to `TSQLQuery` than explained here: it can use parameters (see `UsingParams` (774)) and it can automatically update the data that you edit in it (see `UpdateSQLs` (773)).

See also: `TSQLConnection` (796), `TSQLTransaction` (837), `TSQLQuery` (810), `TSQLConnector` (807), `TSQLScript` (825), `UsingParams` (774), `UpdateSQLs` (773)

## 34.4 Using the universal `TSQLConnector` type

The normal procedure when using `SQLDB` is to use one of the `TSQLConnection` (796) descendent components. When the database backend changes, another descendent of `TSQLConnection` must be used. When using a lot of different connection types and components, this may be confusing and a lot of work.

There is a universal connector component `TSQLConnector` (807) which can connect to any database supported by `SQLDB`: it works as a proxy. Behind the scenes it uses a normal `TSQLConnection` descendent to do the real work. All this happens transparently to the user code, the universal connector acts and works like any normal connection component.

The type of database can be set in its `ConnectorType` (808) property. By setting the `ConnectorType` property, the connector knows which `TSQLConnection` descendent must be created.

Each `TSQLConnection` descendent registers itself with a unique name in the initialization section of the unit implementing it: this is the name that should be specified in the `ConnectorType` of the universal connection. The list of available connections can be retrieved with the `GetConnectionList` (783) call.

From this mechanism it follows that before a particular connection type can be used, its definition must be present in the list of connector types. This means that the unit of the connection type

(`ibconnection`, `pqconnection` etc.) must be included in the `uses` clause of the program file: if it is not included, the connection type will not be registered, and it will not be available for use in the universal connector.

The universal connector only exposes the properties common to all connection types (the ones in `TSQLConnection`). It does not expose properties for all the properties available in specific `TSQLConnection` descendents. This means that if connection-specific options must be used, they must be included in the `Params` (807) property of the universal connector in the form `Name=Value`. When the actual connection instance is created, the connection-specific properties will be set from the specified parameters.

See also: `TSQLConnection` (796), `TSQLConnector` (807)

## 34.5 Retrieving Schema Information

Schema Information (lists of available database objects) can be retrieved using some specialized calls in `TSQLConnection` (796):

- `TSQLConnection.GetTableNames` (799) retrieves a list of available tables. The system tables can be requested.
- `TSQLConnection.GetProcedureNames` (799) retrieves a list of available stored procedures.
- `TSQLConnection.GetFieldNames` (799) retrieves a list of fields for a given table.

These calls are pretty straightforward and need little explanation. A more versatile system is the schema info query: the `TCustomSQLQuery.SetSchemaInfo` (790) method can be used to create a result set (dataset) with schema information. The parameter `SchemaType` determines the resulting information when the dataset is opened. The following information can be requested:

**stTables** Retrieves the list of user Tables in database. This is used internally by `TSQLConnection.GetTableNames` (799).

**stSysTables** Retrieves the list of system Tables in database. This is used internally by `TSQLConnection.GetTableNames` (799) when the system tables are requested

**stProcedures** Retrieves a list of stored procedures in database. This is used internally by `TSQLConnection.GetProcedureNames` (799).

**stColumns** Retrieves the list of columns (fields) in a table. This is used internally by `TSQLConnection.GetFieldNames` (799).

**stProcedureParams** This retrieves the parameters for a stored procedure.

**stIndexes** Retrieves the indexes for one or more tables. (currently not implemented)

**stPackages** Retrieves packages for databases that support them. (currently not implemented).

## 34.6 Automatic generation of update SQL statements

SQLDB (more in particular, `TSQLQuery` (810)) can automatically generate update statements for the data it fetches. To this end, it will scan the SQL statement and determine the main table in the query: this is the first table encountered in the `FROM` part of the `SELECT` statement.

For INSERT and UPDATE operations, the SQL statement will update/insert all fields that have `pfInUpdate` in their `ProviderFlags` property. Read-only fields will not be added to the SQL statement. Fields that are NULL will not be added to an insert query, which means that the database server will insert whatever is in the `DEFAULT` clause of the corresponding field definition.

The WHERE clause for update and delete statements consists of all fields with `pfInKey` in their `ProviderFlags` property. Depending on the value of the `UpdateMode` (822) property, additional fields may be added to the WHERE clause:

**upWhereKeyOnly** No additional fields are added: only fields marked with `pfInKey` are used in the WHERE clause

**upWhereChanged** All fields whose value changed are added to the WHERE clause, using their old value.

**upWhereAll** All fields are added to the WHERE clause, using their old value.

In order to let SQLDB generate correct statements, it is important to set the `ProviderFlags` (445) properties correct for all fields.

In many cases, for example when only a single table is queried, and no AS field aliases are used, setting `TSQLQuery.UsePrimaryKeyAsKey` (823) combined with `UpdateMode` equal to `upWhereKeyOnly` is sufficient.

If the automatically generated queries are not correct, it is possible to specify the SQL statements to be used in the `UpdateSQL` (819), `InsertSQL` (819) and `DeleteSQL` (820) properties. The new field values should be specified using params with the same name as the field. The old field values should be specified using the `OLD_` prefix to the field name. The following example demonstrates this:

```
INSERT INTO MYTABLE
  (MYFIELD,MYFIELD2)
VALUES
  (:MYFIELD, :MYFIELD2);

UPDATE MYTABLE SET
  MYFIELD=:MYFIELD
  MYFIELD2=:MYFIELD2
WHERE
  (MYFIELD=:OLD_MYFIELD);

DELETE FROM MYTABLE WHERE (MyField=:OLD_MYFIELD);
```

See also: `UsingParams` (774), `TSQLQuery` (810), `UpdateSQL` (819), `InsertSQL` (819), `DeleteSQL` (819)

## 34.7 Using parameters

SQLDB implements parameterized queries, simulating them if the native SQL client does not support parameterized queries. A parameterized query means that the SQL statement contains placeholders for actual values. The following is a typical example:

```
SELECT * FROM MyTable WHERE (id=:id)
```

The `:id` is a parameter with the name `id`. It does not contain a value yet. The value of the parameter will be specified separately. In SQLDB this happens through the `TParams` collection, where each element of the collection is a named parameter, specified in the SQL statement. The value can be specified as follows:

```
Params.ParamByname('id').AsInteger:=123;
```

This will tell SQLDB that the parameter `id` is of type integer, and has value 123.

SQLDB uses parameters for 3 purposes:

1. When executing a query multiple times, simply with different values, this helps increase the speed if the server supports parameterized queries: the query must be prepared only once.
2. Master-Detail relationships between datasets can be established based on a parameterized detail query: the value of the parameters in the detail query is automatically obtained from fields with the same names in the master dataset. As the user scrolls through the master dataset, the detail dataset is refreshed with the new values of the params.
3. Updating of data in the database happens through parameterized update/delete/insert statements: the `TSQLQuery.UpdateSQL` (819), `TSQLQuery.DeleteSQL` (820), `TSQLQuery.InsertSQL` (819) properties of `TSQLQuery` (810) must contain parameterized queries.

An additional advantage of using parameters is that they help to avoid SQL injection: by specifying a parameter type and value, SQLDB will automatically check whether the value is of the correct type, and will apply proper quoting when the native engine does not support parameters directly.

See also: `TSQLQuery.Params` (821), `UpdateSQLs` (773)

## 34.8 Constants, types and variables

### 34.8.1 Constants

```
DefaultSQLFormatSettings : TFormatSettings = (CurrencyFormat: 1; NegCurrFormat
: 5; ThousandSeparator: #0; DecimalSeparator: '.'; CurrencyDecimals
: 2; DateSeparator: '-'; TimeSeparator: ':'; ListSeparator: ' '; CurrencyString
: '$'; ShortDateFormat: 'yyyy-mm-dd'; LongDateFormat: ''; TimeAMString
: ''; TimePMString: ''; ShortTimeFormat: 'hh:nn:ss'; LongTimeFormat
: 'hh:nn:ss.zzz'; ShortMonthNames: ('', '', '', '', '', '', '', '', ''
, '', '', '', ''); LongMonthNames: ('', '', '', '', '', '', '', '', ''
, '', '', '', ''); ShortDayNames: ('', '', '', '', '', '', '', ''); LongDayNames
: ('', '', '', '', '', '', '', '', '' ); TwoDigitYearCenturyWindow: 50)
```

`DefaultSQLFormatSettings` contains the default settings used when formatting date/time and other special values in Update SQL statements generated by the various `TSQLConnection` (796) descendents.

```
detActualSQL = sqltypes.detActualSQL
```

Alias for `sqltypes.detActualSQL`

```
detCommit = sqltypes.detCommit
```

Alias for `sqltypes.detCommit`

```
detCustom = sqltypes.detCustom
```

Alias for `sqltypes.detCustom`



```
detExecute = sqltypes.detExecute
```

Alias for `sqltypes.detExecute`

```
detFetch = sqltypes.detFetch
```

Alias for `sqltypes.detFetch`

```
detParamValue = sqltypes.detParamValue
```

Alias for `sqltypes.detParamValue`

```
detPrepare = sqltypes.detPrepare
```

Alias for `sqltypes.detPrepare`

```
detRollBack = sqltypes.detRollBack
```

Alias for `sqltypes.detRollBack`

```
DoubleQuotes : TQuoteChars = ('"', '"')
```

`DoubleQuotes` is the set of delimiters used when using double quotes for string literals.

```
LogAllEvents = [detCustom, detPrepare, detExecute, detFetch, detCommit
, detRollBack]
```

`LogAllEvents` is a constant that contains the full set of available event types. It can be used to set `TSQLConnection.LogEvents` ([804](#)).

```
LogAllEventsExtra = [detCustom, detPrepare, detExecute, detFetch,
detCommit, detRollBack, detParamValue, detActualSQL]
```

`LogAllEventsExtra` lists all possible even types that can be reported using the connection logging mechanism.

```
SingleQuotes : TQuoteChars = (''', ''')
```

`SingleQuotes` is the set of delimiters used when using single quotes for string literals.

```
StatementTokens : Array[TStatementType] of string = ('(unknown)',
'select', 'insert', 'update', 'delete', 'create', 'get', 'put', 'execute'
, 'start', 'commit', 'rollback', '?')
```

`StatementTokens` contains an array of string tokens that are used to detect the type of statement, usually the first SQL keyword of the token. The presence of this token in the SQL statement determines the kind of token.

```
stColumns = sqltypes.stColumns
```

Alias for `sqltypes.stColumns`

`stCommit = sqltypes.stCommit`

Alias for `sqltypes.stCommit`

`stDDL = sqltypes.stDDL`

Alias for `sqltypes.stDDL`

`stDelete = sqltypes.stDelete`

Alias for `sqltypes.stDelete`

`stExecProcedure = sqltypes.stExecProcedure`

Alias for `sqltypes.stExecProcedure`

`stGetSegment = sqltypes.stGetSegment`

Alias for `sqltypes.stGetSegment`

`stIndexes = sqltypes.stIndexes`

Alias for `sqltypes.stIndexes`

`stInsert = sqltypes.stInsert`

Alias for `sqltypes.stInsert`

`stNoSchema = sqltypes.stNoSchema`

Alias for `sqltypes.stUnknown`

`stPackages = sqltypes.stPackages`

Alias for `sqltypes.stPackages`

`stProcedureParams = sqltypes.stProcedureParams`

Alias for `sqltypes.stProcedureParams`

`stProcedures = sqltypes.stProcedures`

Alias for `sqltypes.stProcedures`

`stPutSegment = sqltypes.stPutSegment`

Alias for `sqltypes.stPutSegment`

`stRollback = sqltypes.stRollback`

Alias for `sqltypes.stRollback`

```
stSchemata = sqltypes.stSchemata
```

stSchemata is a convenience alias for sqlTypes.stSchemata (769)

```
stSelect = sqltypes.stSelect
```

Alias for sqltypes.stSelect

```
stSelectForUpd = sqltypes.stSelectForUpd
```

Alias for sqltypes.stSelectForUpd

```
stSequences = sqltypes.stSequences
```

Alias for sqltypes.stSequences

```
stStartTrans = sqltypes.stStartTrans
```

Alias for sqltypes.stStartTrans

```
stSysTables = sqltypes.stSysTables
```

Alias for sqltypes.stSysTables

```
stTables = sqltypes.stTables
```

Alias for sqltypes.stTables

```
stUnknown = sqltypes.stUnknown
```

Alias for sqltypes.stUnknown

```
stUpdate = sqltypes.stUpdate
```

Alias for sqltypes.stUpdate

```
TSchemaObjectNames : Array[TSchemaType] of string = ('???' , 'table_name'
, '???' , 'procedure_name' , 'column_name' , 'param_name' , 'index_name'
, 'package_name' , 'schema_name' , 'sequence')
```

Names of the various types of objects

### 34.8.2 Types

```
TCommitRollbackAction = (caNone, caCommit, caCommitRetaining, caRollback
,
caRollbackRetaining)
```

Table 34.2: Enumeration values for type TCommitRollbackAction

Value	Explanation
caCommit	Commit transaction
caCommitRetaining	Commit transaction, retaining transaction context
caNone	Do nothing
caRollback	Rollback transaction
caRollbackRetaining	Rollback transaction, retaining transaction context

TCommitRollbackAction is currently unused in SQLDB.

TConnectionDefClass = Class of TConnectionDef

TConnectionDefClass is used in the RegisterConnection (783) call to register a new TConnectionDef (784) instance.

```
TConnInfoType = (citAll, citServerType, citServerVersion,
  citServerVersionString, citClientName, citClientVersion)
```

Table 34.3: Enumeration values for type TConnInfoType

Value	Explanation
citAll	All connection information
citClientName	Client library name
citClientVersion	Client library version
citServerType	Server type description
citServerVersion	Server version as an integer number
citServerVersionString	Server version as a string

Connection information to be retrieved

```
TConnOption = (sqSupportParams, sqSupportEmptyDatabaseName, sqEscapeSlash
  ,
  sqEscapeRepeat, sqImplicitTransaction, sqLastInsertID
  ,
  sqSupportReturning, sqSequences)
```

Table 34.4: Enumeration values for type TConnOption

Value	Explanation
sqEscapeRepeat	Escapes in string literals are done by repeating the character.
sqEscapeSlash	Escapes in string literals are done with backslash characters.
sqImplicitTransaction	Does the connection support implicit transaction management
sqLastInsertID	Does the connection support getting the ID for the last insert operation.
sqSequences	
sqSupportEmptyDatabaseName	Does the connection allow empty database names ?
sqSupportParams	The connection type has native support for parameters.
sqSupportReturning	The connection type supports INSERT/UPDATE with RETURNING clause

This type describes some of the option that a particular connection type supports.

`TConnOptions = Set of TConnOption`

`TConnOptions` describes the full set of options defined by a database.

`TDBEventType = sqltypes.TDBEventType`

`TDBEventType` describes the type of a database event message as generated by `TSQLConnection` (796) through the `TSQLConnection.OnLog` (804) event.

`TDBEventTypes = sqltypes.TDBEventTypes`

`TDBEventTypes` is a set of `TDBEventType` (780) values, which is used to filter the set of event messages that should be sent. The `TSQLConnection.LogEvents` (804) property determines which events a particular connection will send.

```
TDBLogNotifyEvent = procedure(Sender: TSQLConnection;
    EventType: TDBEventType;
    const Msg: string) of object
```

`TDBLogNotifyEvent` is the prototype for the `TSQLConnection.OnLog` (804) event handler and for the global `GlobalDBLogHook` (782) event handling hook. `Sender` will contain the `TSQLConnection` (796) instance that caused the event, `EventType` will contain the event type, and `Msg` will contain the actual message: the content depends on the type of the message.

`TLibraryLoadFunction = function(const S: AnsiString) : Integer`

`TLibraryLoadFunction` is the function prototype for dynamically loading a library when the universal connection component is used. It receives the name of the library to load (`S`), and should return `True` if the library was successfully loaded. It is used in the connection definition.

`TLibraryUnloadFunction = procedure`

`TLibraryUnloadFunction` is the function prototype for dynamically unloading a library when the universal connection component is used. It has no parameters, and should simply unload the library loaded with `TLibraryLoadFunction` (780)

`TQuoteChars = sqltypes.TQuoteChars`

`TQuoteChars` is an array of characters that describes the used delimiters for string values.

`TRowCount = LargeInt`

A type to contain a result row count.

`TSchemaType = sqltypes.TSchemaType`

`TSchemaType` describes which schema information to retrieve in the `TCustomSQLQuery.SetSchemaInfo` (790) call. Depending on its value, the result set of the dataset will have different fields, describing the requested schema data. The result data will always have the same structure.

```
TSQLConnectionClass = Class of TSQLConnection
```

TSQLConnectionClass is used when registering a new connection type for use in the universal connector TSQLConnector.ConnectorType (808)

```
TSQLConnectionOption = (scoExplicitConnect,
    scoApplyUpdatesChecksRowsAffected)
```

Table 34.5: Enumeration values for type TSQLConnectionOption

Value	Explanation
scoApplyUpdatesChecksRowsAffected	ApplyUpdates will check that the RowsAffected is 1 after an update.
scoExplicitConnect	Require explicit connection to the database (default is implicit)

TSQLConnectionOption enumerates several options that can be set for TSQLConnection (796) instances using TSQLConnection.Options (805)

```
TSQLConnectionOptions = Set of TSQLConnectionOption
```

Set of TSQLConnectionOption

```
TSQLQueryOption = (sqoKeepOpenOnCommit, sqoAutoApplyUpdates,
    sqoAutoCommit, sqoCancelUpdatesOnRefresh,
    sqoRefreshUsingSelect)
```

Table 34.6: Enumeration values for type TSQLQueryOption

Value	Explanation
sqoAutoApplyUpdates	Call ApplyUpdates on Post or Delete
sqoAutoCommit	Call commit after every ApplyUpdates or ExecSQL
sqoCancelUpdatesOnRefresh	Cancel any pending updates when refresh is called
sqoKeepOpenOnCommit	Keep the dataset open after the query was committed (will fetch all records).
sqoRefreshUsingSelect	Force a refresh using the provided select instead of using RETURNING clause

TSQLQueryOption enumerates several options available to control the behaviour of an TSQL-Query (810) instance.

```
TSQLQueryOptions = Set of TSQLQueryOption
```

TSQLQueryOptions is the type of the TSQLQuery.Options (821) property.

```
TSQLSequenceApplyEvent = (saeOnNewRecord, saeOnPost)
```

Table 34.7: Enumeration values for type TSQLSequenceApplyEvent

Value	Explanation
saeOnNewRecord	Fetch an ID when a new record is appended to a dataset
saeOnPost	Fetch an ID when a new record is posted in the dataset

TSQLSequenceApplyEvent enumerates the moments when a new ID must be fetched for a sequence field.

```
TSQLTransactionOption = (stoUseImplicit,stoExplicitStart)
```

Table 34.8: Enumeration values for type TSQLTransactionOption

Value	Explanation
stoExplicitStart	Require explicit start of transactions by TSQLQuery
stoUseImplicit	Use implicit transaction control if the engine allows it.

TSQLTransactionOption enumerates several options that can be used to control the transaction behaviour of TSQLTransaction (837).

```
TSQLTransactionOptions = Set of TSQLTransactionOption
```

TSQLTransactionOptions is the property type of TSQLTransaction.Options (841).

```
TStatementType = sqltypes.TStatementType
```

TStatementType describes the kind of SQL statement that was entered in the SQL property of a TSQLQuery (810) component.

### 34.8.3 Variables

```
GlobalDBLogHook : TDBLogNotifyEvent
```

GlobalDBLogHook can be set in addition to local TSQLConnection.Onlog (804) event handlers. All connections will report events through this global event handler in addition to their OnLog event handlers. The global log event handler can be set only once, so when setting the handler, it is important to set up chaining: saving the previous value, and calling the old handler (if it was set) in the new handler.

## 34.9 Procedures and functions

### 34.9.1 GetConnectionDef

Synopsis: Search for a connection definition by name

Declaration: `function GetConnectionDef (ConnectorName: string) : TConnectionDef`

Visibility: default

**Description:** `GetConnectionDef` will search in the list of connection type definitions, and will return the one definition with the name that matches `ConnectorName`. The search is case insensitive.

If no definition is found, `Nil` is returned.

See also: `RegisterConnection` (783), `TConnectionDef` (784), `TConnectionDef.TypeName` (785)

### 34.9.2 GetConnectionList

**Synopsis:** Return a list of connection definition names.

**Declaration:** `procedure GetConnectionList (List: TStrings)`

**Visibility:** default

**Description:** `GetConnectionList` clears `List` and fills it with the list of currently known connection type names, as registered with `RegisterConnection` (783). The names are the names as returned by `TConnectionDef.TypeName` (785)

See also: `RegisterConnection` (783), `TConnectionDef.TypeName` (785)

### 34.9.3 RegisterConnection

**Synopsis:** Register a new connection type for use in the universal connector

**Declaration:** `procedure RegisterConnection (Def: TConnectionDefClass)`

**Visibility:** default

**Description:** `RegisterConnection` must be called with a class pointer to a `TConnectionDef` (784) descendent to register the connection type described in the `TConnectionDef` (784) descendent. The connection type is registered with the name as returned by `TConnectionDef.TypeName` (785).

The various connection types distributed by Free Pascal automatically call `RegisterConnection` from the `initialization` section of their unit, so simply including the unit with a particular connection type is enough to register it.

Connection types registered with this call can be unregistered with `UnRegisterConnection` (783).

**Errors:** if `Def` is `Nil`, access violations will occur.

See also: `TConnectionDef` (784), `UnRegisterConnection` (783)

### 34.9.4 UnRegisterConnection

**Synopsis:** Unregister a registered connection type

**Declaration:** `procedure UnRegisterConnection (Def: TConnectionDefClass)`  
`procedure UnRegisterConnection (ConnectionName: string)`

**Visibility:** default

**Description:** `UnRegisterConnection` will unregister the connection `Def`. If a connection with `ConnectionName` or with name as returned by the `TypeName` (785) method from `Def` was previously registered, it will be removed from the list of registered connection types.

**Errors:** if `Def` is `Nil`, access violations will occur.

See also: `TConnectionDef` (784), `RegisterConnection` (783)



## 34.10 TSQLStatementInfo

```
TSQLStatementInfo = record
  StatementType : TStatementType;
  TableName
  : string;
  Updateable : Boolean;
  WhereStartPos : Integer;
  WhereStopPos
  : Integer;
end
```

TSQLStatementInfo is a record used to describe an SQL statement. It is used internally by the TSQLStatement (834) and TSQLQuery (810) objects to analyse SQL statements.

It is used to be able to modify the SQL statement (for additional filtering) or to determine the table to update when applying dataset updates to the database.

## 34.11 ESQLError

### 34.11.1 Description

ESQLError is raised by SQLDB routines if the underlying engine raises an error. The error code returned by the engine is contained in ESQLError.ErrorCode (??), and an Ansi SQL compliant SQL state can be passed in ESQLError.SQLState (??)

See also: db.EDatabaseError (769), ESQLError.SQLState (??), ESQLError.ErrorCode (??)

### 34.11.2 Method overview

Page	Method	Description
784	CreateFmt	Create a new instance of ESQLError

### 34.11.3 ESQLError.CreateFmt

Synopsis: Create a new instance of ESQLError

Declaration: constructor CreateFmt(const Fmt: string; const Args: Array of const;  
Comp: TComponent; AErrorCode: Integer;  
ASQLState: string); Overload

Visibility: public

Description: CreateFmt is overloaded in ESQLError to be able to specify the ErrorCode (??) and SQLState (??).

See also: ESQLError.ErrorCode (??), ESQLError.SQLState (??)

## 34.12 TConnectionDef

### 34.12.1 Description

TConnectionDef is an abstract class. When registering a new connection type for use in the universal connector, a descendent of this class must be made and registered using RegisterConnec-

tion (783). A descendent class should override at least the `TConnectionDef.TypeName` (785) and `TConnectionDef.ConnectionClass` (785) methods to return the specific name and connection class to use.

See also: `TConnectionDef.TypeName` (785), `TConnectionDef.ConnectionClass` (785), `RegisterConnection` (783)

### 34.12.2 Method overview

Page	Method	Description
<a href="#">787</a>	<code>ApplyParams</code>	Apply parameters to an instance of <code>TSQLConnection</code>
<a href="#">785</a>	<code>ConnectionClass</code>	Class to instantiate when this connection is requested
<a href="#">786</a>	<code>DefaultLibraryName</code>	Default library name
<a href="#">786</a>	<code>Description</code>	A descriptive text for this connection type
<a href="#">787</a>	<code>LoadedLibraryName</code>	Currently loaded library.
<a href="#">786</a>	<code>LoadFunction</code>	Return a function to call when the client library must be loaded
<a href="#">785</a>	<code>TypeName</code>	Name of the connection type
<a href="#">786</a>	<code>UnLoadFunction</code>	Return a function to call when the client library must be unloaded

### 34.12.3 TConnectionDef.TypeName

Synopsis: Name of the connection type

Declaration: `class function TypeName : string; Virtual`

Visibility: default

Description: `TypeName` is overridden by descendent classes to return the unique name for this connection type. It is what the `TSQLConnector.ConnectorType` (808) property should be set to select this connection type for the universal connection, and is the name that the `GetConnectionDef` (782) call will use when looking for a connection type. It must be overridden by descendents of `TConnectionDef`.

This name is also returned in the list returned by `GetConnectionList` (783)

This name can be an arbitrary name, no restrictions on the allowed characters exist.

See also: `TSQLConnector.ConnectorType` (808), `GetConnectionDef` (782), `GetConnectionList` (783), `TConnectionDef.ConnectionClass` (785)

### 34.12.4 TConnectionDef.ConnectionClass

Synopsis: Class to instantiate when this connection is requested

Declaration: `class function ConnectionClass : TSQLConnectionClass; Virtual`

Visibility: default

Description: `ConnectionClass` should return the connection class to use when a connection of this type is requested. It must be overridden by descendents of `TConnectionDef`.

It may not be `Nil`.

See also: `TConnectionDef.TypeName` (785)

### 34.12.5 TConnectionDef.Description

Synopsis: A descriptive text for this connection type

Declaration: `class function Description : string; Virtual`

Visibility: default

Description: `Description` should return a descriptive text for this connection type. It is used for display purposes only, so ideally it should be a one-liner. It can be used to provide more information about the particulars of the connection type.

See also: `TConnectionDef.TypeName` ([785](#))

### 34.12.6 TConnectionDef.DefaultLibraryName

Synopsis: Default library name

Declaration: `class function DefaultLibraryName : string; Virtual`

Visibility: default

Description: `DefaultLibraryName` should be set to the default library name for the connection. This can be used to let SQLDB automatically load the library needed when a connection of this type is requested.

See also: `TLibraryLoadFunction` ([780](#)), `TConnectionDef` ([784](#)), `TLibraryUnLoadFunction` ([780](#))

### 34.12.7 TConnectionDef.LoadFunction

Synopsis: Return a function to call when the client library must be loaded

Declaration: `class function LoadFunction : TLibraryLoadFunction; Virtual`

Visibility: default

Description: `LoadFunction` must return the function that will be called when the client library for this connection type must be loaded. This method must be overridden by descendent classes to return a function that will correctly load the client library when a connection of this type is used.

See also: `TLibraryLoadFunction` ([780](#)), `TConnectionDef.UnLoadFunction` ([786](#)), `TConnectionDef.DefaultLibraryName` ([786](#)), `TConnectionDef.LoadedLibraryName` ([787](#))

### 34.12.8 TConnectionDef.UnLoadFunction

Synopsis: Return a function to call when the client library must be unloaded

Declaration: `class function UnLoadFunction : TLibraryUnLoadFunction; Virtual`

Visibility: default

Description: `UnLoadFunction` must return the function that will be called when the client library for this connection type must be unloaded. This method must be overridden by descendent classes to return a function that will correctly unload the client library when a connection of this type is no longer used.

See also: `TLibraryUnLoadFunction` ([780](#)), `TConnectionDef.LoadFunction` ([786](#)), `TConnectionDef.DefaultLibraryName` ([786](#)), `TConnectionDef.LoadedLibraryName` ([787](#))

### 34.12.9 TConnectionDef.LoadedLibraryName

Synopsis: Currently loaded library.

Declaration: `class function LoadedLibraryName : string; Virtual`

Visibility: default

Description: `LoadedLibraryName` must be overridden by descendents to return the filename of the currently loaded client library for this connection type. If no library is loaded, an empty string must be returned.

See also: `TLibraryLoadFunction` (780), `TLibraryUnLoadFunction` (780), `TConnectionDef.LoadFunction` (786), `TConnectionDef.UnLoadFunction` (786), `TConnectionDef.DefaultLibraryName` (786)

### 34.12.10 TConnectionDef.ApplyParams

Synopsis: Apply parameters to an instance of `TSQLConnection`

Declaration: `procedure ApplyParams(Params: TStrings; AConnection: TSQLConnection); Virtual`

Visibility: default

Description: `ApplyParams` must be overridden to apply any params specified in the `Params` argument to the `TSQLConnection` (796) descendent in `AConnection`. It can be used to convert `Name=Value` pairs to properties of the actual connection instance.

When called, `AConnection` is guaranteed to be of the same type as returned by `TConnectionDef.ConnectionClass` (785). `Params` contains the contents of the `TSQLConnection.Params` (807) property of the connector.

See also: `TSQLConnection.Params` (807)

## 34.13 TCustomSQLQuery

### 34.13.1 Description

`TCustomSQLQuery` encapsulates a SQL statement: it implements all the necessary `#fcl.db.TDataset` (371) functionality to be able to handle a result set. It can also be used to execute SQL statements that do not return data, using the `ExecSQL` (789) method.

Do not instantiate a `TCustomSQLQuery` class directly, instead use the `TSQLQuery` (810) descendent.

See also: `TSQLQuery` (810)

**34.13.2 Method overview**

Page	Method	Description
<a href="#">791</a>	ApplyUpdates	Apply updates and check result
<a href="#">788</a>	Create	Create a new instance of TCustomSQLQuery.
<a href="#">791</a>	Delete	Delete and optionally apply updates
<a href="#">788</a>	Destroy	Destroy instance of TCustomSQLQuery
<a href="#">789</a>	ExecSQL	Execute a SQL statement that does not return a result set
<a href="#">790</a>	ParamByName	Return parameter by name
<a href="#">791</a>	Post	Post pending changes and optionally apply updates
<a href="#">788</a>	Prepare	Prepare a query for execution.
<a href="#">790</a>	RowsAffected	Return the number of rows (records) affected by the last DML/DDI statement
<a href="#">790</a>	SetSchemaInfo	SetSchemaInfo prepares the dataset to retrieve schema info.
<a href="#">789</a>	UnPrepare	Unprepare a prepared query

**34.13.3 Property overview**

Page	Properties	Access	Description
<a href="#">791</a>	Prepared	r	Is the query prepared ?
<a href="#">792</a>	SQLConnection	rw	Database as TSQLConnection
<a href="#">792</a>	SQLTransaction	rw	Transaction as TSQLTransaction

**34.13.4 TCustomSQLQuery.Create**

Synopsis: Create a new instance of TCustomSQLQuery.

Declaration: constructor Create(AOwner: TComponent); Override

Visibility: public

Description: Create allocates a new instance on the heap and will allocate all resources for the SQL statement. After this it calls the inherited constructor.

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

See also: TCustomSQLQuery.Destroy ([788](#))

**34.13.5 TCustomSQLQuery.Destroy**

Synopsis: Destroy instance of TCustomSQLQuery

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy cleans up the instance, closing the dataset and freeing all allocated resources.

See also: TCustomSQLQuery.Create ([788](#))

**34.13.6 TCustomSQLQuery.Prepare**

Synopsis: Prepare a query for execution.

Declaration: procedure Prepare; Virtual

Visibility: public

**Description:** `Prepare` will prepare the SQL for execution. It will open the database connection if it was not yet open, and will start a transaction if none was started yet. It will then determine the statement type. Finally, it will pass the statement on to the database engine if it supports preparing of queries.

Strictly speaking, it is not necessary to call `prepare`, the component will prepare the statement whenever it is necessary. If a query will be executed repeatedly, it is good practice to prepare it once before starting to execute it. This will speed up execution, since resources must be allocated only once.

**Errors:** If the SQL server cannot prepare the statement, an exception will be raised.

**See also:** `TSQLQuery.StatementType` (813), `TCustomSQLQuery.UnPrepare` (789), `TCustomSQLQuery.ExecSQL` (789)

### 34.13.7 TCustomSQLQuery.UnPrepare

**Synopsis:** Unprepare a prepared query

**Declaration:** `procedure UnPrepare; Virtual`

Visibility: public

**Description:** `UnPrepare` will unprepare a prepared query. This means that server resources for this statement are deallocated. After a query was unprepared, any `ExecSQL` or `Open` command will prepare the SQL statement again.

Several actions will unprepare the statement: Setting the `TSQLQuery.SQL` (818) property, setting the `Transaction` property or setting the `Database` property will automatically call `UnPrepare`. Closing the dataset will also unprepare the query.

**Errors:** If the SQL server cannot unprepare the statement, an exception may be raised.

**See also:** `TSQLQuery.StatementType` (813), `TCustomSQLQuery.Prepare` (788), `TCustomSQLQuery.ExecSQL` (789)

### 34.13.8 TCustomSQLQuery.ExecSQL

**Synopsis:** Execute a SQL statement that does not return a result set

**Declaration:** `procedure ExecSQL; Virtual`

Visibility: public

**Description:** `ExecSQL` will execute the statement in `TSQLQuery.SQL` (818), preparing the statement if necessary. It cannot be used to get results from the database (such as returned by a `SELECT` statement): for this, the `Open` (389) method must be used.

The `SQL` property should be a single SQL command. To execute multiple SQL statements, use the `TSQLScript` (825) component instead.

If the statement is a DML statement, the number of deleted/updated/inserted rows can be determined using `TCustomSQLQuery.RowsAffected` (790).

The `Database` and `Transaction` properties must be assigned before calling `ExecSQL`. Executing an empty SQL statement is also an error.

**Errors:** If the server reports an error, an exception will be raised.

**See also:** `TCustomSQLQuery.RowsAffected` (790), `TDataset.Open` (389)

### 34.13.9 TCustomSQLQuery.SetSchemaInfo

Synopsis: SetSchemaInfo prepares the dataset to retrieve schema info.

Declaration: `procedure SetSchemaInfo (ASchemaType: TSchemaType;  
ASchemaObjectName: string;  
ASchemaPattern: string); Virtual`

Visibility: public

Description: SetSchemaInfo will prepare the dataset to retrieve schema information from the connection, and represents the schema info as a dataset.

SetSchemaInfo is used internally to prepare a query to retrieve schema information from a connection. It will store the 3 passed parameters, which are then used in the ParseSQL and Prepare stages to optimize the allocated resources. setting the schema type to anything other than stNoSchema will also set (or mimic) the SQL statement as soon as the query is prepared. For connection types that support this, the SQL statement is then set to whatever statement the database connection supports to retrieve schema information.

This is used internally by TSQLConnection.GetTableNames (799) and TSQLConnection.GetProcedureNames (799) to get the necessary schema information from the database.

See also: TSQLConnection.GetTableNames (799), TSQLConnection.GetProcedureNames (799), RetrievingSchemaInformation (773)

### 34.13.10 TCustomSQLQuery.RowsAffected

Synopsis: Return the number of rows (records) affected by the last DML/DDI statement

Declaration: `function RowsAffected : TRowsCount; Virtual`

Visibility: public

Description: RowsAffected returns the number of rows affected by the last statement executed using ExecSQL (789).

Errors: If the connection or database type does not support returning this number, -1 is returned. If the query is not connected to a database, -1 is returned.

See also: TCustomSQLQuery.ExecSQL (789), TSQLConnection (796)

### 34.13.11 TCustomSQLQuery.ParamByName

Synopsis: Return parameter by name

Declaration: `function ParamByName (const AParamName: string) : TParam`

Visibility: public

Description: ParamByName is a shortcut for Params.ParamByName (500). The 2 following pieces of code are completely equivalent:

```
Qry.ParamByName ('id') .AsInteger:=123;
```

and

```
Qry.Params.ParamByName ('id') .AsInteger:=123;
```

See also: Params.ParamByName (500), TSQLQuery.Params (821)

**34.13.12 TCustomSQLQuery.ApplyUpdates**

Synopsis: Apply updates and check result

Declaration: `procedure ApplyUpdates(MaxErrors: Integer); Override; Overload`

Visibility: public

Description: `ApplyUpdates` is overridden in `TCustomSQLQuery` (787) to check the result of the update (using `RowsAffected` (790)).

See also: `TSQLQueryOptions` (781), `TSQLQuery.Options` (821)

**34.13.13 TCustomSQLQuery.Post**

Synopsis: Post pending changes and optionally apply updates

Declaration: `procedure Post; Override`

Visibility: public

Description: `Post` is overridden from `DB.TDataset.Post` (769) to implement the auto-applyupdates mechanism: if `TSQLQuery.Options` (821) contains `sqaAutoApplyUpdates`, then `ApplyUpdates` is called as the last step of the `Post` operation.

See also: `TSQLQuery.Options` (821), `TCustomSQLQuery.ApplyUpdates` (791)

**34.13.14 TCustomSQLQuery.Delete**

Synopsis: Delete and optionally apply updates

Declaration: `procedure Delete; Override`

Visibility: public

Description: `Delete` is overridden from `DB.TDataset.Delete` (769) to implement the auto-applyupdates mechanism: if `TSQLQuery.Options` (821) contains `sqaAutoApplyUpdates`, then `ApplyUpdates` is called as the last step of the `Post` operation.

See also: `TSQLQuery.Options` (821), `TCustomSQLQuery.ApplyUpdates` (791)

**34.13.15 TCustomSQLQuery.Prepared**

Synopsis: Is the query prepared ?

Declaration: `Property Prepared : Boolean`

Visibility: public

Access: Read

Description: `Prepared` is true if `Prepare` (788) was called for this query, and an `UnPrepare` (789) was not done after that (take care: several actions call `UnPrepare` implicitly). Initially, `Prepared` will be `False`. Calling `Prepare` if the query was already prepared has no effect.

See also: `TCustomSQLQuery.Prepare` (788), `TCustomSQLQuery.UnPrepare` (789)



### 34.13.16 TCustomSQLQuery.SQLConnection

Synopsis: Database as TSQLConnection

Declaration: Property SQLConnection : TSQLConnection

Visibility: public

Access: Read,Write

Description: SQLConnection equals the Database property, but typecasted as a TSQLConnection (796) descendent.

See also: TSQLConnection (796), TCustomSQLQuery.SQLTransaction (792)

### 34.13.17 TCustomSQLQuery.SQLTransaction

Synopsis: Transaction as TSQLTransaction

Declaration: Property SQLTransaction : TSQLTransaction

Visibility: public

Access: Read,Write

Description: SQLTransaction equals the Transaction property, but typecasted as a TSQLTransaction (837) descendent.

See also: TSQLConnection (796), TCustomSQLQuery.SQLConnection (792)

## 34.14 TCustomSQLStatement

### 34.14.1 Description

TCustomSQLStatement is a light-weight object that can be used to execute SQL statements on a database. It does not support result sets, and has none of the methods that a TDataset (769) component has. It can be used to execute SQL statements on a database that update data, execute stored procedures and DDL statements etc.

The TCustomSQLStatement is equivalent to TSQLQuery (810) in that it supports transactions (in the Transaction (803) property) and parameters (in the Params (807) property) and as such is a more versatile tool than executing queries using TSQLConnection.ExecuteDirect (798).

To use a TCustomSQLStatement is simple and similar to the use of TSQLQuery (810): set the Database (835) property to an existing connection component, and set the Transaction (837) property. After setting the SQL (836) property and filling Params (836), the SQL statement can be executed with the Execute (769) method.

TCustomSQLStatement is a parent class. Many of the properties are only made public (or published) in the TSQLStatement (834) class, which should be instantiated instead of the TCustomSQLStatement class.

See also: TSQLStatement (834), TDataset (769), TSQLQuery (810), TSQLStatement.Transaction (837), TSQLStatement.Params (836), Execute (769), TSQLStatement.Database (835), TSQLConnection.ExecuteDirect (798)

**34.14.2 Method overview**

Page	Method	Description
<a href="#">793</a>	Create	Create a new instance of TCustomSQLStatement
<a href="#">793</a>	Destroy	Destroy a TCustomSQLStatement instance.
<a href="#">794</a>	Execute	Execute the SQL statement.
<a href="#">794</a>	ParamByName	Find a parameter by name
<a href="#">793</a>	Prepare	Prepare the statement for execution
<a href="#">795</a>	RowsAffected	Number of rows affected by the SQL statement.
<a href="#">794</a>	Unprepare	Unprepare a previously prepared statement

**34.14.3 Property overview**

Page	Properties	Access	Description
<a href="#">795</a>	Prepared	r	Is the statement prepared or not

**34.14.4 TCustomSQLStatement.Create**

Synopsis: Create a new instance of TCustomSQLStatement

Declaration: constructor Create(AOwner: TComponent); Override

Visibility: public

Description: Create initializes a new instance of TCustomSQLStatement and sets the SQL ([836](#))Params ([836](#)), ParamCheck ([835](#)) and ParseSQL ([836](#)) to their initial values.

See also: TSQLStatement.SQL ([836](#)), TSQLStatement.Params ([836](#)), TSQLStatement.ParamCheck ([835](#)), TSQLStatement.ParseSQL ([836](#)), Destroy ([769](#))

**34.14.5 TCustomSQLStatement.Destroy**

Synopsis: Destroy a TCustomSQLStatement instance.

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy disconnects the TCustomSQLStatement instance from the transaction and database, and then frees the memory taken by the instance and its properties.

See also: TSQLStatement.Database ([835](#)), TSQLStatement.Transaction ([837](#))

**34.14.6 TCustomSQLStatement.Prepare**

Synopsis: Prepare the statement for execution

Declaration: procedure Prepare

Visibility: public

Description: Prepare prepares the SQL statement for execution. It is called automatically if Execute ([769](#)) is called and the statement was not yet prepared. Depending on the database engine, it will also allocate the necessary resources on the database server.

Errors: An exception is raised if there is no SQL ([836](#)) statement set or the Database ([835](#)) or Transaction ([837](#)) properties are empty.

See also: `TSQLStatement.SQL` (836), `TSQLStatement.Database` (835), `TSQLStatement.Transaction` (837), `Execute` (769)

### 34.14.7 TCustomSQLStatement.Execute

Synopsis: Execute the SQL statement.

Declaration: `procedure Execute`

Visibility: `public`

Description: `Execute` executes the SQL (769) statement on the database. If necessary, it will first open the connection and start a transaction, followed by a call to `Prepare`.

Errors: An exception is raised if there is no SQL (836) statement set or the Database (835) or Transaction (837) properties are empty.

If an error occurs at the database level (the SQL failed to execute properly) then an exception is raised as well.

See also: `TSQLStatement.SQL` (836), `TSQLStatement.Database` (835), `TSQLStatement.Transaction` (837)

### 34.14.8 TCustomSQLStatement.Unprepare

Synopsis: Unprepare a previously prepared statement

Declaration: `procedure Unprepare`

Visibility: `public`

Description: `Unprepare` unprepares a prepared SQL statement. It is called automatically when the SQL statement is changed. Depending on the database engine, it will also de-allocate any allocated resources on the database server. if the statement is not in a prepared state, nothing happens.

Errors: If an error occurs at the database level (the unprepare operation failed to execute properly) then an exception is raised.

See also: `TSQLStatement.SQL` (836), `TSQLStatement.Database` (835), `TSQLStatement.Transaction` (837), `Prepare` (769)

### 34.14.9 TCustomSQLStatement.ParamByName

Synopsis: Find a parameter by name

Declaration: `function ParamByName(const AParamName: string) : TParam`

Visibility: `public`

Description: `ParamByName` finds the parameter `AParamName` in the `Params` (836) property.

Errors: If no parameter with the given name is found, an exception is raised.

See also: `TSQLStatement.Params` (836), `TParams.ParamByname` (769)

### 34.14.10 TCustomSQLStatement.RowsAffected

Synopsis: Number of rows affected by the SQL statement.

Declaration: `function RowsAffected : TRowCount; Virtual`

Visibility: `public`

Description: `RowsAffected` is set to the number of affected rows after `Execute` (769) was called. Not all databases may support this.

See also: `Execute` (769)

### 34.14.11 TCustomSQLStatement.Prepared

Synopsis: Is the statement prepared or not

Declaration: `Property Prepared : Boolean`

Visibility: `public`

Access: `Read`

Description: `Prepared` equals `True` if `Prepare` (769) was called (implicitly or explicitly), it returns `False` if not. It can be set to `True` or `False` to call `Prepare` (769) or `UnPrepare` (769), respectively.

See also: `Prepare` (769), `UnPrepare` (769)

## 34.15 TServerIndexDefs

### 34.15.1 Description

`TServerIndexDefs` is a simple descendent of `TIndexDefs` (469) that implements the necessary methods to update the list of definitions using the `TSQLConnection` (796). It should not be used directly.

See also: `TSQLConnection` (796)

### 34.15.2 Method overview

Page	Method	Description
795	<code>Create</code>	Create a new instance of <code>TServerIndexDefs</code>
796	<code>Update</code>	Updates the list of indexes

### 34.15.3 TServerIndexDefs.Create

Synopsis: Create a new instance of `TServerIndexDefs`

Declaration: `constructor Create (ADataset: TDataSet); Override`

Visibility: `public`

Description: `Create` will raise an exception if `ADataset` is not a `TCustomSQLQuery` (787) descendent.

Errors: An `EDatabaseError` exception will be raised if `ADataset` is not a `TCustomSQLQuery` (787) descendent.

### 34.15.4 TServerIndexDefs.Update

Synopsis: Updates the list of indexes

Declaration: `procedure Update; Override`

Visibility: `public`

Description: `Update` updates the list of indexes, it uses the `TSQLConnection` (796) methods for this.

## 34.16 TSQLConnection

### 34.16.1 Description

`TSQLConnection` is an abstract class for making a connection to a SQL Database. This class will never be instantiated directly, for each database type a descendent class specific for this database type must be created.

Most of common properties to SQL databases are implemented in this class.

See also: `TSQLQuery` (810), `TSQLTransaction` (837)

### 34.16.2 Method overview

Page	Method	Description
797	Create	Create a new instance of <code>TSQLConnection</code>
801	CreateDB	Create a new Database on the server
797	Destroy	Destroys the instance of the connection.
801	DropDB	Procedure to drop or remove a Database
798	EndTransaction	End the Transaction associated with this connection
798	ExecuteDirect	Execute a piece of SQL code directly, using a Transaction if specified
800	GetConnectionInfo	Return some information about the connection
799	GetFieldNames	Gets a list of the field names in the specified table
801	GetNextValue	Get next value for a sequence
799	GetObjectNames	Return a collection of object names for a given type of object
799	GetProcedureNames	Gets a list of Stored Procedures in the Database
800	GetSchemaNames	Get database schema names
800	GetSequenceNames	Return a list of sequence names
800	GetStatementInfo	Get statement information
799	GetTableNames	Get a list of the tables in the specified database
798	StartTransaction	Start the Transaction associated with this Connection

**34.16.3 Property overview**

Page	Properties	Access	Description
<a href="#">803</a>	CharSet	rw	The character set to be used in this database
<a href="#">805</a>	Connected		Is a connection to the server active or not
<a href="#">802</a>	ConnOptions	r	The set of Connection options being used in the Connection
<a href="#">806</a>	DatabaseName		The name of the database to which connection is required.
<a href="#">802</a>	FieldNameQuoteChars	rw	Characters used to quote field names.
<a href="#">802</a>	Handle	r	Low level handle used by the connection.
<a href="#">804</a>	HostName	rw	The name of the host computer where the database resides
<a href="#">806</a>	KeepConnection		Attempt to keep the connection open once it is established.
<a href="#">804</a>	LogEvents	rw	Filter for events to log
<a href="#">806</a>	LoginPrompt		Should SQLDB prompt for user credentials when a connection is activated.
<a href="#">804</a>	OnLog	rw	Event handler for logging events
<a href="#">807</a>	OnLogin		Event handler for login process
<a href="#">805</a>	Options	rw	Options to observe for this connection.
<a href="#">807</a>	Params		Extra connection parameters
<a href="#">802</a>	Password	rw	Password used when authenticating on the database server
<a href="#">805</a>	Role	rw	Role in which the user is connecting to the database
<a href="#">803</a>	Transaction	rw	Default transaction to be used for this connection
<a href="#">803</a>	UserName	rw	The username for authentication on the database server

**34.16.4 TSQLConnection.Create**

Synopsis: Create a new instance of `TSQLConnection`

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: `public`

Description: `Create` initialized a new instance of `TSQLConnection` ([796](#)). After calling the inherited constructor, it will initialize the `FieldNameQuoteChars` ([802](#)) property and some other fields for internal use.

See also: `FieldNameQuoteChars` ([802](#))

**34.16.5 TSQLConnection.Destroy**

Synopsis: Destroys the instance of the connection.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` removes the connection from memory. When a connection is removed, all datasets are closed, and all transactions too.

### 34.16.6 TSQLConnection.StartTransaction

Synopsis: Start the Transaction associated with this Connection

Declaration: `procedure StartTransaction; Override`

Visibility: `public`

Description: `StartTransaction` is a convenience method which starts the default transaction (`Transaction (803)`). It is equivalent to

```
Connection.Transaction.StartTransaction
```

Errors: If no transaction is assigned, an exception will be raised.

See also: `EndTransaction (798)`

### 34.16.7 TSQLConnection.EndTransaction

Synopsis: End the Transaction associated with this connection

Declaration: `procedure EndTransaction; Override`

Visibility: `public`

Description: `StartTransaction` is a convenience method which ends the default transaction (`TSQLConnection.Transaction (803)`). It is equivalent to

```
Connection.Transaction.EndTransaction
```

Errors: If no transaction is assigned, an exception will be raised.

See also: `StartTransaction (798)`

### 34.16.8 TSQLConnection.ExecuteDirect

Synopsis: Execute a piece of SQL code directly, using a Transaction if specified

Declaration: `procedure ExecuteDirect(SQL: string); Virtual; Overload`  
`procedure ExecuteDirect(SQL: string; ATransaction: TSQLTransaction)`  
`; Virtual; Overload`

Visibility: `public`

Description: `ExecuteDirect` executes an SQL statement directly. If `ATransaction` is `Nil` then the default transaction is used, otherwise the specified transaction is used.

`ExecuteDirect` does not offer support for parameters, so only statements that do not need parsing and parameters substitution can be handled. If parameter substitution is required, use a `TSQLQuery (810)` component and its `ExecSQL (789)` method.

Errors: If no transaction is assigned, and no transaction is passed, an exception will be raised.

See also: `TSQLQuery (810)`, `ExecSQL (789)`

### 34.16.9 TSQLConnection.GetObjectNames

Synopsis: Return a collection of object names for a given type of object

Declaration: `function GetObjectNames (ASchemaType: TSchemaType;  
  AList: TSqlObjectIdentifierList) : Integer  
  ; Virtual`

Visibility: public

Description: `GetObjectNames` returns all the names of objects of a given type `ASchemaType` in the database, and returns the number of found objects as a result. The object names are placed in the collection `AList`.

The collection is not cleared.

See also: `#fcl.sqltypes.TSqlObjectIdentifierList` ([845](#))

### 34.16.10 TSQLConnection.GetTableNames

Synopsis: Get a list of the tables in the specified database

Declaration: `procedure GetTableNames (List: TStrings; SystemTables: Boolean); Virtual`  
Visibility: public

Description: `GetTableNames` will return the names of the tables in the database in `List`. If `SystemTables` is `True` then only the names of system tables will be returned.

`List` is cleared before adding the names.

**Remark** Note that the list may depend on the access rights of the user.

See also: `TSQLConnection.GetProcedureNames` ([799](#)), `TSQLConnection.GetFieldNames` ([799](#))

### 34.16.11 TSQLConnection.GetProcedureNames

Synopsis: Gets a list of Stored Procedures in the Database

Declaration: `procedure GetProcedureNames (List: TStrings); Virtual`  
Visibility: public

Description: `GetProcedureNames` will return the names of the stored procedures in the database in `List`.

`List` is cleared before adding the names.

See also: `TSQLConnection.GetTableNames` ([799](#)), `TSQLConnection.GetFieldNames` ([799](#))

### 34.16.12 TSQLConnection.GetFieldNames

Synopsis: Gets a list of the field names in the specified table

Declaration: `procedure GetFieldNames (const TableName: string; List: TStrings)  
  ; Virtual`

Visibility: public

Description: `GetFieldNames` will return the names of the fields in `TableName` in `list`

`List` is cleared before adding the names.

Errors: If a non-existing tablename is passed, no error will be raised.

See also: `TSQLConnection.GetTableNames` ([799](#)), `TSQLConnection.GetProcedureNames` ([799](#))



### 34.16.13 TSQLConnection.GetSchemaNames

Synopsis: Get database schema names

Declaration: `procedure GetSchemaNames(List: TStrings); Virtual`

Visibility: public

Description: `GetSchemaNames` returns a list of schemas defined in the database.

See also: `TSQLConnection.GetTableNames` (799), `TSQLConnection.GetProcedureNames` (799), `TSQLConnection.GetFieldNames` (799)

### 34.16.14 TSQLConnection.GetSequenceNames

Synopsis: Return a list of sequence names

Declaration: `procedure GetSequenceNames(List: TStrings); Virtual`

Visibility: public

Description: `GetSequenceNames` returns the names of all defined sequences (Generators in Firebird) in the databases in `List`, if the database engine supports them: Not all database types support sequences.

This call is a convenience call, a simple wrapper around `GetObjectNames` (769).

Errors: None.

See also: `GetObjectNames` (769)

### 34.16.15 TSQLConnection.GetConnectionInfo

Synopsis: Return some information about the connection

Declaration: `function GetConnectionInfo(InfoType: TConnInfoType) : string; Virtual`

Visibility: public

Description: `GetConnectionInfo` can be used to return some information about the connection. Which information is returned depends on the `InfoType` parameter. The information is returned as a string. If `citAll` is passed, then the result will be a comma-separated list of values, each of the values enclosed in double quotes.

See also: `TConnInfoType` (779)

### 34.16.16 TSQLConnection.GetStatementInfo

Synopsis: Get statement information

Declaration: `function GetStatementInfo(const ASQL: string) : TSQLStatementInfo; Virtual`

Visibility: public

Description: `GetStatementInfo` returns information about the ASQL SQL command. To this end, it will partially parse the statement.

See also: `TStatementInfo` (769)

**34.16.17 TSQLConnection.CreateDB**

Synopsis: Create a new Database on the server

Declaration: `procedure CreateDB; Virtual`

Visibility: `public`

Description: `CreateDB` will create a new database on the server. Whether or not this functionality is present depends on the type of the connection. The name for the new database is taken from the `TSQLConnection.DatabaseName` (806) property, the user credentials are taken from the `TSQLConnection.UserName` (803) and `TSQLConnection.Password` (802) properties.

Errors: If the connection type does not support creating a database, then an `EDatabaseError` exception is raised. Other exceptions may be raised if the operation fails, e.g. when the user does not have the necessary access rights.

See also: `TSQLConnection.DropDB` (801)

**34.16.18 TSQLConnection.DropDB**

Synopsis: Procedure to drop or remove a Database

Declaration: `procedure DropDB; Virtual`

Visibility: `public`

Description: `DropDB` does the opposite of `CreateDB` (801). It removes the database from the server. The database must be connected before this command may be used. Whether or not this functionality is present depends on the type of the connection.

Errors: If the connection type does not support creating a database, then an `EDatabaseError` exception is raised. Other exceptions may be raised if the operation fails, e.g. when the user does not have the necessary access rights.

See also: `TSQLConnection.CreateDB` (801)

**34.16.19 TSQLConnection.GetNextValue**

Synopsis: Get next value for a sequence

Declaration: `function GetNextValue(const SequenceName: string; IncrementBy: Integer)  
: Int64; Virtual`

Visibility: `public`

Description: `GetNextValue` returns the next value for the sequence `SequenceName`, incrementing the current value with `IncrementBy` (default 1).

Errors: Not all databases support sequences, in that case an SQL error will be raised.

See also: `TSQLConnection.GetSequenceNames` (800)

### 34.16.20 TSQLConnection.ConnOptions

Synopsis: The set of Connection options being used in the Connection

Declaration: `Property ConnOptions : TConnOptions`

Visibility: public

Access: Read

Description: `ConnOptions` is the set of options used by this connection component. It is normally the same value for all connections of the same type

See also: `TConnOption` ([779](#))

### 34.16.21 TSQLConnection.Handle

Synopsis: Low level handle used by the connection.

Declaration: `Property Handle : Pointer`

Visibility: public

Access: Read

Description: `Handle` represents the low-level handle that the `TSQLConnection` component has received from the client library of the database. Under normal circumstances, this property must not be used.

### 34.16.22 TSQLConnection.FieldNameQuoteChars

Synopsis: Characters used to quote field names.

Declaration: `Property FieldNameQuoteChars : TQuoteChars`

Visibility: public

Access: Read,Write

Description: `FieldNameQuoteChars` can be set to specify the characters that should be used to delimit field names in SQL statements generated by SQLDB. It is normally initialized correctly by the `TSQLConnection` ([796](#)) descendent to the default for that particular connection type.

See also: `TSQLConnection` ([796](#))

### 34.16.23 TSQLConnection.Password

Synopsis: Password used when authenticating on the database server

Declaration: `Property Password : string`

Visibility: published

Access: Read,Write

Description: `Password` is used when authenticating the user specified in `UserName` ([803](#)) when connecting to the database server

This property must be set prior to activating the connection. Changing it while the connection is active has no effect.

See also: `TSQLConnection.UserName` ([803](#)), `TSQLConnection.HostName` ([804](#))

**34.16.24 TSQLConnection.Transaction**

Synopsis: Default transaction to be used for this connection

Declaration: `Property Transaction : TSQLTransaction`

Visibility: published

Access: Read,Write

Description: `Transaction` should be set to a `TSQLTransaction` (837) instance. It is set as the default transaction when a query is connected to the database, and is used in several metadata operations such as `TSQLConnection.GetTableNames` (799)

See also: `TSQLTransaction` (837)

**34.16.25 TSQLConnection.UserName**

Synopsis: The username for authentication on the database server

Declaration: `Property UserName : string`

Visibility: published

Access: Read,Write

Description: `UserName` is used to authenticate on the database server when the connection to the database is established.

This property must be set prior to activating the connection. Changing it while the connection is active has no effect.

See also: `TSQLConnection.Password` (802), `TSQLConnection.HostName` (804), `TSQLConnection.Role` (805), `TSQLConnection.Charset` (803)

**34.16.26 TSQLConnection.CharSet**

Synopsis: The character set to be used in this database

Declaration: `Property CharSet : string`

Visibility: published

Access: Read,Write

Description: `Charset` can be used to tell the user in which character set the data will be sent to the server, and in which character set the results should be sent to the client. Some connection types will ignore this property, and the data will be sent to the client in the encoding used on the server.

This property must be set prior to activating the connection. Changing it while the connection is active has no effect.

**Remark** `SQLDB` will not do anything with this setting except pass it on to the server if a specific connection type supports it. It does not perform any conversions by itself based on the value of this setting.

See also: `TSQLConnection.Password` (802), `TSQLConnection.HostName` (804), `TSQLConnection.UserName` (803), `TSQLConnection.Role` (805)

### 34.16.27 TSQLConnection.HostName

Synopsis: The name of the host computer where the database resides

Declaration: `Property HostName : string`

Visibility: published

Access: Read,Write

Description: `HostName` is the name of the host computer where the database server is listening for connection. An empty value means the local machine is used.

This property must be set prior to activating the connection. Changing it while the connection is active has no effect.

See also: `TSQLConnection.Role` (805), `TSQLConnection.Password` (802), `TSQLConnection.UserName` (803), `TSQLConnection.DatabaseName` (806), `TSQLConnection.Charset` (803)

### 34.16.28 TSQLConnection.OnLog

Synopsis: Event handler for logging events

Declaration: `Property OnLog : TDBLogNotifyEvent`

Visibility: published

Access: Read,Write

Description: `TSQLConnection` can send events for all the actions that it performs: executing SQL statements, commit and rollback of transactions etc. This event handler must be set to react on these events: they can for example be written to a log file. Only events specified in the `LogEvents` (804) property will be logged.

The events received by this event handler are specific for this connection. To receive events from all active connections in the application, set the global `GlobalDBLogHook` (782) event handler.

See also: `GlobalDBLogHook` (782), `TSQLConnection.LogEvents` (804)

### 34.16.29 TSQLConnection.LogEvents

Synopsis: Filter for events to log

Declaration: `Property LogEvents : TDBEventTypes`

Visibility: published

Access: Read,Write

Description: `LogEvents` can be used to filter the events which should be sent to the `OnLog` (804) and `GlobalDBLogHook` (782) event handlers. Only event types that are listed in this property will be sent.

See also: `GlobalDBLogHook` (782), `TSQLConnection.OnLog` (804)

**34.16.30 TSQLConnection.Options**

Synopsis: Options to observe for this connection.

Declaration: `Property Options : TSQLConnectionOptions`

Visibility: published

Access: Read,Write

Description: `Options` can be used to control the behaviour of SQLDB for this connection. The following options can be set:

**scoExplicitConnect** When set, the connection must be explicitly made. Default behaviour is for `TSQLQuery` to implicitly open the connection as needed.

**scoApplyUpdatesChecksRowsAffected** When set, whenever an update SQL Statement is executed during `ApplyOptions` of a dataset, the `TSQLStatement.RowsAffected` (834) is checked and must be equal to 1.

See also: `TSQLQuery.ApplyUpdates` (810), `TSQLStatement.RowsAffected` (834), `TSQLQuery.RowsAffected` (810)

**34.16.31 TSQLConnection.Role**

Synopsis: Role in which the user is connecting to the database

Declaration: `Property Role : string`

Visibility: published

Access: Read,Write

Description: `Role` is used to specify the user's role when connecting to the database user. Not all connection types support roles, for those that do not, this property is ignored.

This property must be set prior to activating the connection. Changing it while the connection is active has no effect.

See also: `TSQLConnection.Password` (802), `TSQLConnection.UserName` (803), `TSQLConnection.DatabaseName` (806), `TSQLConnection.Hostname` (804)

**34.16.32 TSQLConnection.Connected**

Synopsis: Is a connection to the server active or not

Declaration: `Property Connected :`

Visibility: published

Access:

Description: `Connected` indicates whether a connection to the server is active or not. No queries to this server can be activated as long as the value is `False`

Setting the property to `True` will attempt a connection to the database `DatabaseName` (806) on host `HostName` (804) using the credentials specified in `UserName` (803) and `Password` (802). If the connection or authentication fails, an exception is raised. This has the same effect as calling `Open` (359).

Setting the property to `False` will close the connection to the database. All datasets connected to the database will be closed, all transactions will be closed as well. This has the same effect as calling `Close` (769)

See also: [TSQLConnection.Password \(802\)](#), [TSQLConnection.UserName \(803\)](#), [TSQLConnection.DatabaseName \(806\)](#), [TSQLConnection.Role \(805\)](#)

### 34.16.33 TSQLConnection.DatabaseName

Synopsis: The name of the database to which connection is required.

Declaration: `Property DatabaseName :`

Visibility: published

Access:

Description: `DatabaseName` is the name of the database to which a connection must be made. Some servers need a complete path to a file, others need a symbolic name (an alias): the interpretation of this name depends on the connection type.

This property must be set prior to activating the connection. Changing it while the connection is active has no effect.

See also: [TSQLConnection.Password \(802\)](#), [TSQLConnection.UserName \(803\)](#), [TSQLConnection.Charset \(803\)](#), [TSQLConnection.Hostname \(804\)](#)

### 34.16.34 TSQLConnection.KeepConnection

Synopsis: Attempt to keep the connection open once it is established.

Declaration: `Property KeepConnection :`

Visibility: published

Access:

Description: `KeepConnection` can be used to attempt to keep the connection open once it is established. This property is currently not implemented.

### 34.16.35 TSQLConnection.LoginPrompt

Synopsis: Should SQLDB prompt for user credentials when a connection is activated.

Declaration: `Property LoginPrompt :`

Visibility: published

Access:

Description: `LoginPrompt` can be set to `True` to force the system to get a username/password pair from the user. How these data are fetched from the user depends on the [OnLogin \(807\)](#) event handler. The [UserName \(803\)](#) and [Password \(802\)](#) properties are ignored in this case.

See also: [TSQLConnection.Password \(802\)](#), [TSQLConnection.UserName \(803\)](#), [OnLogin \(807\)](#)

### 34.16.36 TSQLConnection.Params

Synopsis: Extra connection parameters

Declaration: `Property Params :`

Visibility: published

Access:

Description: `Params` can be used to specify extra parameters to use when establishing a connection to the database. Which parameters can be specified depends on the connection type.

See also: `TSQLConnection.Password` (802), `TSQLConnection.UserName` (803), `TSQLConnection.Hostname` (804), `TSQLConnection.DatabaseName` (806)

### 34.16.37 TSQLConnection.OnLogin

Synopsis: Event handler for login process

Declaration: `Property OnLogin :`

Visibility: published

Access:

Description: `OnLogin` will be used when `loginPrompt` (806) is `True`. It will be called, and can be used to present a user with a dialog in which the username and password can be asked.

See also: `TSQLConnection.LoginPrompt` (806)

## 34.17 TSQLConnector

### 34.17.1 Description

`TSQLConnector` implements a general connection type. When switching database backends, the normal procedure is to replace one instance of `TSQLConnection` (796) descendent with another, and connect all instances of `TSQLQuery` (810) and `TSQLTransaction` (837) to the new connection.

Using `TSQLConnector` avoids this: the type of connection can be set using the `ConnectorType` (808) property, which is a string property. The `TSQLConnector` class will (in the background) create the correct `TSQLConnection` (796) descendent to handle all actual operations on the database.

In all other respects, `TSQLConnector` acts like a regular `TSQLConnection` instance. Since no access to the actually used `TSQLConnection` descendent is available, connection-specific calls are not available.

See also: `TSQLConnector.ConnectorType` (808), `UniversalConnectors` (772)

### 34.17.2 Property overview

Page	Properties	Access	Description
808	<code>ConnectorType</code>	rw	Name of the connection type to use



### 34.17.3 TSQLConnector.ConnectorType

Synopsis: Name of the connection type to use

Declaration: `Property ConnectorType : string`

Visibility: `published`

Access: `Read,Write`

Description: `ConnectorType` should be set to one of the available connector types in the application. The list of possible connector types can be retrieved using `GetConnectionList` (783) call. The `ConnectorType` property can only be set when the connection is not active.

Errors: Attempting to change the `ConnectorType` property while the connection is active will result in an exception.

See also: `GetConnectionList` (783)

## 34.18 TSQLCursor

### 34.18.1 Description

`TSQLCursor` is an abstract internal object representing a result set returned by a single SQL select statement (`TSQLHandle` (810)). statement. It is used by the `TSQLQuery` (810) component to handle result sets returned by SQL statements.

This object must not be used directly.

See also: `TSQLQuery` (810), `TSQLHandle` (810)

## 34.19 TSQLDBFieldDef

### 34.19.1 Description

`TSQLDBFieldDef` is a `SQLDB` specific `db.TFieldDef` (769) descendent which has room for storing engine-specific data for the result set fields.

See also: `TSQLDBFieldDef.SQLDBData` (808)

### 34.19.2 Property overview

Page	Properties	Access	Description
<a href="#">808</a>	<code>SQLDBData</code>	<code>rw</code>	Pointer to store engine-specific data for the result field

### 34.19.3 TSQLDBFieldDef.SQLDBData

Synopsis: Pointer to store engine-specific data for the result field

Declaration: `Property SQLDBData : Pointer`

Visibility: `public`

Access: `Read,Write`

**Description:** `SQLDBData` can be used by the `TSQLConnection` (796) descendants to store additional data about fields in a result set. It is not used by `TSQLQuery` itself.

See also: `TSQLQuery` (810)

## 34.20 TSQLDBFieldDefs

### 34.20.1 Description

`TSQLDBFieldDefs` is a `TFieldDefs` descendent which creates `TSQLDBFieldDef` (808) descendants when a new field is added to the field set, to provide storage for engine-specific field data.

See also: `TSQLDBFieldDef` (808)

## 34.21 TSQLDBParam

### 34.21.1 Description

`TSQLDBParam` is used to be able to create parameters which can store info about a field on which the parameter is based in the `TSQLDBParam.FieldDef` (809) field. This is useful when constructing update or insert queries. It can also store private data needed for the various `TSQLConnection` (796) descendants in the `TSQLDBParam.SQLDBData` (809) property.

See also: `TSQLDBFieldDef` (808), `TSQLDBParam.FieldDef` (809), `TSQLDBParam.SQLDBData` (809)

### 34.21.2 Property overview

Page	Properties	Access	Description
<a href="#">809</a>	<code>FieldDef</code>	rw	Field definition for update SQL
<a href="#">809</a>	<code>SQLDBData</code>	rw	Private data for TSQLDB descendants.

### 34.21.3 TSQLDBParam.FieldDef

**Synopsis:** Field definition for update SQL

**Declaration:** `Property FieldDef : TFieldDef`

**Visibility:** public

**Access:** Read,Write

**Description:** `FieldDef` is used during generation of update SQL statements to store information about the field to be updated.

See also: `TSQLDBFieldDef` (808), `TSQLDBParam.SQLDBData` (809)

### 34.21.4 TSQLDBParam.SQLDBData

**Synopsis:** Private data for TSQLDB descendants.

**Declaration:** `Property SQLDBData : Pointer`

**Visibility:** public

Access: Read,Write

Description: `SQLDBData` should not be used by the end-user, it is for internal use by the `TSQLConnection` (796) descendents.

See also: `TSQLConnection` (796), `SQLDBParam.FieldDef` (809)

## 34.22 TSQLDBParams

### 34.22.1 Description

`TSQLDBParams` is a `TParams` descendent which creates `TSQLDBParams` (810) descendents when a new field is added to the field set, to provide storage for engine-specific field data.

See also: `TSQLDBParam` (809)

## 34.23 TSQLHandle

### 34.23.1 Description

`TSQLHandle` is an abstract internal object representing a database client handle. It is used by the various connections to implement the connection-specific functionality, and usually represents a low-level handle. It is used by the `TSQLQuery` (810) component to communicate with the `TSQLConnection` (796) descendent.

This object must not be used directly.

See also: `TSQLQuery` (810), `TSQLCursor` (808)

## 34.24 TSQLQuery

### 34.24.1 Description

`TSQLQuery` exposes the properties and some methods introduced in `TCustomSQLQuery` (787). It encapsulates a single SQL statement: it implements all the necessary `#fcl.db.TDataset` (371) functionality to be able to handle a result set. It can also be used to execute a single SQL statement that does not return data, using the `TCustomSQLQuery.ExecSQL` (789) method.

Typically, the `TSQLQuery.Database` (818) property must be set once, the `TSQLQuery.Transaction` (818) property as well. Then the `TSQLQuery.SQL` (818) property can be set. Depending on the kind of SQL statement, the `Open` (389) method can be used to retrieve data, or the `ExecSQL` method can be used to execute the SQL statement (this can be used for DDL statements, or update statements).

See also: `TSQLTransaction` (837), `TSQLConnection` (796), `TCustomSQLQuery.ExecSQL` (789), `TSQLQuery.SQL` (818)



**34.24.2 Property overview**

Page	Properties	Access	Description
<a href="#">814</a>	Active		
<a href="#">814</a>	AfterCancel		
<a href="#">814</a>	AfterClose		
<a href="#">814</a>	AfterDelete		
<a href="#">815</a>	AfterEdit		
<a href="#">815</a>	AfterInsert		
<a href="#">815</a>	AfterOpen		
<a href="#">815</a>	AfterPost		
<a href="#">815</a>	AfterRefresh		Event triggered after refresh
<a href="#">815</a>	AfterScroll		
<a href="#">814</a>	AutoCalcFields		
<a href="#">815</a>	BeforeCancel		
<a href="#">816</a>	BeforeClose		
<a href="#">816</a>	BeforeDelete		
<a href="#">816</a>	BeforeEdit		
<a href="#">816</a>	BeforeInsert		
<a href="#">816</a>	BeforeOpen		
<a href="#">816</a>	BeforePost		
<a href="#">816</a>	BeforeRefresh		Event triggered before refresh
<a href="#">817</a>	BeforeScroll		
<a href="#">818</a>	Database		The <code>TSQLConnection</code> instance on which to execute SQL Statements
<a href="#">823</a>	DataSource		Source for parameter values for unbound parameters
<a href="#">820</a>	DeleteSQL		Statement to be used when inserting a new row in the database
<a href="#">813</a>	FieldDefs		List of field definitions.
<a href="#">814</a>	Filter		
<a href="#">814</a>	Filtered		
<a href="#">820</a>	IndexDefs		List of local index Definitions
<a href="#">819</a>	InsertSQL		Statement to be used when inserting a new row in the database
<a href="#">813</a>	MaxIndexesCount		Maximum allowed number of indexes.
<a href="#">817</a>	OnCalcFields		
<a href="#">817</a>	OnDeleteError		
<a href="#">817</a>	OnEditError		
<a href="#">817</a>	OnFilterRecord		
<a href="#">817</a>	OnNewRecord		
<a href="#">817</a>	OnPostError		
<a href="#">821</a>	Options		Options controlling the behaviour of the dataset
<a href="#">822</a>	ParamCheck		Should the SQL statement be checked for parameters
<a href="#">821</a>	Params		Parameters detected in the SQL statement.
<a href="#">822</a>	ParseSQL		Should the SQL statement be parsed or not
<a href="#">818</a>	ReadOnly		
<a href="#">820</a>	RefreshSQL		Refresh query to re-fetch field values after a DB update
<a href="#">813</a>	SchemaType		Schema type
<a href="#">823</a>	Sequence		Sequence to use for auto-generating values using a sequence
<a href="#">824</a>	ServerFilter		Append server-side filter to SQL statement
<a href="#">824</a>	ServerFiltered		Should server-side filter be applied
<a href="#">824</a>	ServerIndexDefs		List of indexes on the primary table of the query
<a href="#">818</a>	SQL		The SQL statement to execute
<a href="#">813</a>	StatementType		SQL statement type
<a href="#">818</a>	Transaction		Transaction in which to execute SQL statements
<a href="#">822</a>	UpdateMode		How to create update SQL statements.
<a href="#">819</a>	UpdateSQL		Statement to be used when updating an existing row in the database
<a href="#">823</a>	UsePrimaryKeyAsKey		Should primary key fields be marked <code>pfInKey</code>

### 34.24.3 TSQLQuery.SchemaType

Synopsis: Schema type

Declaration: `Property SchemaType :`

Visibility: public

Access:

Description: `SchemaType` is the schema type set by `TCustomSQLQuery.SetSchemaInfo` (790). It determines what kind of schema information will be returned by the `TSQLQuery` instance.

See also: `TCustomSQLQuery.SetSchemaInfo` (790), `RetrievingSchemaInformation` (773)

### 34.24.4 TSQLQuery.StatementType

Synopsis: SQL statement type

Declaration: `Property StatementType :`

Visibility: public

Access:

Description: `StatementType` is determined during the `Prepare` (788) call when `ParseSQL` (822) is set to `True`. It gives an indication of the type of SQL statement that is being executed.

See also: `TSQLQuery.SQL` (818), `TSQLQuery.ParseSQL` (822), `TSQLQuery.Params` (821)

### 34.24.5 TSQLQuery.MaxIndexesCount

Synopsis: Maximum allowed number of indexes.

Declaration: `Property MaxIndexesCount :`

Visibility: published

Access:

Description: `MaxIndexesCount` determines the number of index entries that the dataset will reserve for indexes. No more indexes than indicated here can be used. The property must be set before the dataset is opened. The minimum value for this property is 1. The default value is 2.

If an index is added and the current index count equals `MaxIndexesCount`, an exception will be raised.

Errors: Attempting to set this property while the dataset is active will raise an exception.

### 34.24.6 TSQLQuery.FieldDefs

Synopsis: List of field definitions.

Declaration: `Property FieldDefs :`

Visibility: published

Access:

### **34.24.7 TSQLQuery.Active**

Declaration: `Property Active :`

Visibility: published

Access:

### **34.24.8 TSQLQuery.AutoCalcFields**

Declaration: `Property AutoCalcFields :`

Visibility: published

Access:

### **34.24.9 TSQLQuery.Filter**

Declaration: `Property Filter :`

Visibility: published

Access:

### **34.24.10 TSQLQuery.Filtered**

Declaration: `Property Filtered :`

Visibility: published

Access:

### **34.24.11 TSQLQuery.AfterCancel**

Declaration: `Property AfterCancel :`

Visibility: published

Access:

### **34.24.12 TSQLQuery.AfterClose**

Declaration: `Property AfterClose :`

Visibility: published

Access:

### **34.24.13 TSQLQuery.AfterDelete**

Declaration: `Property AfterDelete :`

Visibility: published

Access:

#### **34.24.14 TSQLQuery.AfterEdit**

Declaration: `Property AfterEdit` :

Visibility: published

Access:

#### **34.24.15 TSQLQuery.AfterInsert**

Declaration: `Property AfterInsert` :

Visibility: published

Access:

#### **34.24.16 TSQLQuery.AfterOpen**

Declaration: `Property AfterOpen` :

Visibility: published

Access:

#### **34.24.17 TSQLQuery.AfterPost**

Declaration: `Property AfterPost` :

Visibility: published

Access:

#### **34.24.18 TSQLQuery.AfterRefresh**

Synopsis: Event triggered after refresh

Declaration: `Property AfterRefresh` :

Visibility: published

Access:

#### **34.24.19 TSQLQuery.AfterScroll**

Declaration: `Property AfterScroll` :

Visibility: published

Access:

#### **34.24.20 TSQLQuery.BeforeCancel**

Declaration: `Property BeforeCancel` :

Visibility: published

Access:



#### **34.24.21 TSQLQuery.BeforeClose**

Declaration: `Property BeforeClose :`

Visibility: published

Access:

#### **34.24.22 TSQLQuery.BeforeDelete**

Declaration: `Property BeforeDelete :`

Visibility: published

Access:

#### **34.24.23 TSQLQuery.BeforeEdit**

Declaration: `Property BeforeEdit :`

Visibility: published

Access:

#### **34.24.24 TSQLQuery.BeforeInsert**

Declaration: `Property BeforeInsert :`

Visibility: published

Access:

#### **34.24.25 TSQLQuery.BeforeOpen**

Declaration: `Property BeforeOpen :`

Visibility: published

Access:

#### **34.24.26 TSQLQuery.BeforePost**

Declaration: `Property BeforePost :`

Visibility: published

Access:

#### **34.24.27 TSQLQuery.BeforeRefresh**

Synopsis: Event triggered before refresh

Declaration: `Property BeforeRefresh :`

Visibility: published

Access:

#### **34.24.28 TSQLQuery.BeforeScroll**

Declaration: Property BeforeScroll :

Visibility: published

Access:

#### **34.24.29 TSQLQuery.OnCalcFields**

Declaration: Property OnCalcFields :

Visibility: published

Access:

#### **34.24.30 TSQLQuery.OnDeleteError**

Declaration: Property OnDeleteError :

Visibility: published

Access:

#### **34.24.31 TSQLQuery.OnEditError**

Declaration: Property OnEditError :

Visibility: published

Access:

#### **34.24.32 TSQLQuery.OnFilterRecord**

Declaration: Property OnFilterRecord :

Visibility: published

Access:

#### **34.24.33 TSQLQuery.OnNewRecord**

Declaration: Property OnNewRecord :

Visibility: published

Access:

#### **34.24.34 TSQLQuery.OnPostError**

Declaration: Property OnPostError :

Visibility: published

Access:

### 34.24.35 TSQLQuery.Database

Synopsis: The `TSQLConnection` instance on which to execute SQL Statements

Declaration: `Property Database :`

Visibility: published

Access:

Description: `Database` is the SQL connection (of type `TSQLConnection` (796)) on which SQL statements will be executed, and from which result sets will be retrieved. This property must be set before any form of SQL command can be executed, just like the `Transaction` (818) property must be set.

Multiple `TSQLQuery` instances can be connected to a database at the same time.

See also: `TSQLQuery.Transaction` (818), `TSQLConnection` (796), `TSQLTransaction` (837)

### 34.24.36 TSQLQuery.Transaction

Synopsis: Transaction in which to execute SQL statements

Declaration: `Property Transaction :`

Visibility: published

Access:

Description: `Transaction` must be set to a SQL transaction (of type `TSQLTransaction` (837)) component. All SQL statements (`SQL / InsertSQL / updateSQL / DeleteSQL`) etc.) will be executed in the context of this transaction.

The transaction must be connected to the same database instance as the query itself.

Multiple `TSQLQuery` instances can be connected to a transaction at the same time. If the transaction is rolled back, all changes done by all `TSQLQuery` instances will be rolled back.

See also: `TSQLQuery.Database` (818), `TSQLConnection` (796), `TSQLTransaction` (837)

### 34.24.37 TSQLQuery.ReadOnly

Declaration: `Property ReadOnly :`

Visibility: published

Access:

### 34.24.38 TSQLQuery.SQL

Synopsis: The SQL statement to execute

Declaration: `Property SQL :`

Visibility: published

Access:

**Description:** SQL is the SQL statement that will be executed when ExecSQL (789) is called, or Open (389) is called. It should contain a valid SQL statement for the connection to which the TSQLQuery (810) component is connected. SQLDB will not attempt to modify the SQL statement so it is accepted by the SQL engine.

Setting or modifying the SQL statement will call UnPrepare (789)

If ParseSQL (822) is True, the SQL statement will be parsed and the Params (821) property will be updated with the names of the parameters found in the SQL statement.

See also Using parameters

See also: TSQLQuery.ParseSQL (822), TSQLQuery.Params (821), TCustomSQLQuery.ExecSQL (789), TDataSet.Open (389)

### 34.24.39 TSQLQuery.InsertSQL

**Synopsis:** Statement to be used when inserting a new row in the database

**Declaration:** Property InsertSQL :

**Visibility:** published

**Access:**

**Description:** InsertSQL can be used to specify an SQL INSERT statement, which is used when a new record was appended to the dataset, and the changes must be written to the database. TSQLQuery can generate an insert statement by itself for many cases, but in case it fails, the statement to be used for the insert can be specified here.

The SQL statement should be parameterized according to the conventions for specifying parameters.

Note that old field values can be specified as :OLD\_FIELDNAME

See also: TSQLQuery.SQL (818), TSQLQuery.UpdateSQL (819), TSQLQuery.DeleteSQL (820), TSQLQuery.UpdateMode (822), UsingParams (774), UpdateSQLS (773)

### 34.24.40 TSQLQuery.UpdateSQL

**Synopsis:** Statement to be used when updating an existing row in the database

**Declaration:** Property UpdateSQL :

**Visibility:** published

**Access:**

**Description:** UpdateSQL can be used to specify an SQL UPDATE statement, which is used when an existing record was modified in the dataset, and the changes must be written to the database. TSQLQuery can generate an update statement by itself for many cases, but in case it fails, the statement to be used for the update can be specified here.

The SQL statement should be parameterized according to the conventions for specifying parameters.

Note that old field values can be specified as :OLD\_FIELDNAME

See also: TSQLQuery.SQL (818), TSQLQuery.InsertSQL (819), TSQLQuery.DeleteSQL (820), TSQLQuery.UpdateMode (822), UsingParams (774), UpdateSQLS (773)

**34.24.41 TSQLQuery.DeleteSQL**

Synopsis: Statement to be used when inserting a new row in the database

Declaration: `Property DeleteSQL :`

Visibility: published

Access:

Description: `DeleteSQL` can be used to specify an SQL `DELETE` statement, which is used when an existing record was deleted from the dataset, and the changes must be written to the database. `TSQLQuery` can generate a delete statement by itself for many cases, but in case it fails, the statement to be used for the insert can be specified here.

The SQL statement should be parameterized according to the conventions for specifying parameters.

Note that old field values can be specified as `:OLD_FIELDNAME`

See also: `TSQLQuery.SQL` (818), `TSQLQuery.UpdateSQL` (819), `TSQLQuery.DeleteSQL` (820), `TSQLQuery.UpdateMode` (822), `UsingParams` (774), `UpdateSQLS` (773)

**34.24.42 TSQLQuery.RefreshSQL**

Synopsis: Refresh query to re-fetch field values after a DB update

Declaration: `Property RefreshSQL :`

Visibility: published

Access:

Description: `RefreshSQL` can be used to specify a SQL statement that is executed after an `UPDATE` or `INSERT` operation. The query will be executed, and the values of all fields in the result set will be copied to the dataset. This SQL statement is only executed during the `ApplyUpdates` operation, not during the `Post` call itself.

A `RefreshSQL` can be constructed automatically by `SQLDB` by setting the `pfRefreshOnUpdate` or `pfRefreshOnInsert` flags in the `ProviderFlags` (769) of the fields in the dataset, depending on whether the operation was an update or insert.

For SQL engines that support `RETURNING` clauses, the `RETURNING` clause will be used to refresh field values, unless `sqrPreferRefresh` is specified in `TSQLQuery.Options` (821)

See also: `TField.ProviderFlags` (769), `TSQLQuery.Options` (821)

**34.24.43 TSQLQuery.IndexDefs**

Synopsis: List of local index Definitions

Declaration: `Property IndexDefs :`

Visibility: published

Access:

Description: List of local index Definitions

See also: `TCustomBufDataset.IndexDefs` (769)

### 34.24.44 TSQLQuery.Options

Synopsis: Options controlling the behaviour of the dataset

Declaration: `Property Options :`

Visibility: published

Access:

Description: `Options` controls the behaviour of the dataset. The following options can be specified:

**sqoKeepOpenOnCommit** The default SQLDB behaviour is to close all datasets connected to a transaction when a transaction is committed or rolled back, which means that transactions must remain active as long as the dataset is open. This can create problems with locking of records etc. With this option set, the dataset will be kept open. Note that setting this option will cause SQLDB to fetch all records in the result set in memory.

**sqoAutoApplyUpdates** Setting this option will make `TSQLQuery` call `ApplyUpdates` after every `Post` or `Delete` operation.

**sqoAutoCommit** Setting this option will make `TSQLQuery` call `commit` after every `ApplyUpdates`

**sqoCancelUpdatesOnRefresh** Setting this option will cause `TSQLQuery` to abandon all pending changes when `Refresh` is called. The default behaviour is to raise an exception when `Refresh` is called and there are pending changes

**sqoPreferRefresh** If the database engine supports `RETURNING`, then the returning mechanism is used to fetch field values after an update of the database. Setting this option will disable the use of `RETURNING` and will fetch updated or new values instead with the `TSQLQuery.RefreshSQL` (820) property or a constructed refresh SQL statement.

See also: `TCustomSQLQuery.ApplyUpdates` (791), `TCustomSQLQuery.Post` (791), `TCustomSQLQuery.Delete` (791)

### 34.24.45 TSQLQuery.Params

Synopsis: Parameters detected in the SQL statement.

Declaration: `Property Params :`

Visibility: published

Access:

Description: `Params` contains the parameters used in the SQL statement. This collection is only updated when `ParseSQL` (822) is `True`. For each named parameter in the `SQL` (818) property, a named item will appear in the collection, and the collection will be used to retrieve values from.

When `Open` (389) or `ExecSQL` (789) is called, and the `Datasource` (823) property is not `Nil`, then for each parameter for which no value was explicitly set (its `Bound` (494) property is `False`), the value will be retrieved from the dataset connected to the datasource.

For each parameter, a field with the same name will be searched, and its value and type will be copied to the (unbound) parameter. The parameter remains unbound.

The Update, delete and insert SQL statements are not scanned for parameters.

See also: `TSQLQuery.SQL` (818), `TSQLQuery.ParseSQL` (822), `TParam.Bound` (494), `UsingParams` (774), `UpdateSQLS` (773)

**34.24.46 TSQLQuery.ParamCheck**

Synopsis: Should the SQL statement be checked for parameters

Declaration: `Property ParamCheck :`

Visibility: published

Access:

Description: `ParamCheck` must be set to `False` to disable the parameter check. The default value `True` indicates that the SQL statement should be checked for parameter names (in the form `:ParamName`), and corresponding `TParam` (484) instances should be added to the `Params` (769) property.

When executing some DDL statements, e.g. a "create procedure" SQL statement can contain parameters. These parameters should not be converted to `TParam` instances.

See also: `TParam` (484), `Params` (769), `TSQLStatement.ParamCheck` (835)

**34.24.47 TSQLQuery.ParseSQL**

Synopsis: Should the SQL statement be parsed or not

Declaration: `Property ParseSQL :`

Visibility: published

Access:

Description: `ParseSQL` can be set to `False` to prevent `TSQLQuery` from parsing the `SQL` (818) property and attempting to detect the statement type or updating the `Params` (821) or `StatementType` (813) properties.

This can be used when `SQLDB` has problems parsing the SQL statement, or when the SQL statement contains parameters that are part of a DDL statement such as a `CREATE PROCEDURE` statement to create a stored procedure.

Note that in this case the statement will be passed as-is to the SQL engine, no parameter values will be passed on.

See also: `TSQLQuery.SQL` (818), `TSQLQuery.Params` (821)

**34.24.48 TSQLQuery.UpdateMode**

Synopsis: How to create update SQL statements.

Declaration: `Property UpdateMode :`

Visibility: published

Access:

Description: `UpdateMode` determines how the `WHERE` clause of the `UpdateSQL` (819) and `DeleteSQL` (820) statements are auto-generated.

**upWhereAll** Use all old field values

**upWhereChanged** Use only old field values of modified fields

**upWhereKeyOnly** Only use key fields in the where clause.

See also: `TSQLQuery.UpdateSQL` (819), `TSQLQuery.InsertSQL` (819)

**34.24.49 TSQLQuery.UsePrimaryKeyAsKey**

Synopsis: Should primary key fields be marked `pfInKey`

Declaration: `Property UsePrimaryKeyAsKey :`

Visibility: published

Access:

Description: `UsePrimaryKeyAsKey` can be set to `True` to let `TSQLQuery` fetch all server indexes and if there is a primary key, update the `ProviderFlags` (445) of the fields in the primary key with `pfInKey` (329).

The effect of this is that when `UpdateMode` (822) equals `upWhereKeyOnly`, then only the fields that are part of the primary key of the table will be used in the update statements. For more information, see `UpdateSQLs` (773).

Note that this property only takes effect if the fields are the default fields: if persistent fields were created, the providerflags of the fields are not updated.

See also: `TSQLQuery.UpdateMode` (822), `TCustomBufDataset.Unidirectional` (769), `TField.ProviderFlags` (445), `pfInKey` (329), `UpdateSQLs` (773)

**34.24.50 TSQLQuery.DataSource**

Synopsis: Source for parameter values for unbound parameters

Declaration: `Property DataSource :`

Visibility: published

Access:

Description: `Datasource` can be set to a dataset which will be used to retrieve values for the parameters if they were not explicitly specified.

When `Open` (389) or `ExecSQL` (789) is called, and the `Datasource` property is not `Nil` then for each parameter for which no value was explicitly set (its `Bound` (494) property is `False`), the value will be retrieved from the dataset connected to the `datasource`.

For each parameter, a field with the same name will be searched, and its value and type will be copied to the (unbound) parameter. The parameter remains unbound.

See also: `Params` (821), `ExecSQL` (789), `UsingParams` (774), `TParam.Bound` (494)

**34.24.51 TSQLQuery.Sequence**

Synopsis: Sequence to use for auto-generating values using a sequence

Declaration: `Property Sequence :`

Visibility: published

Access:

Description: `Sequence` allows `TSQLQuery` to automate generation of a new value for a field using a sequence in the database.

To this end, the properties in `TSQLSequence` (832) must be set to appropriate values, and `TSQLQuery` will automatically generate a new value for the indicated field during insert or post (depending on the value of `TSQLSequence.ApplyEvent` (834)).

See also: `TSQLSequence` (832)



**34.24.52 TSQLQuery.ServerFilter**

Synopsis: Append server-side filter to SQL statement

Declaration: `Property ServerFilter :`

Visibility: published

Access:

Description: `ServerFilter` can be set to a valid `WHERE` clause (without the `WHERE` keyword). It will be appended to the `select` statement in SQL (818), when `ServerFiltered` (824) is set to `True`. if `ServerFiltered` (824) is set to `False`, `ServerFilter` is ignored.

If the dataset is active and `ServerFiltered` (824) is set to `true`, then changing this property will re-fetch the data from the server.

This property cannot be used when `ParseSQL` (822) is `False`, because the statement must be parsed in order to know where the `WHERE` clause must be inserted: the `TSQLQuery` class will intelligently insert the clause in an SQL `select` statement.

Errors: Setting this property when `ParseSQL` (822) is `False` will result in an exception.

See also: `ServerFiltered` (824)

**34.24.53 TSQLQuery.ServerFiltered**

Synopsis: Should server-side filter be applied

Declaration: `Property ServerFiltered :`

Visibility: published

Access:

Description: `ServerFiltered` can be set to `True` to apply `ServerFilter` (824). A change in the value for this property will re-fetch the query results if the dataset is active.

Errors: Setting this property to `True` when `ParseSQL` (822) is `False` will result in an exception.

See also: `ParseSQL` (822), `ServerFilter` (824)

**34.24.54 TSQLQuery.ServerIndexDefs**

Synopsis: List of indexes on the primary table of the query

Declaration: `Property ServerIndexDefs :`

Visibility: published

Access:

Description: `ServerIndexDefs` will be filled - during the `Prepare` call - with the list of indexes defined on the primary table in the query if `UsePrimaryKeyAsKey` (823) is `True`. If a primary key is found, then the fields in it will be marked

See also: `UsePrimaryKeyAsKey` (823), `Prepare` (788)

## 34.25 TSQLScript

### 34.25.1 Description

`TSQLScript` is a component that can be used to execute many SQL statements using a `TSQLQuery` (810) component. The SQL statements are specified in a script `TSQLScript.Script` (829) separated by a terminator character (typically a semicolon (;)).

See also: `TSQLTransaction` (837), `TSQLConnection` (796), `TCustomSQLQuery.ExecSQL` (789), `TSQLQuery.SQL` (818)

### 34.25.2 Method overview

Page	Method	Description
825	<code>Create</code>	Create a new <code>TSQLScript</code> instance.
826	<code>Destroy</code>	Remove the <code>TSQLScript</code> instance from memory.
826	<code>Execute</code>	Execute the script.
826	<code>ExecuteScript</code>	Convenience function, simply calls <code>Execute</code>

### 34.25.3 Property overview

Page	Properties	Access	Description
826	<code>Aborted</code>		True when the script was aborted
828	<code>AutoCommit</code>		Automatically commit every statement
830	<code>CommentsinSQL</code>		Should comments be passed to the SQL engine ?
827	<code>DataBase</code>	rw	Database on which to execute the script
829	<code>Defines</code>		Defined macros
829	<code>Directives</code>		List of directives
828	<code>DollarStrings</code>		List of alternate string delimiter token sequences
827	<code>Line</code>		Current line of execution in the script
827	<code>OnDirective</code>	rw	Event handler if a directive is encountered
832	<code>OnException</code>		Exception handling event
829	<code>Script</code>		The script to execute
830	<code>Terminator</code>		Terminator character.
827	<code>Transaction</code>	rw	Transaction to use in the script
831	<code>UseCommit</code>		Control automatic handling of the <code>COMMIT</code> command.
831	<code>UseDefines</code>		Automatically handle pre-processor defines
828	<code>UseDollarString</code>		Enable support for dollarstrings
830	<code>UseSetTerm</code>		Should the <code>SET TERM</code> directive be recognized

### 34.25.4 TSQLScript.Create

**Synopsis:** Create a new `TSQLScript` instance.

**Declaration:** `constructor Create(AOwner: TComponent); Override`

**Visibility:** `public`

**Description:** `Create` instantiates a `TSQLQuery` (810) instance which will be used to execute the queries, and then calls the inherited constructor.

See also: `TSQLScript.Destroy` (826)

### 34.25.5 TSQLScript.Destroy

Synopsis: Remove the TSQLScript instance from memory.

Declaration: `destructor Destroy;` Override

Visibility: public

Description: `Destroy` frees the TSQLQuery (810) instance that was created during the `Create` constructor from memory and then calls the inherited destructor.

See also: TSQLScript.Create (825)

### 34.25.6 TSQLScript.Execute

Synopsis: Execute the script.

Declaration: `procedure Execute;` Override

Visibility: public

Description: `Execute` will execute the statements specified in `Script` (829) one by one, till the last statement is processed or an exception is raised.

If an error occurs during execution, normally an exception is raised. If the TSQLScript.OnException (832) event handler is set, it may stop the event handler.

Errors: Handle errors using TSQLScript.OnException (832).

See also: `Script` (829), TSQLScript.OnException (832)

### 34.25.7 TSQLScript.ExecuteScript

Synopsis: Convenience function, simply calls `Execute`

Declaration: `procedure ExecuteScript`

Visibility: public

Description: `ExecuteScript` is a convenience function, it simply calls `Execute`. The statements in the script will be executed one by one.

### 34.25.8 TSQLScript.Aborted

Synopsis: True when the script was aborted

Declaration: `Property Aborted :`

Visibility: public

Access:

Description: `Aborted` is set to `True` if the SQL script execution is aborted by one of the directives in the script. It is read-only.

### 34.25.9 TSQLScript.Line

Synopsis: Current line of execution in the script

Declaration: `Property Line :`

Visibility: public

Access:

Description: `Line` is the line number (0 based) of the currently executed statement in the script. For multiline statements, the last line of the statement is counted as the current line.

### 34.25.10 TSQLScript.DataBase

Synopsis: Database on which to execute the script

Declaration: `Property DataBase : TDatabase`

Visibility: published

Access: Read,Write

Description: `Database` should be set to the `TSQLConnection` (796) descendent. All SQL statements in the `Script` (829) property will be executed on this database.

See also: `TSQLConnection` (796), `TSQLScript.Transaction` (827), `TSQLScript.Script` (829)

### 34.25.11 TSQLScript.Transaction

Synopsis: Transaction to use in the script

Declaration: `Property Transaction : TDBTransaction`

Visibility: published

Access: Read,Write

Description: `Transaction` is the transaction instance to use when executing statements. If the SQL script contains any `COMMIT` statements, they will be handled using the `TSQLTransaction.CommitRetaining` (838) method.

See also: `TSQLTransaction` (837), `TSQLTransaction.CommitRetaining` (838), `TSQLScript.Database` (827)

### 34.25.12 TSQLScript.OnDirective

Synopsis: Event handler if a directive is encountered

Declaration: `Property OnDirective : TSQLScriptDirectiveEvent`

Visibility: published

Access: Read,Write

Description: `OnDirective` is called when a directive is encountered. When parsing the script, the script engine checks the first word of the statement. If it matches one of the words in `Directives` (829) property then the `OnDirective` event handler is called with the name of the directive and the rest of the statement as parameters. This can be used to handle all kind of pre-processing actions such as `Set term \;`

See also: `Directives` (829)

### 34.25.13 TSQLScript.AutoCommit

Synopsis: Automatically commit every statement

Declaration: `Property AutoCommit :`

Visibility: published

Access:

Description: `AutoCommit` can be set to `True` to commit every executed statement in the script. By default, this is set to `false`.

See also: `TSQLScript.Transaction` ([827](#))

### 34.25.14 TSQLScript.UseDollarString

Synopsis: Enable support for dollarstrings

Declaration: `Property UseDollarString :`

Visibility: published

Access:

Description: `UseDollarString` enables support for so-called "DollarString" delimiters for string literals. This means that the normal string literal delimiter (') is enhanced with any string appearing in the `DollarStrings` ([828](#)) property.

Setting `UseDollarString` to `true` incurs a speed penalty, so it is better not to enable it unless it is really necessary.

This is needed for instance for PostgreSQL, where stored procedure code blocks are enclosed in "\$\$" signs, and are treated as a string literal.

See also: `TSQLScript.DollarStrings` ([828](#))

### 34.25.15 TSQLScript.DollarStrings

Synopsis: List of alternate string delimiter token sequences

Declaration: `Property DollarStrings :`

Visibility: published

Access:

Description: `DollarStrings` contains a list of additional string delimiter tokens. The value of this property is ignored unless `TSQLScript.UseDollarString` ([828](#)) is also set to `True`.

For PostgreSQL, this should be set to `$$`, as this is the most commonly used string delimiter for stored procedures.

See also: `TSQLScript.UseDollarString` ([828](#))

### 34.25.16 TSQLScript.Directives

Synopsis: List of directives

Declaration: `Property Directives :`

Visibility: published

Access:

Description: `Directives` is a stringlist with words that should be recognized as directives. They will be handled using the `OnDirective` (827) event handler. The list should contain one word per line, no spaces allowed.

See also: `OnDirective` (827)

### 34.25.17 TSQLScript.Defines

Synopsis: Defined macros

Declaration: `Property Defines :`

Visibility: published

Access:

Description: `Defines` contains the list of defined macros for use with the `TSQLScript.UseDefines` (831) property. Each line should contain a macro name. The names of the macros are case insensitive. The `#DEFINE` and `#UNDEFINE` directives will add or remove macro names from this list.

See also: `TSQLScript.UseDefines` (831)

### 34.25.18 TSQLScript.Script

Synopsis: The script to execute

Declaration: `Property Script :`

Visibility: published

Access:

Description: `Script` contains the list of SQL statements to be executed. The statements should be separated by the character specified in the `Terminator` (830) property. Each of the statement will be executed on the database specified in `Database` (827). using the equivalent of the `TCustomSQLQuery.ExecSQL` (789) statement. The statements should not return result sets, but other than that all kind of statements are allowed.

Comments will be conserved and passed on in the statements to be executed, depending on the value of the `TSQLScript.CommentsinSQL` (830) property. If that property is `False`, comments will be stripped prior to executing the SQL statements.

See also: `TSQLScript.CommentsinSQL` (830), `TSQLScript.Terminator` (830), `TSQLScript.DataBase` (827)

### 34.25.19 TSQLScript.Terminator

Synopsis: Terminator character.

Declaration: `Property Terminator :`

Visibility: published

Access:

Description: `Terminator` is the character used by `TSQLScript` to delimit SQL statements. By default it equals the semicolon (`;`), which is the customary SQL command terminating character. By itself `TSQLScript` does not recognize complex statements such as `Create Procedure` which can contain terminator characters such as `;"`. Instead, `TSQLScript` will scan the script for the `Terminator` character. Using directives such as `SET TERM` the terminator character may be changed in the script.

See also: `OnDirective` ([827](#)), `Directives` ([829](#))

### 34.25.20 TSQLScript.CommentsinSQL

Synopsis: Should comments be passed to the SQL engine ?

Declaration: `Property CommentsinSQL :`

Visibility: published

Access:

Description: `CommentsInSQL` can be set to `True` to let `TSQLScript` preserve any comments it finds in the script. The comments will be passed to the `SQLConnection` as part of the commands. If the property is set to `False` the comments are discarded.

By default, `TSQLScript` discards comments.

See also: `TSQLScript.Script` ([829](#))

### 34.25.21 TSQLScript.UseSetTerm

Synopsis: Should the SET TERM directive be recognized

Declaration: `Property UseSetTerm :`

Visibility: published

Access:

Description: `UseSetTerm` can be set to `True` to let `TSQLScript` automatically handle the `SET TERM` directive and set the `TSQLScript.Terminator` ([830](#)) character based on the value specified in the `SET TERM` directive. This means that the following directive:

```
SET TERM ^ ;
```

will set the terminator to the caret character. Conversely, the

```
SET TERM ; ^
```

will then switch the terminator character back to the commonly used semicolon (`;`).

See also: `TSQLScript.Terminator` ([830](#)), `TSQLScript.Script` ([829](#)), `TSQLScript.Directives` ([829](#))

### 34.25.22 TSQLScript.UseCommit

Synopsis: Control automatic handling of the COMMIT command.

Declaration: Property UseCommit :

Visibility: published

Access:

Description: UseCommit can be set to True to let TSQLScript automatically handle the commit command as a directive. If it is set, the COMMIT command is registered as a directive, and the TSQLScript.Transaction (827) will be committed and restarted at once whenever the COMMIT directive appears in the script.

If this property is set to False then the commit command will be passed on to the SQL engine like any other SQL command in the script.

See also: TSQLScript.Transaction (827), TSQLScript.Directives (829)

### 34.25.23 TSQLScript.UseDefines

Synopsis: Automatically handle pre-processor defines

Declaration: Property UseDefines :

Visibility: published

Access:

Description: UseDefines will automatically register the following pre-processing directives:

```
#IFDEF
#IFNDEF
#ELSE
#ENDIF
#DEFINE
#UNDEF
#UNDEFINE
```

Additionally, these directives will be automatically handled by the TSQLScript component. This can be used to add conditional execution of the SQL script: they are treated as the conditional compilation statements found in the C macro preprocessor or the FPC conditional compilation features. The initial list of defined macros can be specified in the Defines (829) property, where one define per line can be specified.

In the following example, the correct statement to create a sequence is selected based on the presence of the macro FIREBIRD in the list of defines:

```
#IFDEF FIREBIRD
CREATE GENERATOR GEN_MYID;
#ELSE
CREATE SEQUENCE GEN_MYID;
#ENDIF
```

See also: TSQLScript.Script (829), TSQLScript.Defines (829)



### 34.25.24 TSQLScript.OnException

Synopsis: Exception handling event

Declaration: `Property OnException :`

Visibility: `published`

Access:

Description: `OnException` can be set to handle an exception during the execution of a statement or directive when the script is executed. The exception is passed to the handler in the `TheException` parameter. On return, the value of the `Continue` parameter is checked: if it is set to `True`, then the exception is ignored. If it is set to `False` (the default), then the exception is re-raised, and script execution will stop.

See also: `TSQLScript.Execute` ([826](#))

## 34.26 TSQLSequence

### 34.26.1 Description

`TSQLSequence` is an auxiliary class, used to auto-generate numerical values for fields in databases that support sequences; it is used as a property of `TSQLQuery` ([810](#)) and its properties determine which field must be auto-generated, and at what moment this value must be generated.

See also: `TSQLConnection.GetSequenceNames` ([800](#)), `TSQLConnection.GetNextValue` ([801](#))

### 34.26.2 Method overview

Page	Method	Description
<a href="#">833</a>	<code>Apply</code>	Apply a new value to a field
<a href="#">833</a>	<code>Assign</code>	Assign one <code>TSQLSequence</code> to another
<a href="#">832</a>	<code>Create</code>	Create a new instance
<a href="#">833</a>	<code>GetNextValue</code>	Get a next value for the sequence

### 34.26.3 Property overview

Page	Properties	Access	Description
<a href="#">834</a>	<code>ApplyEvent</code>	<code>rw</code>	When to apply the new value
<a href="#">833</a>	<code>FieldName</code>	<code>rw</code>	Field to apply sequence to
<a href="#">834</a>	<code>IncrementBy</code>	<code>rw</code>	Value to increment sequence with
<a href="#">834</a>	<code>SequenceName</code>	<code>rw</code>	Sequence name to get values from

### 34.26.4 TSQLSequence.Create

Synopsis: Create a new instance

Declaration: `constructor Create(AQuery: TCustomSQLQuery)`

Visibility: `public`

Description: `Create` instantiates a new sequence. It requires a `TSQLQuery` ([810](#)) instance, which it needs to have access to a connection.

See also: `TSQLQuery` ([810](#))

### 34.26.5 TSQLSequence.Assign

Synopsis: Assign one TSQLSequence to another

Declaration: `procedure Assign(Source: TPersistent); Override`

Visibility: `public`

Description: Assign is overridden by TSQLSequence to copy all properties from one instance to another.

Errors: None.

See also: TSQLSequence.FieldName (833), TSQLSequence.SequenceName (834), TSQLSequence.IncrementBy (834)

### 34.26.6 TSQLSequence.Apply

Synopsis: Apply a new value to a field

Declaration: `procedure Apply`

Visibility: `public`

Description: Apply applies the new value it gets for TSQLSequence.SequenceName (834) using TSQLSequence.GetNextValue (833) to the field TSQLSequence.FieldName (833) of the dataset it is attached to.

Errors: If the dataset is not attached to a connected database, an exception will be raised.

See also: GetNextValue (769), TSQLSequence.FieldName (833), TSQLSequence.SequenceName (834), TSQLSequence.IncrementBy (834)

### 34.26.7 TSQLSequence.GetNextValue

Synopsis: Get a next value for the sequence

Declaration: `function GetNextValue : Int64`

Visibility: `public`

Description: GetNextValue gets a new value for generator TSQLSequence.SequenceName (834) using TSQLSequence.IncrementBy (834)

Errors: If the dataset is not attached to a connected database, an exception will be raised.

See also: TSQLSequence.FieldName (833), TSQLSequence.SequenceName (834), TSQLSequence.IncrementBy (834), TSQLSequence.Apply (833), TSQLConnection.GetNextValue (801)

### 34.26.8 TSQLSequence.FieldName

Synopsis: Field to apply sequence to

Declaration: `Property FieldName : string`

Visibility: `published`

Access: Read,Write

Description: FieldName is the name of the field TSQLSequence will apply the new value to when Apply (769) is called. It must be a valid fieldname of the dataset that owns the TSQLSequence instance.

See also: TSQLSequence.Apply (833), TSQLSequence.SequenceName (834), TSQLSequence.IncrementBy (834)

### 34.26.9 TSQLSequence.SequenceName

Synopsis: Sequence name to get values from

Declaration: `Property SequenceName : string`

Visibility: published

Access: Read,Write

Description: `SequenceName` is the name of the sequence `TSQLSequence` will get a new value of when `GetNextValue` (833) is called.

See also: `TSQLSequence.Apply` (833), `TSQLSequence.GetNextValue` (833), `TSQLSequence.IncrementBy` (834)

### 34.26.10 TSQLSequence.IncrementBy

Synopsis: Value to increment sequence with

Declaration: `Property IncrementBy : Integer`

Visibility: published

Access: Read,Write

Description: `IncrementBy` is the value that will be added to the current value of the sequence `TSQLSequence.SequenceName` (834) when `TSQLSequence.GetNextValue` (833) is called.

See also: `TSQLSequence.Apply` (833), `TSQLSequence.GetNextValue` (833), `TSQLSequence.SequenceName` (834)

### 34.26.11 TSQLSequence.ApplyEvent

Synopsis: When to apply the new value

Declaration: `Property ApplyEvent : TSQLSequenceApplyEvent`

Visibility: published

Access: Read,Write

Description: `ApplyEvent` determines when the new value will be applied to a field: On new record (i.e. when `Insert` (386) or `Append` (377) is called) or when a newly inserted record is saved (when `Post` (389) is called).

See also: `TSQLSequenceApplyEvent` (781)

## 34.27 TSQLStatement

### 34.27.1 Description

`TSQLStatement` is a descendent of `TCustomSQLStatement` (792) which simply publishes the protected properties of that component.

See also: `TCustomSQLStatement` (792)

### 34.27.2 Property overview

Page	Properties	Access	Description
<a href="#">835</a>	Database		Database instance to execute statement on.
<a href="#">835</a>	DataSource		Datasource to copy parameter values from
<a href="#">835</a>	ParamCheck		Should SQL be checked for parameters
<a href="#">836</a>	Params		List of parameters.
<a href="#">836</a>	ParseSQL		Parse the SQL statement
<a href="#">836</a>	SQL		The SQL statement to execute
<a href="#">837</a>	Transaction		The transaction in which the SQL statement should be executed.

### 34.27.3 TSQLStatement.Database

Synopsis: Database instance to execute statement on.

Declaration: `Property Database :`

Visibility: published

Access:

Description: `Database` must be set to an instance of a `TSQLConnection` ([796](#)) descendent. It must be set, together with `Transaction` ([769](#)) in order to be able to call `Prepare` ([793](#)) or `Execute` ([794](#)).

See also: `Transaction` ([769](#)), `Prepare` ([793](#)), `Execute` ([794](#))

### 34.27.4 TSQLStatement.DataSource

Synopsis: Datasource to copy parameter values from

Declaration: `Property DataSource :`

Visibility: published

Access:

Description: `Datasource` can be set to a `#fcl.db.TDatasource` ([409](#)) instance. When `Execute` ([794](#)) is called, any unbound parameters remain empty, but if `DataSource` is set, the value of these parameters will be searched in the fields of the associated dataset. If a field with a name equal to the parameter is found, the value of that field is copied to the parameter. No such field exists, an exception is raised.

See also: `#fcl.db.TDatasource` ([409](#)), `Execute` ([794](#)), `#fcl.db.TParam.Bound` ([494](#))

### 34.27.5 TSQLStatement.ParamCheck

Synopsis: Should SQL be checked for parameters

Declaration: `Property ParamCheck :`

Visibility: published

Access:

Description: `ParamCheck` must be set to `False` to disable the parameter check. The default value `True` indicates that the SQL statement should be checked for parameter names (in the form `:ParamName`), and corresponding `TParam` ([484](#)) instances should be added to the `Params` ([769](#)) property.

When executing some DDL statements, e.g. a "create procedure" SQL statement can contain parameters. These parameters should not be converted to `TParam` instances.

See also: `TParam` ([484](#)), `Params` ([769](#)), `TSQLQuery.ParamCheck` ([822](#))

### 34.27.6 TSQLStatement.Params

Synopsis: List of parameters.

Declaration: `Property Params :`

Visibility: published

Access:

Description: `Params` contains an item for each of the parameters in the SQL (769) statement (in the form `:ParamName`). The collection is filled automatically if the `ParamCheck` (769) property is `True`.

See also: SQL (769), `ParamCheck` (769), `ParseSQL` (769)

### 34.27.7 TSQLStatement.ParseSQL

Synopsis: Parse the SQL statement

Declaration: `Property ParseSQL :`

Visibility: published

Access:

Description: `ParseSQL` can be set to `False` to disable parsing of the SQL (769) property when it is set. The default behaviour (`ParseSQL=True`) is to parse the statement and detect what kind of SQL statement it is.

See also: SQL (769), `ParamCheck` (769)

### 34.27.8 TSQLStatement.SQL

Synopsis: The SQL statement to execute

Declaration: `Property SQL :`

Visibility: published

Access:

Description: `SQL` must be set to the SQL statement to execute. It must not be a statement that returns a result set. This is the statement that will be passed on to the database engine when `Prepare` (793) is called.

If `ParamCheck` (769) equals `True` (the default), the SQL statement can contain parameter names where literal values can occur, in the form `:ParamName`. Keywords or table names cannot be specified as parameters. If the underlying database engine supports it, the parameter support of the database will be used to transfer the values from the `Params` (769) collection. If not, it will be emulated. The `Params` collection is automatically populated when the SQL statement is set.

Some databases support executing multiple SQL statements in 1 call. Therefore, no attempt is done to ensure that `SQL` contains a single SQL statement. However, error reporting and the `RowsAffected` (795) function may be wrong in such a case.

See also: `ParseSQL` (769), `CheckParams` (769), `Params` (769), `Prepare` (793), `RowsAffected` (795)

### 34.27.9 TSQLStatement.Transaction

Synopsis: The transaction in which the SQL statement should be executed.

Declaration: `Property Transaction :`

Visibility: published

Access:

Description: `Transaction` should be set to a transaction connected to the instance of the database set in the `Database` (769) property. This must be set before `Prepare` (793) is called.

See also: `Database` (769), `Prepare` (793), `TSQLTransaction` (837)

## 34.28 TSQLTransaction

### 34.28.1 Description

`TSQLTransaction` represents the transaction in which one or more `TSQLQuery` (810) instances are doing their work. It contains the methods for committing or doing a rollback of the results of query. At least one `TSQLTransaction` must be used for each `TSQLConnection` (796) used in an application.

See also: `TSQLQuery` (810), `TSQLConnection` (796)

### 34.28.2 Method overview

Page	Method	Description
<a href="#">838</a>	<code>Commit</code>	Commit the transaction, end transaction context.
<a href="#">838</a>	<code>CommitRetaining</code>	Commit the transaction, retain transaction context.
<a href="#">837</a>	<code>Create</code>	Create a new transaction
<a href="#">838</a>	<code>Destroy</code>	Destroy transaction component
<a href="#">840</a>	<code>EndTransaction</code>	End the transaction
<a href="#">839</a>	<code>Rollback</code>	Roll back all changes made in the current transaction.
<a href="#">839</a>	<code>RollbackRetaining</code>	Roll back changes made in the transaction, keep transaction context.
<a href="#">839</a>	<code>StartTransaction</code>	Start a new transaction

### 34.28.3 Property overview

Page	Properties	Access	Description
<a href="#">840</a>	<code>Action</code>	rw	Currently unused in SQLDB
<a href="#">841</a>	<code>Database</code>		Database for which this component is handling connections
<a href="#">840</a>	<code>Handle</code>	r	Low-level transaction handle
<a href="#">841</a>	<code>Options</code>	rw	Transaction options
<a href="#">841</a>	<code>Params</code>	rw	Transaction parameters
<a href="#">840</a>	<code>SQLConnection</code>	rw	Database as <code>TSQLConnection</code>

### 34.28.4 TSQLTransaction.Create

Synopsis: Create a new transaction

Declaration: `constructor Create(AOwner: TComponent); Override`

Visibility: public

**Description:** `Create` creates a new `TSQLTransaction` instance, but does not yet start a transaction context.

### 34.28.5 TSQLTransaction.Destroy

**Synopsis:** Destroy transaction component

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

**Visibility:** `public`

**Description:** `Destroy` will close all datasets connected to it, prior to removing the object from memory.

### 34.28.6 TSQLTransaction.Commit

**Synopsis:** Commit the transaction, end transaction context.

**Declaration:** `procedure Commit;` `Override`

**Visibility:** `public`

**Description:** `Commit` commits an active transaction. The changes will be irreversibly written to the database.

After this, the transaction is deactivated and must be reactivated with the `StartTransaction` (839) method. To commit data while retaining an active transaction, execute `CommitRetaining` (838) instead.

**Errors:** Executing `Commit` when no transaction is active will result in an exception. A transaction must be started by calling `StartTransaction` (839). If the database backend reports an error, an exception is raised as well.

See also: `StartTransaction` (839), `CommitRetaining` (838), `Rollback` (839), `RollbackRetaining` (839)

### 34.28.7 TSQLTransaction.CommitRetaining

**Synopsis:** Commit the transaction, retain transaction context.

**Declaration:** `procedure CommitRetaining;` `Override`

**Visibility:** `public`

**Description:** `CommitRetaining` commits an active transaction. The changes will be irreversibly written to the database.

After this, the transaction is still active. To commit data and deactivate the transaction, execute `Commit` (838) instead.

**Errors:** Executing `CommitRetaining` when no transaction is active will result in an exception. A transaction must be started by calling `StartTransaction` (839). If the database backend reports an error, an exception is raised as well.

See also: `StartTransaction` (839), `Retaining` (838), `Rollback` (839), `RollbackRetaining` (839)

### 34.28.8 TSQLTransaction.Rollback

**Synopsis:** Roll back all changes made in the current transaction.

**Declaration:** `procedure Rollback; Override`

**Visibility:** public

**Description:** `Rollback` undoes all changes in the database since the start of the transaction. It can only be executed in an active transaction.

After this, the transaction is no longer active. To undo changes but keep an active transaction, execute `RollbackRetaining` (839) instead.

**Remark** Changes posted in datasets that are coupled to this transaction will not be undone in memory: these datasets must be reloaded from the database (using `Close` and `Open` to reload the data as it is in the database).

**Errors:** Executing `Rollback` when no transaction is active will result in an exception. A transaction must be started by calling `StartTransaction` (839). If the database backend reports an error, an exception is raised as well.

See also: `StartTransaction` (839), `CommitRetaining` (838), `Commit` (838), `RollbackRetaining` (839)

### 34.28.9 TSQLTransaction.RollbackRetaining

**Synopsis:** Roll back changes made in the transaction, keep transaction context.

**Declaration:** `procedure RollbackRetaining; Override`

**Visibility:** public

**Description:** `RollbackRetaining` undoes all changes in the database since the start of the transaction. It can only be executed in an active transaction.

After this, the transaction is kept in an active state. To undo changes and close the transaction, execute `Rollback` (839) instead.

**Remark** Changes posted in datasets that are coupled to this transaction will not be undone in memory: these datasets must be reloaded from the database (using `Close` and `Open` to reload the data as it is in the database).

**Errors:** Executing `RollbackRetaining` when no transaction is active will result in an exception. A transaction must be started by calling `StartTransaction` (839). If the database backend reports an error, an exception is raised as well.

See also: `StartTransaction` (839), `Commit` (838), `Rollback` (839), `CommitRetaining` (838)

### 34.28.10 TSQLTransaction.StartTransaction

**Synopsis:** Start a new transaction

**Declaration:** `procedure StartTransaction; Override`

**Visibility:** public

**Description:** `StartTransaction` starts a new transaction context. All changes written to the database must be confirmed with a `Commit` (838) or can be undone with a `Rollback` (839) call.

Calling `StartTransaction` is equivalent to setting `Active` to `True`.

**Errors:** If `StartTransaction` is called while the transaction is still active, an exception will be raised.



See also: [StartTransaction \(839\)](#), [Commit \(838\)](#), [Rollback \(839\)](#), [CommitRetaining \(838\)](#), [EndTransaction \(840\)](#)

### 34.28.11 TSQLTransaction.EndTransaction

Synopsis: End the transaction

Declaration: `procedure EndTransaction; Override`

Visibility: `public`

Description: `EndTransaction` is equivalent to [RollBack \(839\)](#).

See also: [RollBack \(839\)](#)

### 34.28.12 TSQLTransaction.Handle

Synopsis: Low-level transaction handle

Declaration: `Property Handle : Pointer`

Visibility: `public`

Access: `Read`

Description: `Handle` is the low-level transaction handle object. It must not be used in application code. The actual type of this object depends on the type of [TSQLConnection \(796\)](#) descendent.

### 34.28.13 TSQLTransaction.SQLConnection

Synopsis: Database as [TSQLConnection](#)

Declaration: `Property SQLConnection : TSQLConnection`

Visibility: `public`

Access: `Read,Write`

Description: `SQLConnection` returns or sets the `Database` property, typecasted to [TSQLConnection](#).

See also: [TSQLConnection \(796\)](#)

### 34.28.14 TSQLTransaction.Action

Synopsis: Currently unused in SQLDB

Declaration: `Property Action : TCommitRollbackAction`

Visibility: `published`

Access: `Read,Write`

Description: `Action` is currently unused in SQLDB.

### 34.28.15 TSQLTransaction.Database

Synopsis: Database for which this component is handling connections

Declaration: `Property Database :`

Visibility: published

Access:

Description: `Database` should be set to the particular `TSQLConnection` (796) instance this transaction is handling transactions in. All datasets connected to this transaction component must have the same value for their `Database` (818) property.

See also: `TSQLQuery.Database` (818), `TSQLConnection` (796)

### 34.28.16 TSQLTransaction.Params

Synopsis: Transaction parameters

Declaration: `Property Params : TStringList`

Visibility: published

Access: Read,Write

Description: `Params` can be used to set connection-specific parameters in the form of `Key=Value` pairs. The contents of this property therefor depends on the type of connection.

See also: `TSQLConnection` (796)

### 34.28.17 TSQLTransaction.Options

Synopsis: Transaction options

Declaration: `Property Options : TSQLTransactionOptions`

Visibility: published

Access: Read,Write

Description: `Options` can be used to control the behaviour of `SQLDB` for this transaction.

**stoUseImplicit** Use the implicit transaction support of the DB engine. This means that no explicit transaction start and stop commands will be sent to the server when the `Commit` or `Rollback` methods are called (effectively making them a no-op at the DB level).

**stoExplicitStart** When set, whenever an SQL statement is executed, the transaction must have been started explicitly. Default behaviour is that the `TSQLStatement` (834) or `TSQLQuery` (810) start the transaction as needed.

See also: `TSQLStatement` (834), `TSQLQuery` (810)

# Chapter 35

## Reference for unit 'SQLTypes'

### 35.1 Used units

Table 35.1: Used units by unit 'SQLTypes'

Name	Page
Classes	??
sysutils	??

### 35.2 Constants, types and variables

#### 35.2.1 Types

```
TDBEventType = (detCustom, detPrepare, detExecute, detFetch, detCommit
,
detRollBack, detParamValue, detActualSQL)
```

Table 35.2: Enumeration values for type TDBEventType

Value	Explanation
detActualSQL	Actual SQL as sent to engine message
detCommit	Transaction Commit message
detCustom	Custom event message
detExecute	SQLExecute message
detFetch	Fetch data message
detParamValue	Parameter name and value message
detPrepare	SQL prepare message
detRollBack	Transaction rollback message

TDBEventType describes the type of a database event message as generated by TSQLConnection (842) through the TSQLConnection.OnLog (842) event.

TDBEventTypes = Set of TDBEventType

TDBEventTypes is a set of TDBEventType (842) values, which is used to filter the set of event messages that should be sent. The TSQLConnection.LogEvents (842) property determines which events a particular connection will send.

Array = Array[0..1] of Char

TQuoteChars is an array of characters that describes the used delimiters for string values.

```
TSchemaType = (stNoSchema, stTables, stSysTables, stProcedures, stColumns
,
                stProcedureParams, stIndexes, stPackages, stSchemata
,
                stSequences)
```

Table 35.3: Enumeration values for type TSchemaType

Value	Explanation
stColumns	Columns in a table
stIndexes	Indexes for a table
stNoSchema	No schema
stPackages	Packages (for databases that support them)
stProcedureParams	Parameters for a stored procedure
stProcedures	Stored procedures in database
stSchemata	List of schemas in database(s) (for databases that support them)
stSequences	Sequences (for databases that support them)
stSysTables	System tables in database
stTables	User Tables in database

TSchemaType describes which schema information to retrieve in the TCustomSQLQuery.SetSchemaInfo (842) call. Depending on its value, the result set of the dataset will have different fields, describing the requested schema data. The result data will always have the same structure.

```
TStatementType = (stUnknown, stSelect, stInsert, stUpdate, stDelete, stDDL
,
                  stGetSegment, stPutSegment, stExecProcedure,
stStartTrans, stCommit, stRollback, stSelectForUpd)
```

Table 35.4: Enumeration values for type TStatementType

Value	Explanation
stCommit	The statement commits a transaction
stDDL	The statement is a SQL DDL (Data Definition Language) statement
stDelete	The statement is a SQL DELETE statement
stExecProcedure	The statement executes a stored procedure
stGetSegment	The statement is a SQL get segment statement
stInsert	The statement is a SQL INSERT statement
stPutSegment	The statement is a SQL put segment statement
stRollback	The statement rolls back a transaction
stSelect	The statement is a SQL SELECT statement
stSelectForUpd	The statement selects data for update
stStartTrans	The statement starts a transaction
stUnknown	The statement type could not be detected.
stUpdate	The statement is a SQL UPDATE statement

TStatementType describes the kind of SQL statement that was entered in the SQL property of a TSQLQuery (842) component.

## 35.3 TSqlObjectIdentifier

### 35.3.1 Description

TSqlObjectIdentifier is a class that represents an SQL identifier in a database. It has 2 parts: the schema name and the object name.

See also: TSqlObjectIdentifierList (842), #fcl.sqlldb.TSQLConnection.GetObjectNames (799)

### 35.3.2 Method overview

Page	Method	Description
844	Create	
845	FullName	Return the full name of the object

### 35.3.3 Property overview

Page	Properties	Access	Description
845	ObjectName	rw	Name of the object in the database
845	SchemaName	rw	Schema name

### 35.3.4 TSqlObjectIdentifier.Create

Synopsis:

```
Declaration: constructor Create (ACollection: TSqlObjectIdentifierList;
                                const AObjectName: string; const ASchemaName: string)
```

Visibility: public

Description:

### 35.3.5 TSqLObjectIdentifier.FullName

Synopsis: Return the full name of the object

Declaration: `function FullName : string`

Visibility: public

Description: `FullName` returns the name of the object, prepended with the schema name if there is one. Both parts are separated by a dot (.) character.

See also: `TSqLObjectIdentifier.SchemaName` (845), `TSqLObjectIdentifier.ObjectName` (845)

### 35.3.6 TSqLObjectIdentifier.SchemaName

Synopsis: Schema name

Declaration: `Property SchemaName : string`

Visibility: public

Access: Read,Write

Description: `SchemaName` is the name of the schema in which the object is defined. This is only set if the database actually supports schemas, for other databases, it is empty.

See also: `TSqLObjectIdentifierList` (842), `#fcl.sqlldb.TSQLConnection.GetObjectNames` (799), `TSqLObjectIdentifier.ObjectName` (845)

### 35.3.7 TSqLObjectIdentifier.ObjectName

Synopsis: Name of the object in the database

Declaration: `Property ObjectName : string`

Visibility: public

Access: Read,Write

Description: `ObjectName` is the name of the object in the database. If the database supports schemas, then it must be combined with the `SchemaName` (842) property in order to create a unique name for the object.

See also: `TSqLObjectIdentifierList` (842), `#fcl.sqlldb.TSQLConnection.GetObjectNames` (799), `TSqLObjectIdentifier.SchemaName` (845)

## 35.4 TSqLObjectIdentifierList

### 35.4.1 Description

`TSqLObjectIdentifierList` is a list class holding a list of `TSqLObjectIdentifier` (844) elements. It is used in the `TSQLConnection.GetObjectNames` (799) to return the list of objects requested.

See also: `TSqLObjectIdentifier` (844), `#fcl.sqlldb.TSQLConnection.GetObjectNames` (799)

### 35.4.2 Method overview

Page	Method	Description
<a href="#">846</a>	AddIdentifier	Add an identifier to the list

### 35.4.3 Property overview

Page	Properties	Access	Description
<a href="#">846</a>	Identifiers	rw	Indexed access to all identifiers in the list.

### 35.4.4 TSqLObjectIdentifierList.AddIdentifier

Synopsis: Add an identifier to the list

Declaration: `function AddIdentifier : TSqLObjectIdentifier; Overload`  
`function AddIdentifier(const AObjectName: string;`  
`const ASchemaName: string) : TSqLObjectIdentifier`  
`; Overload`

Visibility: public

Description: `AddIdentifier` adds an identifier to the list with schema name `ASchemaName` and object name `AObjectName`. It returns the new identifier object. Both arguments are optional, their default value is the empty string.

No checking on duplicate entries is performed.

See also: `TSqLObjectIdentifier` ([842](#))

### 35.4.5 TSqLObjectIdentifierList.Identifiers

Synopsis: Indexed access to all identifiers in the list.

Declaration: `Property Identifiers[Index: Integer]: TSqLObjectIdentifier; default`

Visibility: public

Access: Read,Write

Description: `Identifiers` gives indexed access to all `TSQLObjectIdentifier` ([842](#)) objects in the list. Valid indexes run from 0 to `Count-1`. It is the default property of the list class.

See also: `TSQLObjectIdentifierList.AddIdentifier` ([846](#))

## Chapter 36

# Reference for unit 'streamcoll'

### 36.1 Used units

Table 36.1: Used units by unit 'streamcoll'

Name	Page
Classes	??
sysutils	??

### 36.2 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` (850) 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.

### 36.3 Procedures and functions

#### 36.3.1 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` (849) and returns the read value. The value cannot be read and written across systems that have different endian values.



See also: [ColReadDateTime \(848\)](#), [ColWriteBoolean \(849\)](#), [ColReadString \(849\)](#), [ColReadInteger \(848\)](#), [ColReadFloat \(848\)](#), [ColReadCurrency \(848\)](#)

### 36.3.2 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 (849)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColReadDateTime \(848\)](#), [ColReadBoolean \(847\)](#), [ColReadString \(849\)](#), [ColReadInteger \(848\)](#), [ColReadFloat \(848\)](#), [ColWriteCurrency \(849\)](#)

### 36.3.3 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 (849)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColWriteDateTime \(849\)](#), [ColReadBoolean \(847\)](#), [ColReadString \(849\)](#), [ColReadInteger \(848\)](#), [ColReadFloat \(848\)](#), [ColReadCurrency \(848\)](#)

### 36.3.4 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 (850)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColReadDateTime \(848\)](#), [ColReadBoolean \(847\)](#), [ColReadString \(849\)](#), [ColReadInteger \(848\)](#), [ColWriteFloat \(850\)](#), [ColReadCurrency \(848\)](#)

### 36.3.5 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` (850) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: `ColReadDateTime` (848), `ColReadBoolean` (847), `ColReadString` (849), `ColWriteInteger` (850), `ColReadFloat` (848), `ColReadCurrency` (848)

### 36.3.6 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` (850) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: `ColReadDateTime` (848), `ColReadBoolean` (847), `ColWriteString` (850), `ColReadInteger` (848), `ColReadFloat` (848), `ColReadCurrency` (848)

### 36.3.7 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` (847), `ColWriteString` (850), `ColWriteInteger` (850), `ColWriteCurrency` (849), `ColWriteDateTime` (849), `ColWriteFloat` (850)

### 36.3.8 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` (849), `ColWriteString` (850), `ColWriteInteger` (850), `ColWriteDateTime` (849), `ColWriteFloat` (850), `ColReadCurrency` (848)

### 36.3.9 ColWriteDateTime

**Synopsis:** Write a `TDateTime` value to stream

**Declaration:** `procedure ColWriteDateTime(S: TStream; AValue: TDateTime)`

**Visibility:** default

**Description:** `ColWriteDateTime` writes the `TDateTimeAValue` to the stream `S`.

See also: `ColReadDateTime` (848), `ColWriteBoolean` (849), `ColWriteString` (850), `ColWriteInteger` (850), `ColWriteFloat` (850), `ColWriteCurrency` (849)

### 36.3.10 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` (849), `ColWriteBoolean` (849), `ColWriteString` (850), `ColWriteInteger` (850), `ColReadFloat` (848), `ColWriteCurrency` (849)

### 36.3.11 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` (849), `ColWriteString` (850), `ColReadInteger` (848), `ColWriteCurrency` (849), `ColWriteDateTime` (849)

### 36.3.12 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` (849), `ColReadString` (849), `ColWriteInteger` (850), `ColWriteCurrency` (849), `ColWriteDateTime` (849), `ColWriteFloat` (850)

## 36.4 EStreamColl

### 36.4.1 Description

Exception raised when an error occurs when streaming the collection.

## 36.5 TStreamCollection

### 36.5.1 Description

`TStreamCollection` is a `TCollection` (??) descendent which implements 2 calls `LoadFromStream` (851) and `SaveToStream` (851) which load and save the contents of the collection to a stream.

The collection items must be descendents of the `TStreamCollectionItem` (852) class for the streaming to work correctly.

Note that the stream must be used to load collections of the same type.

See also: `TStreamCollectionItem` (852)

### 36.5.2 Method overview

Page	Method	Description
<a href="#">851</a>	LoadFromStream	Load the collection from a stream
<a href="#">851</a>	SaveToStream	Load the collection from the stream.

### 36.5.3 Property overview

Page	Properties	Access	Description
<a href="#">851</a>	Streaming	r	Indicates whether the collection is currently being written to stream

### 36.5.4 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` ([851](#)). 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` ([851](#))

### 36.5.5 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` ([851](#)). 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` ([851](#))

### 36.5.6 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` ([851](#)), `TStreamCollection.SaveToStream` ([851](#))

## 36.6 TStreamCollectionItem

### 36.6.1 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 (850) to save or load the item from the stream.

See also: TStreamCollection (850)

## Chapter 37

# Reference for unit 'streamex'

### 37.1 Used units

Table 37.1: Used units by unit 'streamex'

Name	Page
Classes	??
RtlConsts	??
sysutils	??

### 37.2 Overview

`streamex` implements some extensions to be used together with streams from the `classes` unit.

### 37.3 Constants, types and variables

#### 37.3.1 Constants

`BUFFER_SIZE = 4096`

Default buffer size for `TStreamReader`

`FILE_RIGHTS = 438`

Default file rights for `TStreamReader`

`MIN_BUFFER_SIZE = 128`

Minimum buffer size for `TStreamReader`

## 37.4 TBidirBinaryObjectReader

### 37.4.1 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.

See also: `TBidirBinaryObjectWriter` (854), `TDelphiReader` (855)

### 37.4.2 Property overview

Page	Properties	Access	Description
<a href="#">854</a>	Position	rw	Position in the stream

### 37.4.3 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` (855) class.

See also: `TDelphiReader` (855)

## 37.5 TBidirBinaryObjectWriter

### 37.5.1 Description

`TBidirBinaryObjectReader` is a class descendent from `TBinaryObjectWriter` (??), which implements the necessary support for BiDi data.

See also: `TBidirBinaryObjectWriter` (854), `TDelphiWriter` (856)

### 37.5.2 Property overview

Page	Properties	Access	Description
<a href="#">854</a>	Position	rw	Position in the stream

### 37.5.3 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` (856) class.

See also: `TDelphiWriter` (856)

## 37.6 TDelphiReader

### 37.6.1 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` (854) driver class.

See also: `TDelphiWriter` (856), `TBidirBinaryObjectReader` (854)

### 37.6.2 Method overview

Page	Method	Description
<a href="#">855</a>	<code>GetDriver</code>	Return the driver class as a <code>TBidirBinaryObjectReader</code> (854) class
<a href="#">855</a>	<code>Read</code>	Read data from stream
<a href="#">855</a>	<code>ReadStr</code>	Overrides the standard <code>ReadStr</code> method

### 37.6.3 Property overview

Page	Properties	Access	Description
<a href="#">856</a>	<code>Position</code>	rw	Position in the stream

### 37.6.4 TDelphiReader.GetDriver

Synopsis: Return the driver class as a `TBidirBinaryObjectReader` (854) class

Declaration: `function GetDriver : TBidirBinaryObjectReader`

Visibility: public

Description: `GetDriver` simply returns the used driver and typecasts it as `TBidirBinaryObjectReader` (854) class.

See also: `TBidirBinaryObjectReader` (854)

### 37.6.5 TDelphiReader.ReadStr

Synopsis: Overrides the standard `ReadStr` method

Declaration: `function ReadStr : string`

Visibility: public

Description: `ReadStr` makes sure the `TBidirBinaryObjectReader` (854) methods are used, to store additional information about the stream position when reading the strings.

See also: `TBidirBinaryObjectReader` (854)

### 37.6.6 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` (854) class when reading.

See also: `TBidirBinaryObjectReader` (854), `TDelphiReader.Position` (856)

### 37.6.7 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` (855)

## 37.7 TDelphiWriter

### 37.7.1 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` (854) driver class.

See also: `TDelphiReader` (855), `TBidirBinaryObjectWriter` (854)

### 37.7.2 Method overview

Page	Method	Description
857	<code>FlushBuffer</code>	Flushes the stream buffer
856	<code>GetDriver</code>	Return the driver class as a <code>TBidirBinaryObjectWriter</code> (854) class
857	<code>Write</code>	Write raw data to the stream
857	<code>WriteStr</code>	Write a string to the stream
857	<code>WriteValue</code>	Write value type

### 37.7.3 Property overview

Page	Properties	Access	Description
857	<code>Position</code>	<code>rw</code>	Position in the stream

### 37.7.4 TDelphiWriter.GetDriver

**Synopsis:** Return the driver class as a `TBidirBinaryObjectWriter` (854) class

**Declaration:** `function GetDriver : TBidirBinaryObjectWriter`

**Visibility:** `public`

**Description:** `GetDriver` simply returns the used driver and typecasts it as `TBidirBinaryObjectWriter` (854) class.

See also: `TBidirBinaryObjectWriter` (854)

### 37.7.5 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.

### 37.7.6 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.

### 37.7.7 TDelphiWriter.WriteString

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` (854) class methods, which update the position of the stream.

See also: `TBidirBinaryObjectWriter` (854)

### 37.7.8 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` (854) methods, which update the position of the stream.

See also: `TBidirBinaryObjectWriter` (854)

### 37.7.9 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` (854) instance used when streaming.

See also: `TBidirBinaryObjectWriter` (854)

## 37.8 TFileReader

### 37.8.1 Description

TFileReader is a TTextReader descendent that takes a file on disk as the source of text data.

See also: TStreamReader (865), TTextReader (869)

### 37.8.2 Method overview

Page	Method	Description
<a href="#">859</a>	Close	Close the file
<a href="#">858</a>	Create	Create a new instance of TFileReader for a disk file
<a href="#">858</a>	Destroy	Remove the TFileReader instance from memory.
<a href="#">859</a>	ReadLine	Read a line of text
<a href="#">859</a>	Reset	Reset the stream to its original position

### 37.8.3 TFileReader.Create

Synopsis: Create a new instance of TFileReader for a disk file

Declaration: constructor Create(const AFileName: TFileName; AMode: Word;  
                                   ARights: Cardinal; ABufferSize: Integer); Virtual  
 constructor Create(const AFileName: TFileName; AMode: Word;  
                                   ABufferSize: Integer); Virtual  
 constructor Create(const AFileName: TFileName; ABufferSize: Integer)  
                                   ; Virtual  
 constructor Create(const AFileName: TFileName); Virtual

Visibility: public

Description: Create initializes a TFileReader using the provided AFileName. It will allocate a buffer of ABufferSize bytes for faster reading of data. If no buffer size is specified, BUFFER\_SIZE ([853](#)) will be used. If the ABufferSize argument is less than MIN\_BUFFER\_SIZE ([853](#)), then MIN\_BUFFER\_SIZE bytes will be used.

The AMode can be used to specify the mode in which to open the file. This is one of the fmOpenRead and fmShare\* constants which can be used in a TFileStream constructor. The file must be opened for reading.

Errors: If AStream is Nil, an #rtl.sysutils.EArgumentException (??) exception will be raised.

See also: TStreamReader.Destroy ([866](#)), TFileStream (??), TFileStream.Create (??)

### 37.8.4 TFileReader.Destroy

Synopsis: Remove the TFileReader instance from memory.

Declaration: destructor Destroy; Override

Visibility: public

Description: Destroy closes the file and releases the buffer used to read data.

See also: TFileReader.Create ([858](#))

### 37.8.5 TFileReader.Reset

Synopsis: Reset the stream to its original position

Declaration: `procedure Reset; Override`

Visibility: `public`

Description: `Reset` sets the stream to its original position. This is the stream-specific implementation of the abstract `TTextReader.Reset` (870) method.

See also: `TTextReader.Reset` (870)

### 37.8.6 TFileReader.Close

Synopsis: Close the file

Declaration: `procedure Close; Override`

Visibility: `public`

Description: `Close` closes the text file. Any read operations after `Close` is called will fail. This is the stream-specific implementation of the abstract `TTextReader.Close` (870)

See also: `TTextReader.Close` (870)

### 37.8.7 TFileReader.ReadLine

Synopsis: Read a line of text

Declaration: `procedure ReadLine(out AString: string); Override; Overload`

Visibility: `public`

Description: `ReadLine` will read a line of text from the text data source. A line of text is delimited by a CRLF character pair, a LF character or a CR character. The line ending characters are not included in the string.

The method exists in 2 versions: one function where the line of text is returned as the function result, one procedure where the line of text is returned in the `AString` parameter.

This is the `TFileReader` specific implementation of the abstract `TTextReader.ReadLine` (870) method.

See also: `Eof` (871), `TTextReader.ReadLine` (870)

## 37.9 TStreamHelper

### 37.9.1 Description

`TStreamHelper` is a `TStream` (??) helper class which introduces some helper routines to read-/write multi-byte integer values in a way that is endianness-safe.

See also: `TStream` (??)

**37.9.2 Method overview**

Page	Method	Description
<a href="#">864</a>	<code>ReadByte</code>	
<a href="#">863</a>	<code>ReadDouble</code>	Read a double-precision floating point value from the stream
<a href="#">864</a>	<code>ReadDWord</code>	
<a href="#">862</a>	<code>ReadDWordBE</code>	Read a DWord from the stream, big endian
<a href="#">860</a>	<code>ReadDWordLE</code>	Read a DWord from the stream, little endian
<a href="#">864</a>	<code>ReadQWord</code>	
<a href="#">862</a>	<code>ReadQWordBE</code>	Read a QWord from the stream, big endian
<a href="#">861</a>	<code>ReadQWordLE</code>	Read a QWord from the stream, little endian
<a href="#">863</a>	<code>ReadSingle</code>	Read a single-precision floating point value from the stream
<a href="#">864</a>	<code>ReadWord</code>	
<a href="#">862</a>	<code>ReadWordBE</code>	Read a Word from the stream, big endian
<a href="#">860</a>	<code>ReadWordLE</code>	Read a Word from the stream, little endian
<a href="#">865</a>	<code>WriteByte</code>	
<a href="#">864</a>	<code>WriteDouble</code>	Write a double-precision floating point value to the stream
<a href="#">865</a>	<code>WriteDWord</code>	
<a href="#">863</a>	<code>WriteDWordBE</code>	Write a DWord value, big endian
<a href="#">861</a>	<code>WriteDWordLE</code>	Write a DWord value, little endian
<a href="#">865</a>	<code>WriteQWord</code>	
<a href="#">863</a>	<code>WriteQWordBE</code>	Write a QWord value, big endian
<a href="#">861</a>	<code>WriteQWordLE</code>	Write a QWord value, little endian
<a href="#">864</a>	<code>WriteSingle</code>	Write a single-precision floating point value to the stream
<a href="#">865</a>	<code>WriteWord</code>	
<a href="#">862</a>	<code>WriteWordBE</code>	Write a word value, big endian
<a href="#">861</a>	<code>WriteWordLE</code>	Write a word value, little endian

**37.9.3 TStreamHelper.ReadWordLE**

Synopsis: Read a Word from the stream, little endian

Declaration: `function ReadWordLE : Word`

Visibility: default

Description: `ReadWordLE` reads a word from the stream, little-endian (LSB first).

Errors: If not enough data is available an `EReadError` exception is raised.

See also: `ReadDWordLE` ([853](#)), `ReadQWordLE` ([853](#)), `WriteWordLE` ([853](#))

**37.9.4 TStreamHelper.ReadDWordLE**

Synopsis: Read a DWord from the stream, little endian

Declaration: `function ReadDWordLE : dword`

Visibility: default

Description: `ReadWordLE` reads a DWord from the stream, little-endian (LSB first).

Errors: If not enough data is available an `EReadError` exception is raised.

See also: `ReadWordLE` ([853](#)), `ReadQWordLE` ([853](#)), `WriteDWordLE` ([853](#))

### 37.9.5 TStreamHelper.ReadQWordLE

Synopsis: Read a QWord from the stream, little endian

Declaration: `function ReadQWordLE : QWord`

Visibility: default

Description: `ReadWordLE` reads a QWord from the stream, little-endian (LSB first).

Errors: If not enough data is available an `EReadError` exception is raised.

See also: `ReadWordLE` ([853](#)), `ReadDWordLE` ([853](#)), `WriteQWordLE` ([853](#))

### 37.9.6 TStreamHelper.WriteWordLE

Synopsis: Write a word value, little endian

Declaration: `procedure WriteWordLE(w: Word)`

Visibility: default

Description: `WriteWordLE` writes a Word-sized value to the stream, little-endian (LSB first).

Errors: If not all data (2 bytes) can be written, an `EWriteError` exception is raised.

See also: `ReadWordLE` ([853](#)), `WriteDWordLE` ([853](#)), `WriteQWordLE` ([853](#))

### 37.9.7 TStreamHelper.WriteDWordLE

Synopsis: Write a DWord value, little endian

Declaration: `procedure WriteDWordLE(dw: dword)`

Visibility: default

Description: `WriteDWordLE` writes a DWord-sized value to the stream, little-endian (LSB first).

Errors: If not all data (4 bytes) can be written, an `EWriteError` exception is raised.

See also: `ReadDWordLE` ([853](#)), `WriteWordLE` ([853](#)), `WriteQWordLE` ([853](#))

### 37.9.8 TStreamHelper.WriteQWordLE

Synopsis: Write a QWord value, little endian

Declaration: `procedure WriteQWordLE(dq: QWord)`

Visibility: default

Description: `WriteQWordLE` writes a QWord-sized value to the stream, little-endian (LSB first).

Errors: If not all data (8 bytes) can be written, an `EWriteError` exception is raised.

See also: `ReadQWordLE` ([853](#)), `WriteDWordLE` ([853](#)), `WriteWordLE` ([853](#))

### 37.9.9 TStreamHelper.ReadWordBE

Synopsis: Read a Word from the stream, big endian

Declaration: `function ReadWordBE : Word`

Visibility: default

Description: `ReadWordBE` reads a word from the stream, big-endian (MSB first).

Errors: If not enough data is available an `EReadError` exception is raised.

See also: `ReadDWordBE` (853), `ReadQWordBE` (853), `WriteWordBE` (853)

### 37.9.10 TStreamHelper.ReadDWordBE

Synopsis: Read a DWord from the stream, big endian

Declaration: `function ReadDWordBE : dword`

Visibility: default

Description: `ReadWordBE` reads a DWord from the stream, big-endian (MSB first).

Errors: If not enough data is available an `EReadError` exception is raised.

See also: `ReadWordBE` (853), `ReadQWordBE` (853), `WriteDWordBE` (853)

### 37.9.11 TStreamHelper.ReadQWordBE

Synopsis: Read a QWord from the stream, big endian

Declaration: `function ReadQWordBE : QWord`

Visibility: default

Description: `ReadWordBE` reads a QWord from the stream, big-endian (MSB first).

Errors: If not enough data is available an `EReadError` exception is raised.

See also: `ReadWordBE` (853), `ReadDWordBE` (853), `WriteQWordBE` (853)

### 37.9.12 TStreamHelper.WriteWordBE

Synopsis: Write a word value, big endian

Declaration: `procedure WriteWordBE(w: Word)`

Visibility: default

Description: `WriteWordBE` writes a Word-sized value to the stream, big-endian (MSB first).

Errors: If not all data (2 bytes) can be written, an `EWriteError` exception is raised.

See also: `ReadWordBE` (853), `WriteDWordBE` (853), `WriteQWordBE` (853)

### 37.9.13 TStreamHelper.WriteDWordBE

Synopsis: Write a DWord value, big endian

Declaration: `procedure WriteDWordBE(dw: dword)`

Visibility: default

Description: `WriteDWordBE` writes a DWord-sized value to the stream, big-endian (MSB first).

Errors: If not all data (4 bytes) can be written, an `EWriteError` exception is raised.

See also: `ReadDWordBE` (853), `WriteWordBE` (853), `WriteQWordBE` (853)

### 37.9.14 TStreamHelper.WriteQWordBE

Synopsis: Write a QWord value, big endian

Declaration: `procedure WriteQWordBE(dq: QWord)`

Visibility: default

Description: `WriteQWordBE` writes a QWord-sized value to the stream, big-endian (MSB first).

Errors: If not all data (8 bytes) can be written, an `EWriteError` exception is raised.

See also: `ReadQWordBE` (853), `WriteDWordBE` (853), `WriteWordBE` (853)

### 37.9.15 TStreamHelper.ReadSingle

Synopsis: Read a single-precision floating point value from the stream

Declaration: `function ReadSingle : Single`

Visibility: default

Description: `ReadSingle` reads a single-precision floating point value from the stream and returns the value.  
No endianness corrections are performed.

Errors: If the end of stream is reached before all necessary bytes can be read, an `EReadError` (??) exception is raised.

See also: `TStreamHelper.ReadDouble` (863), `TStreamHelper.WriteSingle` (864)

### 37.9.16 TStreamHelper.ReadDouble

Synopsis: Read a double-precision floating point value from the stream

Declaration: `function ReadDouble : Double`

Visibility: default

Description: `ReadDouble` reads a double-precision floating point value from the stream and returns the value.  
No endianness corrections are performed.

Errors: If the end of stream is reached before all necessary bytes can be read, an `EReadError` (??) exception is raised.

See also: `TStreamHelper.ReadSingle` (863), `TStreamHelper.WriteDouble` (864)



### 37.9.17 TStreamHelper.WriteSingle

Synopsis: Write a single-precision floating point value to the stream

Declaration: `procedure WriteSingle(s: Single)`

Visibility: default

Description: `WriteSingle` writes the single-precision floating point value *S* to the stream. No endianness corrections are performed.

Errors: If not all bytes can be written, an `EWriteError` (??) exception is raised.

See also: `TStreamHelper.ReadSingle` (863), `TStreamHelper.WriteDouble` (864)

### 37.9.18 TStreamHelper.WriteDouble

Synopsis: Write a double-precision floating point value to the stream

Declaration: `procedure WriteDouble(d: Double)`

Visibility: default

Description: `WriteDouble` writes the double-precision floating point value *D* to the stream. No endianness corrections are performed.

Errors: If not all bytes can be written, an `EWriteError` (??) exception is raised.

See also: `TStreamHelper.ReadDouble` (863), `TStreamHelper.WriteSingle` (864)

### 37.9.19 TStreamHelper.ReadByte

Declaration: `function ReadByte : Byte`

Visibility: default

### 37.9.20 TStreamHelper.ReadWord

Declaration: `function ReadWord : Word`

Visibility: default

### 37.9.21 TStreamHelper.ReadDWord

Declaration: `function ReadDWord : DWord`

Visibility: default

### 37.9.22 TStreamHelper.ReadQWord

Declaration: `function ReadQWord : QWord`

Visibility: default

**37.9.23 TStreamHelper.WriteByte**

Declaration: `procedure WriteByte(b: Byte)`

Visibility: default

**37.9.24 TStreamHelper.WriteWord**

Declaration: `procedure WriteWord(b: Word)`

Visibility: default

**37.9.25 TStreamHelper.WriteDWord**

Declaration: `procedure WriteDWord(b: DWord)`

Visibility: default

**37.9.26 TStreamHelper.WriteQWord**

Declaration: `procedure WriteQWord(b: QWord)`

Visibility: default

**37.10 TStreamReader****37.10.1 Description**

`TStreamReader` is a `TTextReader` descendent that takes a stream as the source of text data. It can free the stream and the buffer size to use for reading data can be set.

See also: `TTextReader` ([869](#))

**37.10.2 Method overview**

Page	Method	Description
<a href="#">866</a>	<code>Close</code>	Close and possibly free the stream
<a href="#">865</a>	<code>Create</code>	Create a new instance of <code>TStreamReader</code> from a stream
<a href="#">866</a>	<code>Destroy</code>	Destroy the <code>TStreamReader</code> instance.
<a href="#">867</a>	<code>ReadLine</code>	Read a line of text
<a href="#">866</a>	<code>Reset</code>	Reset the stream to its original position

**37.10.3 Property overview**

Page	Properties	Access	Description
<a href="#">867</a>	<code>BaseStream</code>	r	The stream with the text data
<a href="#">867</a>	<code>OwnsStream</code>	rw	Should the stream be freed on close

**37.10.4 TStreamReader.Create**

Synopsis: Create a new instance of `TStreamReader` from a stream

**Declaration:** `constructor Create(AStream: TStream; ABufferSize: Integer;  
                                  AOwnsStream: Boolean); Virtual  
          constructor Create(AStream: TStream); Virtual`

**Visibility:** public

**Description:** `Create` initializes a `TStreamReader` using the provided `AStream`. It will allocate a buffer of `ABufferSize` bytes for faster reading of data. If no buffer size is specified, `BUFFER_SIZE` (853) will be used. If the `ABufferSize` argument is less than `MIN_BUFFER_SIZE` (853), then `MIN_BUFFER_SIZE` bytes will be used.

If `AOwnsStream` is true, the stream will be freed when the `TStreamReader` instance is freed. If omitted, its value is assumed to be `False`.

**Errors:** If `AStream` is `Nil`, an `#rtl.sysutils.EArgumentException` (??) exception will be raised.

See also: `TStreamReader.Destroy` (866), `TStream` (??)

### 37.10.5 TStreamReader.Destroy

**Synopsis:** Destroy the `TStreamReader` instance.

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

**Visibility:** public

**Description:** `Destroy` frees the resources taken by the buffer, and frees the source stream (`BaseStream` (867)) if `OwnsStream` is `True`.

See also: `TStreamReader.OwnsStream` (867), `TStreamReader.Create` (865), `TStreamReader.BaseStream` (867)

### 37.10.6 TStreamReader.Reset

**Synopsis:** Reset the stream to its original position

**Declaration:** `procedure Reset; Override`

**Visibility:** public

**Description:** `Reset` sets the stream to its original position. This is the stream-specific implementation of the abstract `TTextReader.Reset` (870) method.

**Errors:** If the source stream (`TStreamReader.BaseStream` (867)) is not seekable, then this method may raise an exception.

See also: `TStreamReader.BaseStream` (867)

### 37.10.7 TStreamReader.Close

**Synopsis:** Close and possibly free the stream

**Declaration:** `procedure Close; Override`

**Visibility:** public

**Description:** `Close` closed the text data stream. It will free the source stream if `OwnsStream` is `True`

See also: `TStreamReader.BaseStream` (867), `TStreamReader.OwnsStream` (867)

### 37.10.8 TStreamReader.ReadLine

Synopsis: Read a line of text

Declaration: `procedure ReadLine(out AString: string); Override; Overload`

Visibility: public

Description: `ReadLine` will read a line of text from the text data source. A line of text is delimited by a CRLF character pair, a LF character or a CR character. The line ending characters are not included in the string.

The method exists in 2 versions: one function where the line of text is returned as the function result, one procedure where the line of text is returned in the `AString` parameter.

This is the `TStreamReader` specific implementation of the abstract `TTextReader.ReadLine` (870) method.

See also: `Eof` (871), `TTextReader.ReadLine` (870)

### 37.10.9 TStreamReader.BaseStream

Synopsis: The stream with the text data

Declaration: `Property BaseStream : TStream`

Visibility: public

Access: Read

Description: `BaseStream` is the stream that was passed to the `TStreamReader` instance in the `TStreamReader.Create` (865) call.

Manipulating the stream between calls to `TStreamReader.ReadLine` (867) is not allowed, it will lead to wrong data being read from the stream.

See also: `TStreamReader.Create` (865), `TStreamReader.OwnsStream` (867)

### 37.10.10 TStreamReader.OwnsStream

Synopsis: Should the stream be freed on close

Declaration: `Property OwnsStream : Boolean`

Visibility: public

Access: Read, Write

Description: `OwnsStream` determines whether the stream `TStreamReader.BaseStream` (867) must be freed when `TStreamReader.Close` (866) is called or when the `TStreamReader` instance is destroyed.

See also: `TStreamReader.BaseStream` (867), `TStreamReader.Close` (866)

## 37.11 TStringReader

### 37.11.1 Description

`TStreamReader` is a `TTextReader` descendent that takes a single string as the source of text data.

See also: `TTextReader` (869)

**37.11.2 Method overview**

Page	Method	Description
<a href="#">869</a>	Close	Close and possibly free the stream
<a href="#">868</a>	Create	Create a new instance of <code>TStreamReader</code> from a string
<a href="#">868</a>	Destroy	Free the <code>TStringReader</code> instance.
<a href="#">869</a>	ReadLine	Read a line of text
<a href="#">868</a>	Reset	Reset the stream to its original position

**37.11.3 TStringReader.Create**

Synopsis: Create a new instance of `TStreamReader` from a string

Declaration: `constructor Create(const AString: string; ABufferSize: Integer)`  
`; Virtual`  
`constructor Create(const AString: string); Virtual`

Visibility: `public`

Description: `Create` initializes a `TStringReader` instance using the provided `AString`. It will allocate a buffer of `ABufferSize` bytes for faster reading of data. If no buffer size is specified, `BUFFER_SIZE` ([853](#)) will be used. If the `ABufferSize` argument is less than `MIN_BUFFER_SIZE` ([853](#)), then `MIN_BUFFER_SIZE` bytes will be used.

Errors: If `AStream` is `Nil`, an `#rtl.sysutils.EArgumentException` (??) exception will be raised.

See also: `TStreamReader.Destroy` ([866](#)), `TStream` (??)

**37.11.4 TStringReader.Destroy**

Synopsis: Free the `TStringReader` instance.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` deallocates all resources for the `TStringReader` instance.

See also: `TStringReader.Create` ([868](#))

**37.11.5 TStringReader.Reset**

Synopsis: Reset the stream to its original position

Declaration: `procedure Reset; Override`

Visibility: `public`

Description: `Reset` sets the stream to its original position. This is the string-specific implementation of the abstract `TTextReader.Reset` ([870](#)) method.

See also: `TStreamReader.BaseStream` ([867](#))

### 37.11.6 TStringReader.Close

Synopsis: Close and possibly free the stream

Declaration: `procedure Close; Override`

Visibility: `public`

Description: `Close` closes the text reader. This is the string-specific implementation of the abstract `TTextReader.Close` (870) method.

See also: `TTextReader.Close` (870)

### 37.11.7 TStringReader.ReadLine

Synopsis: Read a line of text

Declaration: `procedure ReadLine(out AString: string); Override; Overload`

Visibility: `public`

Description: `ReadLine` will read a line of text from the text data source. A line of text is delimited by a CRLF character pair, a LF character or a CR character. The line ending characters are not included in the string.

The method exists in 2 versions: one function where the line of text is returned as the function result, one procedure where the line of text is returned in the `AString` parameter.

This is the `TStringReader` specific implementation of the abstract `TTextReader.ReadLine` (870) method.

See also: `Eof` (871), `TTextReader.ReadLine` (870)

## 37.12 TTextReader

### 37.12.1 Description

`TTextReader` is an abstract class that provides a line-oriented reading API. It allows to read data from streams or memory blocks as if one was using regular pascal `Read` or `ReadLn` operations: the `ReadLine` (870) procedure. Several descendents of this class exist which implement the reader interface for several sources of text data: `TStreamReader` (865), `TFileReader` (858), `TStringReader` (867).

See also: `TStreamReader` (865), `TFileReader` (858), `TStringReader` (867), `ReadLine` (870)

### 37.12.2 Method overview

Page	Method	Description
<a href="#">870</a>	<code>Close</code>	Close the text data stream
<a href="#">870</a>	<code>Create</code>	Instantiate a new instance.
<a href="#">870</a>	<code>ReadLine</code>	Read a line of text
<a href="#">870</a>	<code>Reset</code>	Reset the reader to the start position

### 37.12.3 Property overview

Page	Properties	Access	Description
<a href="#">871</a>	<code>Eof</code>	<code>r</code>	Check whether the end of the text data is returned.

### 37.12.4 TTextReader.Create

Synopsis: Instantiate a new instance.

Declaration: `constructor Create; Virtual`

Visibility: `public`

Description: `Create` does nothing in `TTextReader`.

See also: `ReadLine` ([870](#))

### 37.12.5 TTextReader.Reset

Synopsis: Reset the reader to the start position

Declaration: `procedure Reset; Virtual; Abstract`

Visibility: `public`

Description: `Reset` resets the position to the start of the text data.

This is an abstract call which must be implemented by descendents.

See also: `TTextReader.Close` ([870](#)), `TTextReader.ReadLine` ([870](#))

### 37.12.6 TTextReader.Close

Synopsis: Close the text data stream

Declaration: `procedure Close; Virtual; Abstract`

Visibility: `public`

Description: `Close` closes the data stream. No `ReadLine` ([870](#)) call can be performed after a call to `Close`.

See also: `TTextReader.Reset` ([870](#)), `TTextReader.ReadLine` ([870](#))

### 37.12.7 TTextReader.ReadLine

Synopsis: Read a line of text

Declaration: `procedure ReadLine(out AString: string); Virtual; Abstract; Overload`  
`function ReadLine : string; Overload`

Visibility: `public`

Description: `ReadLine` will read a line of text from the text data source. A line of text is delimited by a CRLF character pair, a LF character or a CR character. The line ending characters are not included in the string.

The method exists in 2 versions: one function where the line of text is returned as the function result, one procedure where the line of text is returned in the `AString` parameter.

See also: `Eof` ([871](#))

### 37.12.8 TTextReader.Eof

Synopsis: Check whether the end of the text data is returned.

Declaration: `Property Eof : Boolean`

Visibility: `public`

Access: `Read`

Description: `Eof` is `True` if no more data is available for reading. If there is still data, then it is `False`.

See also: `TTextReader.ReadLine` ([870](#))

## 37.13 TWindowedStream

### 37.13.1 Description

`TWindowedStream` is a `TStream` ([853](#)) descendent that can be used to provide a window on the data of another stream. The position and size of the window can be determined in the constructor, and the stream will behave as a normal stream. The actual reading (or writing) will happen on the source stream.

If the source stream has some limitations (e.g. no `Seek` ([??](#))) then the `TWindowedStream` will inherit these limitations.

The `TWindowedStream` will keep track of the last position it used, and will attempt to restore it if it was modified between calls to `Read` and `Write`.

See also: `TStream` ([853](#)), `TWindowedStream.Create` ([871](#))

### 37.13.2 Method overview

Page	Method	Description
<a href="#">871</a>	<code>Create</code>	Initialize a new instance of <code>TWindowedStream</code>
<a href="#">872</a>	<code>Destroy</code>	Destroy the <code>TWindowedStream</code> instance.
<a href="#">872</a>	<code>Read</code>	Read data from the stream
<a href="#">872</a>	<code>Seek</code>	Reposition the stream
<a href="#">872</a>	<code>Write</code>	Read data to the stream

### 37.13.3 TWindowedStream.Create

Synopsis: Initialize a new instance of `TWindowedStream`

Declaration: `constructor Create(aStream: TStream; const aSize: Int64;  
const aPositionHere: Int64)`

Visibility: `public`

Description: `Create` will create a new instance of `TWindowedStream`. The source stream `aStream` must be specified, as well as the start `aPositionHere` position of the window in the source stream and the size of the window `aSize`.

No checks on the validity of `aPositionHere` and `aSize` are done.



### 37.13.4 TWindowedStream.Destroy

Synopsis: Destroy the TWindowedStream instance.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` simply calls the inherited `destroy`, it removes the TWindowedStream instance from memory. The source stream is not freed.

See also: TWindowedStream.Create ([871](#))

### 37.13.5 TWindowedStream.Read

Synopsis: Read data from the stream

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

Visibility: `public`

Description: `Read` attempts to read data from the stream. It will attempt to restore the source stream position if it was changed since the last read, write or seek operation. It then attempts to read `aCount` bytes from the source stream into `aBuffer` and returns the number of actually read bytes. `TWindowedStream.Read` will only read as much data as the window allows, even if the source stream has more data available.

See also: TWindowedStream.Write ([872](#)), TWindowedStream.Seek ([872](#))

### 37.13.6 TWindowedStream.Write

Synopsis: Read data to the stream

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

Visibility: `public`

Description: `Write` attempts to write data to the stream. It will attempt to restore the source stream position if it was changed since the last read, write or seek operation. It then attempts to write `aCount` bytes from `aBuffer` to the source stream and returns the number of actually written bytes.

Errors: `TWindowedStream.Write` will raise an `EWriteError` exception if an attempt is made to write more bytes than will fit in the window, even if the source stream has more room available.

See also: TWindowedStream.Read ([872](#)), TWindowedStream.Seek ([872](#))

### 37.13.7 TWindowedStream.Seek

Synopsis: Reposition the stream

Declaration: `function Seek(const aOffset: Int64; aOrigin: TSeekorigin) : Int64; Override`

Visibility: `public`

Description: `Seek` will reposition the windowed stream based on `aOffset` and `aOrigin`. It will interpret `aOrigin` and `aOffset` relative to the position and size of the window, and will then call `Seek` on the source stream. It will return the new position in the windowed stream.

**Errors:** If the source stream does not support seek operations, an exception may be raised. If the combination of `AOrigin` and `aOffset` falls outside the valid window of the stream, an `EReadError` exception is raised.

See also: `TWindowedStream.Read` ([872](#)), `TWindowedStream.Write` ([872](#))

## Chapter 38

# Reference for unit 'StreamIO'

### 38.1 Used units

Table 38.1: Used units by unit 'StreamIO'

Name	Page
Classes	??
sysutils	??

### 38.2 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.

### 38.3 Procedures and functions

#### 38.3.1 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:** `TStream` (??), `GetStream` ([875](#))

### 38.3.2 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` ([874](#)).

Errors: An invalid class reference will be returned if the file was not associated with a stream.

See also: `AssignStream` ([874](#)), `TStream` (??)

## Chapter 39

# Reference for unit 'syncobjs'

### 39.1 Used units

Table 39.1: Used units by unit 'syncobjs'

Name	Page
sysutils	??

### 39.2 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` ([877](#)) class is a wrapper around low-level critical section routines (semaphores or mutexes). The `TEventObject` ([880](#)) class can be used to send messages between threads (also known as conditional variables in POSIX threads).

### 39.3 Constants, types and variables

#### 39.3.1 Constants

```
INFINITE = Cardinal(- 1)
```

Constant denoting an infinite timeout.

#### 39.3.2 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` ([880](#)) class.

`TEventHandle = Pointer`

`TEventHandle` is an opaque type and should not be used in user code.

`TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError)`

Table 39.2: 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.

## 39.4 ELockException

### 39.4.1 Description

`ELockException` is provided for Delphi compatibility. It is not used in FPC.

See also: `ESyncObjectException` ([877](#)), `ELockRecursionException` ([877](#))

## 39.5 ELockRecursionException

### 39.5.1 Description

`ELockRecursionException` is provided for Delphi compatibility. It is not used in FPC.

See also: `ESyncObjectException` ([877](#)), `ELockException` ([877](#))

## 39.6 ESyncObjectException

### 39.6.1 Description

`ESyncObjectException` is used in the constructor of `TEventObject` ([880](#)) to indicate failure to create a basic event.

See also: `TEventObject` ([880](#)), `ELockRecursionException` ([877](#)), `ELockException` ([877](#))

## 39.7 TCriticalSection

### 39.7.1 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.

See also: Acquire ([878](#)), Release ([878](#))

### 39.7.2 Method overview

Page	Method	Description
<a href="#">878</a>	Acquire	Enter the critical section
<a href="#">879</a>	Create	Create a new critical section.
<a href="#">880</a>	Destroy	Destroy the criticalsection instance
<a href="#">879</a>	Enter	Alias for Acquire
<a href="#">879</a>	Leave	Alias for Release
<a href="#">878</a>	Release	Leave the critical section
<a href="#">879</a>	TryEnter	Try and obtain the critical section

### 39.7.3 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: Release ([878](#))

### 39.7.4 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 immediately.

See also: Acquire ([878](#))

### 39.7.5 TCriticalSection.Enter

Synopsis: Alias for `Acquire`

Declaration: `procedure Enter`

Visibility: `public`

Description: `Enter` just calls `Acquire` (878).

See also: `Leave` (879), `Acquire` (878)

### 39.7.6 TCriticalSection.TryEnter

Synopsis: Try and obtain the critical section

Declaration: `function TryEnter : Boolean`

Visibility: `public`

Description: `TryEnter` tries to enter the critical section: it returns at once and does not wait if the critical section is owned by another thread; if the current thread owns the critical section or the critical section was obtained successfully, `true` is returned. If the critical section is currently owned by another thread, `False` is returned.

Errors: None.

See also: `TCriticalSection.Enter` (879)

### 39.7.7 TCriticalSection.Leave

Synopsis: Alias for `Release`

Declaration: `procedure Leave`

Visibility: `public`

Description: `Leave` just calls `Release` (878)

See also: `Release` (878), `Enter` (879)

### 39.7.8 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: `Destroy` (880)



### 39.7.9 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: [Create \(879\)](#), [Acquire \(878\)](#)

## 39.8 TEventObject

### 39.8.1 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.

See also: `TCriticalSection` ([877](#))

### 39.8.2 Method overview

Page	Method	Description
<a href="#">880</a>	<code>Create</code>	Create a new event object
<a href="#">881</a>	<code>destroy</code>	Clean up the event and release from memory
<a href="#">881</a>	<code>ResetEvent</code>	Reset the event
<a href="#">881</a>	<code>SetEvent</code>	Set the event
<a href="#">881</a>	<code>WaitFor</code>	Wait for the event to be set.

### 39.8.3 Property overview

Page	Properties	Access	Description
<a href="#">882</a>	<code>ManualReset</code>	<code>r</code>	Should the event be reset manually

### 39.8.4 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 with security attributes `EventAttributes` (this parameters is used on Windows only).

The `AManualReset` indicates whether the event must be reset manually (if it is `False`, the event is reset immediately after the first thread waiting for it is notified). `InitialState` determines whether the event is initially set or not.

See also: `ManualReset` ([882](#)), `ResetEvent` ([881](#))

### 39.8.5 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: `Create` ([880](#))

### 39.8.6 TEventObject.ResetEvent

**Synopsis:** Reset the event

**Declaration:** `procedure ResetEvent`

**Visibility:** `public`

**Description:** `ResetEvent` turns off the event. Any `WaitFor` ([881](#)) operation will suspend the calling thread.

See also: `SetEvent` ([881](#)), `WaitFor` ([881](#))

### 39.8.7 TEventObject.SetEvent

**Synopsis:** Set the event

**Declaration:** `procedure SetEvent`

**Visibility:** `public`

**Description:** `SetEvent` sets the event. If the `ManualReset` ([882](#)) is `True` any thread that was waiting for the event to be set (using `WaitFor` ([881](#))) 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 immediately reset after that.

See also: `WaitFor` ([881](#)), `ManualReset` ([882](#))

### 39.8.8 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` (881)) 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` (881)

### 39.8.9 TEventObject.ManualReset

**Synopsis:** Should the event be reset manually

**Declaration:** `Property ManualReset : Boolean`

**Visibility:** `public`

**Access:** `Read`

**Description:** `ManualReset` indicates whether the event must be reset manually: if it is `False`, the event is reset immediately after the first thread waiting for it is notified. if it is `True`, then the event is never reset automatically, and `ResetEvent` (881) must be called manually after a thread was notified.

See also: `ResetEvent` (881)

## 39.9 THandleObject

### 39.9.1 Description

`THandleObject` is a parent class for synchronization classes that need to store an operating system handle. It introduces a property `Handle` (883) which can be used to store the operating system handle. The handle is in no way manipulated by `THandleObject`, only storage is provided.

See also: `Handle` (883)

### 39.9.2 Method overview

Page	Method	Description
882	<code>destroy</code>	Free the instance

### 39.9.3 Property overview

Page	Properties	Access	Description
883	<code>Handle</code>	<code>r</code>	Handle for this object
883	<code>LastError</code>	<code>r</code>	Last operating system error

### 39.9.4 THandleObject.destroy

**Synopsis:** Free the instance

**Declaration:** `destructor destroy; Override`

Visibility: public

Description: `Destroy` does nothing in the Free Pascal implementation of `THandleObject`.

### 39.9.5 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.

### 39.9.6 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` (883).

See also: `Handle` (883)

## 39.10 TSimpleEvent

### 39.10.1 Description

`TSimpleEvent` is a simple descendent of the `TEventObject` (880) class. It creates an event with no name, which must be reset manually, and which is initially not set.

See also: `TEventObject` (880), `TSimpleEvent.Create` (883)

### 39.10.2 Method overview

Page	Method	Description
<a href="#">883</a>	<code>Create</code>	Creates a new <code>TSimpleEvent</code> instance

### 39.10.3 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` (880) with `Nil` for the security attributes, an empty name, `AManualReset` set to `True`, and `InitialState` to `False`.

See also: `TEventObject.Create` (880)

## 39.11 TSynchroObject

### 39.11.1 Description

`TSynchroObject` is an abstract synchronization resource object. It implements 2 virtual methods `Acquire` (884) which can be used to acquire the resource, and `Release` (884) to release the resource.

See also: `Acquire` (884), `Release` (884)

### 39.11.2 Method overview

Page	Method	Description
<a href="#">884</a>	<code>Acquire</code>	Acquire synchronization resource
<a href="#">884</a>	<code>Release</code>	Release previously acquired synchronization resource

### 39.11.3 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: `Release` (884)

### 39.11.4 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` (884) call.

See also: `Acquire` (884)

## Chapter 40

# Reference for unit 'URIParser'

### 40.1 Overview

The `URIParser` unit contains a basic type (`TURI` ([887](#))) and some routines for the parsing (`ParseURI` ([886](#))) and construction (`EncodeURI` ([885](#))) of Uniform Resource Indicators, commonly referred to as URL: Uniform Resource Location. It is used in various other units, and in itself contains no classes. It supports all protocols, username/password/port specification, query parameters and bookmarks etc..

### 40.2 Constants, types and variables

#### 40.2.1 Types

### 40.3 Procedures and functions

#### 40.3.1 EncodeURI

Synopsis: Form a string representation of the URI

Declaration: `function EncodeURI(const URI: TURI) : string`

Visibility: default

Description: `EncodeURI` will return a valid text representation of the URI in the URI record.

See also: `ParseURI` ([886](#))

#### 40.3.2 FilenameToURI

Synopsis: Construct a URI from a filename

Declaration: `function FilenameToURI(const Filename: string; Encode: Boolean) : string`

Visibility: default

Description: `FilenameToURI` takes `Filename` and constructs a `file: protocol` URI from it.

Errors: None.

See also: `URIToFilename` ([887](#))

### 40.3.3 IsAbsoluteURI

Synopsis: Check whether a URI is absolute.

Declaration: `function IsAbsoluteURI(const UriReference: string) : Boolean`

Visibility: default

Description: `IsAbsoluteURI` returns `True` if the URI in `UriReference` is absolute, i.e. contains a protocol part.

Errors: None.

See also: `FilenameToURI` (885), `URIToFileName` (887)

### 40.3.4 ParseURI

Synopsis: Parse a URI and split it into its constituent parts

Declaration: `function ParseURI(const URI: string; Decode: Boolean) : TURI; Overload`  
`function ParseURI(const URI: string; const DefaultProtocol: string;`  
`DefaultPort: Word; Decode: Boolean) : TURI; Overload`

Visibility: default

Description: `ParseURI` decodes URI and returns the various parts of the URI in the result record.

The function accepts the most general URI scheme:

`proto://user:pwd@host:port/path/document?params#bookmark`

Missing (optional) parts in the URI will be left blank in the result record. If a default protocol and port are specified, they will be used in the record if the corresponding part is not present in the URI.

See also: `EncodeURI` (885)

### 40.3.5 ResolveRelativeURI

Synopsis: Return a relative link

Declaration: `function ResolveRelativeURI(const BaseUri: UnicodeString;`  
`const RelUri: UnicodeString;`  
`out ResultUri: UnicodeString) : Boolean`  
`; Overload`  
`function ResolveRelativeURI(const BaseUri: AnsiString;`  
`const RelUri: AnsiString;`  
`out ResultUri: AnsiString) : Boolean`  
`; Overload`

Visibility: default

Description: `ResolveRelativeURI` returns in `ResultUri` an absolute link constructed from a base URI `BaseURI` and a relative link `RelURI`. One of the two URI names must have a protocol specified. If the `RelURI` argument contains a protocol, it is considered a complete (absolute) URI and is returned as the result.

The function returns `True` if a link was successfully returned.

Errors: If no protocols are specified, the function returns `False`

### 40.3.6 URIToFilename

Synopsis: Convert a URI to a filename

Declaration: `function URIToFilename(const URI: string; out Filename: string)  
: Boolean`

Visibility: default

Description: `URIToFilename` returns a filename (using the correct Path Delimiter character) from URI. The URI must be of protocol `File` or have no protocol.

Errors: If the URI contains an unsupported protocol, `False` is returned.

See also: `ResolveRelativeURI` (886), `FilenameToURI` (885)

## 40.4 TURI

```
TURI = record
  Protocol : string;
  Username : string;
  Password
    : string;
  Host : string;
  Port : Word;
  Path : string;
  Document
    : string;
  Params : string;
  Bookmark : string;
  HasAuthority
    : Boolean;
end
```

`TURI` is the basic record that can be filled by the `ParseURI` (886) call. It contains the contents of a URI, parsed out in it's various pieces.



# Chapter 41

## Reference for unit 'Zipper'

### 41.1 Used units

Table 41.1: Used units by unit 'Zipper'

Name	Page
Classes	??
sysutils	??
ZStream	<a href="#">917</a>

### 41.2 Overview

zipper implements zip compression/decompression compatible with the popular .ZIP format. The zip file format is documented at <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>. The Pascal conversion of the standard zlib library was implemented by Jacques Nomssi Nzali. It is used in the FCL to implement the (??) class.

### 41.3 Constants, types and variables

#### 41.3.1 Constants

CENTRAL\_FILE\_HEADER\_SIGNATURE = \$02014B50

Denotes beginning of a file entry inside the zip directory. A file header follows this marker.

```
Crc_32_Tab : Array[0..255] of LongWord = ($00000000, $77073096, $ee0e612c
, $990951ba, $076dc419, $706af48f, $e963a535, $9e6495a3, $0edb8832
, $79dcb8a4, $e0d5e91e, $97d2d988, $09b64c2b, $7eb17cbd, $e7b82d07
, $90bf1d91, $1db71064, $6ab020f2, $f3b97148, $84be41de, $1adad47d
, $6ddde4eb, $f4d4b551, $83d385c7, $136c9856, $646ba8c0, $fd62f97a
, $8a65c9ec, $14015c4f, $63066cd9, $fa0f3d63, $8d080df5, $3b6e20c8
, $4c69105e, $d56041e4, $a2677172, $3c03e4d1, $4b04d447, $d20d85fd
, $a50ab56b, $35b5a8fa, $42b2986c, $dbbbc9d6, $acbcf940, $32d86ce3
, $45df5c75, $dcd60dcf, $abd13d59, $26d930ac, $51de003a, $c8d75180
```

```

, $bfd06116, $21b4f4b5, $56b3c423, $cfba9599, $b8bda50f, $2802b89e
, $5f058808, $c60cd9b2, $b10be924, $2f6f7c87, $58684c11, $c1611dab
, $b6662d3d, $76dc4190, $01db7106, $98d220bc, $efd5102a, $71b18589
, $06b6b51f, $9fbfe4a5, $e8b8d433, $7807c9a2, $0f00f934, $9609a88e
, $e10e9818, $7f6a0dbb, $086d3d2d, $91646c97, $e6635c01, $6b6b51f4
, $1c6c6162, $856530d8, $f262004e, $6c0695ed, $1b01a57b, $8208f4c1
, $f50fc457, $65b0d9c6, $12b7e950, $8bbeb8ea, $fcb9887c, $62dd1ddf
, $15da2d49, $8cd37cf3, $fbd44c65, $4db26158, $3ab551ce, $a3bc0074
, $d4bb30e2, $4adfa541, $3dd895d7, $a4d1c46d, $d3d6f4fb, $4369e96a
, $346ed9fc, $ad678846, $da60b8d0, $44042d73, $33031de5, $aa0a4c5f
, $dd0d7cc9, $5005713c, $270241aa, $be0b1010, $c90c2086, $5768b525
, $206f85b3, $b966d409, $ce61e49f, $5edef90e, $29d9c998, $b0d09822
, $c7d7a8b4, $59b33d17, $2eb40d81, $b7bd5c3b, $c0ba6cad, $edb88320
, $9abfb3b6, $03b6e20c, $74b1d29a, $ead54739, $9dd277af, $04db2615
, $73dc1683, $e3630b12, $94643b84, $0d6d6a3e, $7a6a5aa8, $e40ecf0b
, $9309ff9d, $0a00ae27, $7d079eb1, $f00f9344, $8708a3d2, $1e01f268
, $6906c2fe, $f762575d, $806567cb, $196c3671, $6e6b06e7, $fed41b76
, $89d32be0, $10da7a5a, $67dd4acc, $f9b9df6f, $8ebeeff9, $17b7be43
, $60b08ed5, $d6d6a3e8, $ald1937e, $38d8c2c4, $4fdff252, $d1bb67f1
, $a6bc5767, $3fb506dd, $48b2364b, $d80d2bda, $af0a1b4c, $36034af6
, $41047a60, $df60efc3, $a867df55, $316e8eef, $4669be79, $cb61b38c
, $bc66831a, $256fd2a0, $5268e236, $cc0c7795, $bb0b4703, $220216b9
, $5505262f, $c5ba3bbe, $b2bd0b28, $2bb45a92, $5cb36a04, $c2d7ffa7
, $b5d0cf31, $2cd99e8b, $5bdeae1d, $9b64c2b0, $ec63f226, $756aa39c
, $026d930a, $9c0906a9, $eb0e363f, $72076785, $05005713, $95bf4a82
, $e2b87a14, $7bb12bae, $0cb61b38, $92d28e9b, $e5d5be0d, $7cdcef7b
, $0bdbdf21, $86d3d2d4, $f1d4e242, $68ddb3f8, $1fda836e, $81be16cd
, $f6b9265b, $6fb077e1, $18b74777, $88085ae6, $ff0f6a70, $66063bca
, $11010b5c, $8f659eff, $f862ae69, $616bffd3, $166ccf45, $a00ae278
, $d70dd2ee, $4e048354, $3903b3c2, $a7672661, $d06016f7, $4969474d
, $3e6e77db, $aed16a4a, $d9d65adc, $40df0b66, $37d83bf0, $a9bcae53
, $debb9ec5, $47b2cf7f, $30b5ffe9, $bdbdf21c, $cabac28a, $53b39330
, $24b4a3a6, $bad03605, $cdd70693, $54de5729, $23d967bf, $b3667a2e
, $c4614ab8, $5d681b02, $2a6f2b94, $b40bbe37, $c30c8ea1, $5a05df1b
, $2d02ef8d)

```

Table used in determining CRC-32 values. There are various CRC-32 algorithms in use; please refer to the ZIP file format specifications for details.

EFS\_LANGUAGE\_ENCODING\_FLAG = \$800

END\_OF\_CENTRAL\_DIR\_SIGNATURE = \$06054B50

Marker specifying end of directory within zip file

FIRSTENTRY = 257

INFOZIP\_UNICODE\_PATH\_ID = \$7075

LOCAL\_FILE\_HEADER\_SIGNATURE = \$04034B50

Denotes beginning of a file header within the zip file. A file header follows this marker, followed by the file data proper.

OS\_FAT = 0

OS\_NTFS = 10

OS\_OS2 = 6

OS\_OSX = 19

OS\_UNIX = 3

OS\_VFAT = 14

TABLESIZE = 8191

UNIX\_BLK = \$6000

UNIX\_CHAR = \$2000

UNIX\_DEFAULT = UNIX\_RUSR or UNIX\_WUSR or UNIX\_XUSR or UNIX\_RGRP or  
UNIX\_ROTH

UNIX\_DIR = \$4000

UNIX\_FIFO = \$1000

UNIX\_FILE = \$8000

UNIX\_LINK = \$A000

UNIX\_MASK = \$F000

UNIX\_RGRP = \$0020

UNIX\_ROTH = \$0004

UNIX\_RUSR = \$0100

UNIX SOCK = \$C000

UNIX WGRP = \$0010

UNIX WOTH = \$0002

UNIX WUSR = \$0080

UNIX XGRP = \$0008

UNIX XOTH = \$0001

UNIX XUSR = \$0040

ZIP64\_END\_OF\_CENTRAL\_DIR\_LOCATOR\_SIGNATURE = \$07064B50

ZIP64\_END\_OF\_CENTRAL\_DIR\_SIGNATURE = \$06064B50

ZIP64\_HEADER\_ID = \$0001

### 41.3.2 Types

BufPtr = PByte

Array = Array[0..TABLESIZE] of CodeRec

Array = Array[FIRSTENTRY..TABLESIZE] of Word

FreeListPtr = ^FreeListArray

TablePtr = ^CodeArray

TCustomInputStreamEvent = procedure(Sender: TObject;  
var AStream: TStream) of object

TOnCustomStreamEvent = procedure(Sender: TObject; var AStream: TStream  
;  
Aitem: TFullZipFileEntry) of  
object

```
TOnEndOfFileEvent = procedure(Sender: TObject; const Ratio: Double
)
of object
```

Event procedure for an end of file (de)compression event

```
TOnStartFileEvent = procedure(Sender: TObject; const AFileName: string
)
of object
```

Event procedure for a start of file (de)compression event

```
TProgressEvent = procedure(Sender: TObject; const Pct: Double) of
object
```

Event procedure for capturing compression/decompression progress

```
TProgressEventEx = procedure(Sender: TObject; const ATotPos: Int64
;
const ATotSize: Int64) of object
```

## 41.4 Central\_File\_Header\_Type

```
Central_File_Header_Type = packed record
Signature : LongInt;
MadeBy_Version : Word;
Extract_Version_Reqd : Word;
Bit_Flag
: Word;
Compress_Method : Word;
Last_Mod_Time : Word;
Last_Mod_Date
: Word;
Crc32 : LongWord;
Compressed_Size : LongWord;
Uncompressed_Size
: LongWord;
Filename_Length : Word;
Extra_Field_Length : Word
;
File_Comment_Length : Word;
Starting_Disk_Num : Word;
Internal_Attributes
: Word;
External_Attributes : LongWord;
Local_Header_Offset
: LongWord;
end
```

This record contains the structure for a file header within the central directory.

## 41.5 CodeRec

```
CodeRec = packed record
  Child : SmallInt;
  Sibling : SmallInt;
  Suffix : Byte;
end
```

## 41.6 End\_of\_Central\_Dir\_Type

```
End_of_Central_Dir_Type = packed record
  Signature : LongInt;
  Disk_Number
    : Word;
  Central_Dir_Start_Disk : Word;
  Entries_This_Disk : Word
;
  Total_Entries : Word;
  Central_Dir_Size : LongWord;
  Start_Disk_Offset
    : LongWord;
  ZipFile_Comment_Length : Word;
end
```

The end of central directory is placed at the end of the zip file. Note that the end of central directory record is distinct from the Zip64 end of central directory record and zip64 end of central directory locator, which precede the end of central directory, if implemented.

## 41.7 Extensible\_Data\_Field\_Header\_Type

```
Extensible_Data_Field_Header_Type = packed record
  Header_ID : Word
;
  Data_Size : Word;
end
```

## 41.8 Local\_File\_Header\_Type

```
Local_File_Header_Type = packed record
  Signature : LongInt;
  Extract_Version_Reqd
    : Word;
  Bit_Flag : Word;
  Compress_Method : Word;
  Last_Mod_Time
    : Word;
  Last_Mod_Date : Word;
  Crc32 : LongWord;
```

```

    Compressed_Size
    : LongWord;
    Uncompressed_Size : LongWord;
    Filename_Length : Word
    ;
    Extra_Field_Length : Word;
end

```

Record structure containing local file header

## 41.9 Zip64\_End\_of\_Central\_Dir\_Locator\_type

```

Zip64_End_of_Central_Dir_Locator_type = packed record
    Signature
    : LongInt;
    Zip64_EOCD_Start_Disk : LongWord;
    Central_Dir_Zip64_EOCD_Offset
    : QWord;
    Total_Disks : LongWord;
end

```

## 41.10 Zip64\_End\_of\_Central\_Dir\_type

```

Zip64_End_of_Central_Dir_type = packed record
    Signature : LongInt
    ;
    Record_Size : QWord;
    Version_Made_By : Word;
    Extract_Version_Reqd
    : Word;
    Disk_Number : LongWord;
    Central_Dir_Start_Disk : LongWord
    ;
    Entries_This_Disk : QWord;
    Total_Entries : QWord;
    Central_Dir_Size
    : QWord;
    Start_Disk_Offset : QWord;
end

```

## 41.11 Zip64\_Extended\_Info\_Field\_Type

```

Zip64_Extended_Info_Field_Type = packed record
    Original_Size : QWord
    ;
    Compressed_Size : QWord;
    Relative_Hdr_Offset : QWord;
    Disk_Start_Number

```

```

    : LongWord;
end

```

## 41.12 EZipError

### 41.12.1 Description

Exception specific to the zipper unit

## 41.13 TCompressor

### 41.13.1 Description

This object compresses a stream into a compressed zip stream.

### 41.13.2 Method overview

Page	Method	Description
<a href="#">895</a>	Compress	Compresses input stream to output stream
<a href="#">895</a>	Create	Creates a (??) object
<a href="#">896</a>	Terminate	
<a href="#">896</a>	ZipBitFlag	
<a href="#">896</a>	ZipID	Identifier for type of compression
<a href="#">896</a>	ZipVersionReqd	

### 41.13.3 Property overview

Page	Properties	Access	Description
<a href="#">896</a>	BufferSize	r	Size of buffer used for compression
<a href="#">897</a>	Crc32Val	rw	Running CRC32 value
<a href="#">896</a>	OnPercent	rw	Reference to OnPercent event handler
<a href="#">896</a>	OnProgress	rw	Reference to OnProgress event handler
<a href="#">897</a>	Terminated	r	

### 41.13.4 TCompressor.Create

Synopsis: Creates a (??) object

Declaration: constructor Create(AInFile: TStream; AOutFile: TStream;  
ABufSize: LongWord); Virtual

Visibility: public

### 41.13.5 TCompressor.Compress

Synopsis: Compresses input stream to output stream

Declaration: procedure Compress; Virtual; Abstract

Visibility: public



### **41.13.6 TCompressor.ZipID**

Synopsis: Identifier for type of compression

Declaration: `class function ZipID : Word; Virtual; Abstract`

Visibility: public

### **41.13.7 TCompressor.ZipVersionReqd**

Declaration: `class function ZipVersionReqd : Word; Virtual; Abstract`

Visibility: public

### **41.13.8 TCompressor.ZipBitFlag**

Declaration: `function ZipBitFlag : Word; Virtual; Abstract`

Visibility: public

### **41.13.9 TCompressor.Terminate**

Declaration: `procedure Terminate`

Visibility: public

### **41.13.10 TCompressor.BufferSize**

Synopsis: Size of buffer used for compression

Declaration: `Property BufferSize : LongWord`

Visibility: public

Access: Read

### **41.13.11 TCompressor.OnPercent**

Synopsis: Reference to OnPercent event handler

Declaration: `Property OnPercent : Integer`

Visibility: public

Access: Read,Write

### **41.13.12 TCompressor.OnProgress**

Synopsis: Reference to OnProgress event handler

Declaration: `Property OnProgress : TProgressEvent`

Visibility: public

Access: Read,Write

**41.13.13 TCompressor.Crc32Val**

Synopsis: Running CRC32 value

Declaration: `Property Crc32Val : LongWord`

Visibility: public

Access: Read,Write

Description: Running CRC32 value used when writing zip header

**41.13.14 TCompressor.Terminated**

Declaration: `Property Terminated : Boolean`

Visibility: public

Access: Read

**41.14 TDeCompressor****41.14.1 Description**

This object decompresses a compressed zip stream.

**41.14.2 Method overview**

Page	Method	Description
<a href="#">897</a>	Create	Creates decompressor object
<a href="#">898</a>	DeCompress	Decompress zip stream
<a href="#">898</a>	Terminate	
<a href="#">898</a>	ZipID	Identifier for type of compression

**41.14.3 Property overview**

Page	Properties	Access	Description
<a href="#">898</a>	BufferSize	r	Size of buffer used in decompression
<a href="#">899</a>	Crc32Val	rw	Running CRC32 value used for verifying zip file integrity
<a href="#">898</a>	OnPercent	rw	Percentage of decompression completion
<a href="#">898</a>	OnProgress	rw	Event handler for OnProgress procedure
<a href="#">898</a>	OnProgressEx	rw	
<a href="#">899</a>	Terminated	r	

**41.14.4 TDeCompressor.Create**

Synopsis: Creates decompressor object

Declaration: `constructor Create(AInFile: TStream; AOutFile: TStream;  
ABufSize: LongWord); Virtual`

Visibility: public

#### **41.14.5 TDeCompressor.DeCompress**

Synopsis: Decompress zip stream

Declaration: `procedure DeCompress; Virtual; Abstract`

Visibility: `public`

#### **41.14.6 TDeCompressor.Terminate**

Declaration: `procedure Terminate`

Visibility: `public`

#### **41.14.7 TDeCompressor.ZipID**

Synopsis: Identifier for type of compression

Declaration: `class function ZipID : Word; Virtual; Abstract`

Visibility: `public`

#### **41.14.8 TDeCompressor.BufferSize**

Synopsis: Size of buffer used in decompression

Declaration: `Property BufferSize : LongWord`

Visibility: `public`

Access: `Read`

#### **41.14.9 TDeCompressor.OnPercent**

Synopsis: Percentage of decompression completion

Declaration: `Property OnPercent : Integer`

Visibility: `public`

Access: `Read,Write`

#### **41.14.10 TDeCompressor.OnProgress**

Synopsis: Event handler for OnProgress procedure

Declaration: `Property OnProgress : TProgressEvent`

Visibility: `public`

Access: `Read,Write`

#### **41.14.11 TDeCompressor.OnProgressEx**

Declaration: `Property OnProgressEx : TProgressEventEx`

Visibility: `public`

Access: `Read,Write`

**41.14.12 TDeCompressor.Crc32Val**

Synopsis: Running CRC32 value used for verifying zip file integrity

Declaration: `Property Crc32Val : LongWord`

Visibility: `public`

Access: `Read, Write`

**41.14.13 TDeCompressor.Terminated**

Declaration: `Property Terminated : Boolean`

Visibility: `public`

Access: `Read`

**41.15 TDeflater****41.15.1 Description**

Child of `TCompressor` ([895](#)) that implements the Deflate compression method

**41.15.2 Method overview**

Page	Method	Description
<a href="#">899</a>	<code>Compress</code>	
<a href="#">899</a>	<code>Create</code>	
<a href="#">900</a>	<code>ZipBitFlag</code>	
<a href="#">900</a>	<code>ZipID</code>	
<a href="#">900</a>	<code>ZipVersionReqd</code>	

**41.15.3 Property overview**

Page	Properties	Access	Description
<a href="#">900</a>	<code>CompressionLevel</code>	<code>rw</code>	

**41.15.4 TDeflater.Create**

Declaration: `constructor Create(AInFile: TStream; AOutFile: TStream;  
ABufSize: LongWord); Override`

Visibility: `public`

**41.15.5 TDeflater.Compress**

Declaration: `procedure Compress; Override`

Visibility: `public`

### 41.15.6 TDeflater.ZipID

Declaration: class function ZipID : Word; Override

Visibility: public

### 41.15.7 TDeflater.ZipVersionReqd

Declaration: class function ZipVersionReqd : Word; Override

Visibility: public

### 41.15.8 TDeflater.ZipBitFlag

Declaration: function ZipBitFlag : Word; Override

Visibility: public

### 41.15.9 TDeflater.CompressionLevel

Declaration: Property CompressionLevel : Tcompressionlevel

Visibility: public

Access: Read,Write

## 41.16 TFullZipFileEntries

### 41.16.1 Property overview

Page	Properties	Access	Description
<a href="#">900</a>	FullEntries	rw	

### 41.16.2 TFullZipFileEntries.FullEntries

Declaration: Property FullEntries[AIndex: Integer]: TFullZipFileEntry; default

Visibility: public

Access: Read,Write

## 41.17 TFullZipFileEntry

### 41.17.1 Property overview

Page	Properties	Access	Description
<a href="#">901</a>	BitFlags	r	
<a href="#">901</a>	CompressedSize	r	
<a href="#">901</a>	CompressMethod	r	
<a href="#">901</a>	CRC32	rw	

### 41.17.2 TFullZipFileEntry.BitFlags

Declaration: Property BitFlags : Word

Visibility: public

Access: Read

### 41.17.3 TFullZipFileEntry.CompressMethod

Declaration: Property CompressMethod : Word

Visibility: public

Access: Read

### 41.17.4 TFullZipFileEntry.CompressedSize

Declaration: Property CompressedSize : QWord

Visibility: public

Access: Read

### 41.17.5 TFullZipFileEntry.CRC32

Declaration: Property CRC32 : LongWord

Visibility: public

Access: Read, Write

## 41.18 TInflater

### 41.18.1 Description

Child of TDeCompressor ([897](#)) that implements the Inflate decompression method

### 41.18.2 Method overview

Page	Method	Description
<a href="#">901</a>	Create	
<a href="#">902</a>	DeCompress	
<a href="#">902</a>	ZipID	

### 41.18.3 TInflater.Create

Declaration: constructor Create(AInFile: TStream; AOutFile: TStream;  
ABufSize: LongWord); Override

Visibility: public

### 41.18.4 TInflater.DeCompress

Declaration: procedure DeCompress; Override

Visibility: public

### 41.18.5 TInflater.ZipID

Declaration: class function ZipID : Word; Override

Visibility: public

## 41.19 TShrinker

### 41.19.1 Description

Child of TCompressor ([895](#)) that implements the Shrink compression method

### 41.19.2 Method overview

Page	Method	Description
<a href="#">902</a>	Compress	
<a href="#">902</a>	Create	
<a href="#">902</a>	Destroy	
<a href="#">903</a>	ZipBitFlag	
<a href="#">902</a>	ZipID	
<a href="#">903</a>	ZipVersionReqd	

### 41.19.3 TShrinker.Create

Declaration: constructor Create(AInFile: TStream; AOutFile: TStream;  
ABufSize: LongWord); Override

Visibility: public

### 41.19.4 TShrinker.Destroy

Declaration: destructor Destroy; Override

Visibility: public

### 41.19.5 TShrinker.Compress

Declaration: procedure Compress; Override

Visibility: public

### 41.19.6 TShrinker.ZipID

Declaration: class function ZipID : Word; Override

Visibility: public

### 41.19.7 TShrinker.ZipVersionReqd

Declaration: class function ZipVersionReqd : Word; Override

Visibility: public

### 41.19.8 TShrinker.ZipBitFlag

Declaration: function ZipBitFlag : Word; Override

Visibility: public

## 41.20 TUnZipper

### 41.20.1 Method overview

Page	Method	Description
<a href="#">905</a>	Clear	Removes all entries and files from object
<a href="#">904</a>	Create	
<a href="#">904</a>	Destroy	
<a href="#">905</a>	Examine	Opens zip file and reads the directory entries (list of zipped files)
<a href="#">905</a>	Terminate	
<a href="#">905</a>	Unzip	
<a href="#">904</a>	UnZipAllFiles	Unzips all files in a zip file, writing them to disk
<a href="#">904</a>	UnZipFile	
<a href="#">904</a>	UnZipFiles	Unzips specified files

### 41.20.2 Property overview

Page	Properties	Access	Description
<a href="#">905</a>	BufferSize	rw	
<a href="#">907</a>	Entries	r	
<a href="#">907</a>	FileComment	r	
<a href="#">907</a>	FileName	rw	Zip file to be unzipped/processed
<a href="#">907</a>	Files	r	Files in zip file (deprecated)
<a href="#">908</a>	Flat	rw	
<a href="#">906</a>	OnCloseInputStream	rw	
<a href="#">906</a>	OnCreateStream	rw	
<a href="#">906</a>	OnDoneStream	rw	
<a href="#">907</a>	OnEndFile	rw	Callback procedure that will be called after unzipping a file
<a href="#">905</a>	OnOpenInputStream	rw	
<a href="#">906</a>	OnPercent	rw	
<a href="#">906</a>	OnProgress	rw	
<a href="#">906</a>	OnProgressEx	rw	
<a href="#">906</a>	OnStartFile	rw	Callback procedure that will be called before unzipping a file
<a href="#">907</a>	OutputPath	rw	
<a href="#">908</a>	Terminated	r	Path where archive files will be unzipped
<a href="#">908</a>	UseUTF8	rw	



### 41.20.3 TUnZipper.Create

Declaration: constructor Create

Visibility: public

### 41.20.4 TUnZipper.Destroy

Declaration: destructor Destroy; Override

Visibility: public

### 41.20.5 TUnZipper.UnZipAllFiles

Synopsis: Unzips all files in a zip file, writing them to disk

Declaration: procedure UnZipAllFiles; Virtual  
procedure UnZipAllFiles(const AZipFileName: RawByteString)

Visibility: public

Description: This procedure unzips all files in a (??) object and writes the unzipped files to disk.

The example below unzips the files into "C:\windows\temp":

```

uses
    Zipper;
var
    UnZipper: TUnZipper;
begin
    UnZipper := TUnZipper.Create;
    try
        UnZipper.FileName := ZipFilePath;
        UnZipper.OutputPath := 'C:\Windows\Temp';
        UnZipper.UnZipAllFiles;
    finally
        UnZipper.Free;
    end;
end.

```

### 41.20.6 TUnZipper.UnZipFile

Declaration: procedure UnZipFile(const aExtractFileName: RawByteString)  
procedure UnZipFile(const AZipFileName: RawByteString;  
const aExtractFileName: RawByteString)

Visibility: public

### 41.20.7 TUnZipper.UnZipFiles

Synopsis: Unzips specified files

Declaration: procedure UnZipFiles(const AZipFileName: RawByteString;  
FileList: TStrings)  
procedure UnZipFiles(const AZipFileName: RawByteString;

```
        aFileList: Array of RawByteString)
procedure UnZipFiles(aFileList: TStrings)
```

Visibility: public

#### **41.20.8 TUnZipper.Unzip**

Declaration: 

```
class procedure Unzip(const AZipFileName: RawByteString)
class procedure Unzip(const AZipFileName: RawByteString;
    aExtractFileName: RawByteString)
class procedure Unzip(const AZipFileName: RawByteString;
    aFileList: Array of RawByteString)
class procedure Unzip(const AZipFileName: RawByteString;
    aFileList: TStrings)
```

Visibility: public

#### **41.20.9 TUnZipper.Clear**

Synopsis: Removes all entries and files from object

Declaration: 

```
procedure Clear
```

Visibility: public

#### **41.20.10 TUnZipper.Examine**

Synopsis: Opens zip file and reads the directory entries (list of zipped files)

Declaration: 

```
procedure Examine
```

Visibility: public

#### **41.20.11 TUnZipper.Terminate**

Declaration: 

```
procedure Terminate
```

Visibility: public

#### **41.20.12 TUnZipper.BufferSize**

Declaration: 

```
Property BufferSize : LongWord
```

Visibility: public

Access: Read,Write

#### **41.20.13 TUnZipper.OnOpenInputStream**

Declaration: 

```
Property OnOpenInputStream : TCustomInputStreamEvent
```

Visibility: public

Access: Read,Write

#### **41.20.14 TUnZipper.OnCloseInputStream**

Declaration: Property OnCloseInputStream : TCustomInputStreamEvent

Visibility: public

Access: Read,Write

#### **41.20.15 TUnZipper.OnCreateStream**

Declaration: Property OnCreateStream : TOnCustomStreamEvent

Visibility: public

Access: Read,Write

#### **41.20.16 TUnZipper.OnDoneStream**

Declaration: Property OnDoneStream : TOnCustomStreamEvent

Visibility: public

Access: Read,Write

#### **41.20.17 TUnZipper.OnPercent**

Declaration: Property OnPercent : Integer

Visibility: public

Access: Read,Write

#### **41.20.18 TUnZipper.OnProgress**

Declaration: Property OnProgress : TProgressEvent

Visibility: public

Access: Read,Write

#### **41.20.19 TUnZipper.OnProgressEx**

Declaration: Property OnProgressEx : TProgressEventEx

Visibility: public

Access: Read,Write

#### **41.20.20 TUnZipper.OnStartFile**

Synopsis: Callback procedure that will be called before unzipping a file

Declaration: Property OnStartFile : TOnStartFileEvent

Visibility: public

Access: Read,Write

#### 41.20.21 TUnZipper.OnEndFile

Synopsis: Callback procedure that will be called after unzipping a file

Declaration: `Property OnEndFile : TOnEndOfFileEvent`

Visibility: `public`

Access: `Read,Write`

#### 41.20.22 TUnZipper.FileName

Synopsis: Zip file to be unzipped/processed

Declaration: `Property FileName : RawByteString`

Visibility: `public`

Access: `Read,Write`

#### 41.20.23 TUnZipper.OutputPath

Synopsis: Path where archive files will be unzipped

Declaration: `Property OutputPath : RawByteString`

Visibility: `public`

Access: `Read,Write`

#### 41.20.24 TUnZipper.FileComment

Declaration: `Property FileComment : string`

Visibility: `public`

Access: `Read`

#### 41.20.25 TUnZipper.Files

Synopsis: Files in zip file (deprecated)

Declaration: `Property Files : TStrings`

Visibility: `public`

Access: `Read`

Description: List of files that should be compressed in the zip file. Note: deprecated. Use `Entries.AddFileEntry(FileName)` or `Entries.AddFileEntries(List)` instead.

#### 41.20.26 TUnZipper.Entries

Declaration: `Property Entries : TFullZipFileEntries`

Visibility: `public`

Access: `Read`

**41.20.27 TUnZipper.UseUTF8**

Declaration: `Property UseUTF8 : Boolean`

Visibility: `public`

Access: `Read,Write`

**41.20.28 TUnZipper.Flat**

Declaration: `Property Flat : Boolean`

Visibility: `public`

Access: `Read,Write`

**41.20.29 TUnZipper.Terminated**

Declaration: `Property Terminated : Boolean`

Visibility: `public`

Access: `Read`

**41.21 TZipFileEntries****41.21.1 Description**

Files in the zip archive

**41.21.2 Method overview**

Page	Method	Description
<a href="#">909</a>	<code>AddFileEntries</code>	
<a href="#">908</a>	<code>AddFileEntry</code>	Adds file to zip directory

**41.21.3 Property overview**

Page	Properties	Access	Description
<a href="#">909</a>	<code>Entries</code>	<code>rw</code>	Entries (files) in the zip archive

**41.21.4 TZipFileEntries.AddFileEntry**

Synopsis: Adds file to zip directory

Declaration: `function AddFileEntry(const ADiskFileName: string) : TZipFileEntry`  
`function AddFileEntry(const ADiskFileName: string;`  
`const AArchiveFileName: string) : TZipFileEntry`  
`function AddFileEntry(const AStream: TStream;`  
`const AArchiveFileName: string) : TZipFileEntry`

Visibility: `public`

Description: Adds a file to the list of files that will be written out in the zip file.

### 41.21.5 TZipFileEntries.AddFileEntries

Declaration: `procedure AddFileEntries(const List: TStrings)`

Visibility: `public`

### 41.21.6 TZipFileEntries.Entries

Synopsis: Entries (files) in the zip archive

Declaration: `Property Entries[AIndex: Integer]: TZipFileEntry; default`

Visibility: `public`

Access: `Read,Write`

## 41.22 TZipFileEntry

### 41.22.1 Method overview

Page	Method	Description
<a href="#">910</a>	Assign	
<a href="#">909</a>	Create	
<a href="#">909</a>	IsDirectory	
<a href="#">910</a>	IsLink	

### 41.22.2 Property overview

Page	Properties	Access	Description
<a href="#">910</a>	ArchiveFileName	rw	
<a href="#">911</a>	Attributes	rw	
<a href="#">911</a>	CompressionLevel	rw	
<a href="#">911</a>	DateTime	rw	
<a href="#">910</a>	DiskFileName	rw	
<a href="#">911</a>	OS	rw	Indication of operating system/file system
<a href="#">911</a>	Size	rw	
<a href="#">910</a>	Stream	rw	
<a href="#">910</a>	UTF8ArchiveFileName	rw	
<a href="#">910</a>	UTF8DiskFileName	rw	

### 41.22.3 TZipFileEntry.Create

Declaration: `constructor Create(ACollection: TCollection); Override`

Visibility: `public`

### 41.22.4 TZipFileEntry.IsDirectory

Declaration: `function IsDirectory : Boolean`

Visibility: `public`

#### **41.22.5 TZipFileEntry.IsLink**

Declaration: `function IsLink : Boolean`

Visibility: `public`

#### **41.22.6 TZipFileEntry.Assign**

Declaration: `procedure Assign(Source: TPersistent); Override`

Visibility: `public`

#### **41.22.7 TZipFileEntry.Stream**

Declaration: `Property Stream : TStream`

Visibility: `public`

Access: `Read,Write`

#### **41.22.8 TZipFileEntry.ArchiveFileName**

Declaration: `Property ArchiveFileName : string`

Visibility: `published`

Access: `Read,Write`

#### **41.22.9 TZipFileEntry.UTF8ArchiveFileName**

Declaration: `Property UTF8ArchiveFileName : UTF8String`

Visibility: `published`

Access: `Read,Write`

#### **41.22.10 TZipFileEntry.DiskFileName**

Declaration: `Property DiskFileName : string`

Visibility: `published`

Access: `Read,Write`

#### **41.22.11 TZipFileEntry.UTF8DiskFileName**

Declaration: `Property UTF8DiskFileName : UTF8String`

Visibility: `published`

Access: `Read,Write`

#### **41.22.12 TZipFileEntry.Size**

Declaration: Property Size : Int64

Visibility: published

Access: Read,Write

#### **41.22.13 TZipFileEntry.DateTime**

Declaration: Property DateTime : TDateTime

Visibility: published

Access: Read,Write

#### **41.22.14 TZipFileEntry.OS**

Synopsis: Indication of operating system/file system

Declaration: Property OS : Byte

Visibility: published

Access: Read,Write

Description: Currently either OS\_UNIX (if UNIX is defined) or OS\_FAT.

#### **41.22.15 TZipFileEntry.Attributes**

Declaration: Property Attributes : LongWord

Visibility: published

Access: Read,Write

#### **41.22.16 TZipFileEntry.CompressionLevel**

Declaration: Property CompressionLevel : Tcompressionlevel

Visibility: published

Access: Read,Write



## 41.23 TZipper

### 41.23.1 Method overview

Page	Method	Description
<a href="#">914</a>	Clear	
<a href="#">912</a>	Create	
<a href="#">912</a>	Destroy	
<a href="#">913</a>	SaveToFile	
<a href="#">913</a>	SaveToStream	
<a href="#">914</a>	Terminate	
<a href="#">913</a>	Zip	
<a href="#">912</a>	ZipAllFiles	Zips all files in object and writes zip to disk
<a href="#">913</a>	ZipFile	
<a href="#">913</a>	ZipFiles	

### 41.23.2 Property overview

Page	Properties	Access	Description
<a href="#">914</a>	BufferSize	rw	
<a href="#">915</a>	Entries	rw	
<a href="#">915</a>	FileComment	rw	
<a href="#">915</a>	FileName	rw	
<a href="#">915</a>	Files	r	
<a href="#">915</a>	InMemSize	rw	
<a href="#">915</a>	OnEndFile	rw	
<a href="#">914</a>	OnPercent	rw	
<a href="#">914</a>	OnProgress	rw	
<a href="#">914</a>	OnStartFile	rw	
<a href="#">915</a>	Terminated	r	
<a href="#">916</a>	UseLanguageEncoding	rw	

### 41.23.3 TZipper.Create

Declaration: `constructor Create`

Visibility: `public`

### 41.23.4 TZipper.Destroy

Declaration: `destructor Destroy; Override`

Visibility: `public`

### 41.23.5 TZipper.ZipAllFiles

Synopsis: Zips all files in object and writes zip to disk

Declaration: `procedure ZipAllFiles; Virtual`

Visibility: `public`

Description: This procedure zips up all files in the TZipper ([912](#)) object and writes the resulting zip file to disk.

An example of using this procedure:

```

uses
  Zipper;
var
  Zipper: TZipper;
begin
  try
    Zipper := TZipper.Create;
    Zipper.FileName := ParamStr(1); //Use the first parameter on the command line as
    for I := 2 to ParamCount do //Use the other arguments on the command line as fil
      Zipper.Entries.AddFileEntry(ParamStr(I), ParamStr(I));
    Zipper.ZipAllFiles;
  finally
    Zipper.Free;
  end;
end.

```

#### 41.23.6 TZipper.SaveToFile

Declaration: procedure SaveToFile(const AFileName: RawByteString)

Visibility: public

#### 41.23.7 TZipper.SaveToStream

Declaration: procedure SaveToStream(AStream: TStream)

Visibility: public

#### 41.23.8 TZipper.ZipFile

Declaration: procedure ZipFile(const aFileToBeZipped: RawByteString)

```

  procedure ZipFile(const AZipFileName: RawByteString;
    const aFileToBeZipped: RawByteString)

```

Visibility: public

#### 41.23.9 TZipper.ZipFiles

Declaration: procedure ZipFiles(const AZipFileName: RawByteString;

```

  FileList: TStrings)
  procedure ZipFiles(const AZipFileName: RawByteString;
    const FileList: Array of RawByteString)
  procedure ZipFiles(const aFileList: Array of RawByteString)
  procedure ZipFiles(FileList: TStrings)
  procedure ZipFiles(const AZipFileName: RawByteString;
    Entries: TZipFileEntries)
  procedure ZipFiles(Entries: TZipFileEntries)

```

Visibility: public

#### 41.23.10 TZipper.Zip

Declaration: class procedure Zip(const AZipFileName: RawByteString;

```
const aFileToBeZipped: RawByteString)
class procedure Zip(const AZipFileName: RawByteString;
  aFileList: Array of RawByteString)
class procedure Zip(const AZipFileName: RawByteString;
  aFileList: TStrings)
```

Visibility: public

#### **41.23.11 TZipper.Clear**

Declaration: procedure Clear

Visibility: public

#### **41.23.12 TZipper.Terminate**

Declaration: procedure Terminate

Visibility: public

#### **41.23.13 TZipper.BufferSize**

Declaration: Property BufferSize : LongWord

Visibility: public

Access: Read,Write

#### **41.23.14 TZipper.OnPercent**

Declaration: Property OnPercent : Integer

Visibility: public

Access: Read,Write

#### **41.23.15 TZipper.OnProgress**

Declaration: Property OnProgress : TProgressEvent

Visibility: public

Access: Read,Write

#### **41.23.16 TZipper.OnStartFile**

Declaration: Property OnStartFile : TOnStartFileEvent

Visibility: public

Access: Read,Write

#### 41.23.17 TZipper.OnEndFile

Declaration: Property OnEndFile : TOnEndOfFileEvent

Visibility: public

Access: Read,Write

#### 41.23.18 TZipper.FileName

Declaration: Property FileName : RawByteString

Visibility: public

Access: Read,Write

#### 41.23.19 TZipper.FileComment

Declaration: Property FileComment : string

Visibility: public

Access: Read,Write

#### 41.23.20 TZipper.Files

Declaration: Property Files : TStrings; deprecated;

Visibility: public

Access: Read

#### 41.23.21 TZipper.InMemSize

Declaration: Property InMemSize : Int64

Visibility: public

Access: Read,Write

#### 41.23.22 TZipper.Entries

Declaration: Property Entries : TZipFileEntries

Visibility: public

Access: Read,Write

#### 41.23.23 TZipper.Terminated

Declaration: Property Terminated : Boolean

Visibility: public

Access: Read

#### **41.23.24 TZipper.UseLanguageEncoding**

**Declaration:** Property UseLanguageEncoding : Boolean

**Visibility:** public

**Access:** Read, Write

## Chapter 42

# Reference for unit 'ZStream'

### 42.1 Used units

Table 42.1: Used units by unit 'ZStream'

Name	Page
Classes	??
gzio	??
zbase	??

### 42.2 Overview

The `ZStream` unit implements a `TStream` (??) descendent (`TCompressionStream` (918)) 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` (921)) and decompresses it with the inflate algorithm.

### 42.3 Constants, types and variables

#### 42.3.1 Types

`Tcompressionlevel = (clnone, clfastest, cldefault, clmax)`

Table 42.2: 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.

Compression level for the deflate algorithm

`Tgzopenmode = (gzopenread, gzopenwrite)`

Table 42.3: Enumeration values for type `Tgzopenmode`

Value	Explanation
<code>gzopenread</code>	Open file for reading
<code>gzopenwrite</code>	Open file for writing

Open mode for gzip file.

## 42.4 Ecompressionerror

### 42.4.1 Description

`ECompressionError` is the exception class used by the `TCompressionStream` (918) class.

## 42.5 Edecompressionerror

### 42.5.1 Description

`EDecompressionError` is the exception class used by the `TDeCompressionStream` (921) class.

## 42.6 Egzfileerror

### 42.6.1 Description

`Egzfileerror` is the exception class used to report errors by the `Tgzfilestream` (924) class.

See also: `Tgzfilestream` (924)

## 42.7 Ezliberror

### 42.7.1 Description

Errors which occur in the `zstream` unit are signaled by raising an `EZLibError` exception descendant.

## 42.8 Tcompressionstream

### 42.8.1 Description

`TCompressionStream`

### 42.8.2 Method overview

Page	Method	Description
<a href="#">919</a>	<code>create</code>	Create a new instance of the compression stream.
<a href="#">919</a>	<code>destroy</code>	Flushes data to the output stream and destroys the compression stream.
<a href="#">920</a>	<code>flush</code>	Flush remaining data to the target stream
<a href="#">920</a>	<code>get\_compressionrate</code>	Get the current compression rate
<a href="#">919</a>	<code>write</code>	Write data to the stream

### 42.8.3 Property overview

Page	Properties	Access	Description
<a href="#">920</a>	<code>OnProgress</code>		Progress handler

### 42.8.4 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: `Destroy` ([919](#))

### 42.8.5 Tcompressionstream.destroy

**Synopsis:** Flushes 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: `Create` ([919](#))

### 42.8.6 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 compresses (deflates) them. The compressed result is written to the output stream.

**Errors:** If an error occurs, an `ECompressionError` (918) exception is raised.

**See also:** `Read` (918), `Seek` (918)

### 42.8.7 `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.

### 42.8.8 `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.

### 42.8.9 `Tcompressionstream.OnProgress`

**Synopsis:** Progress handler

**Declaration:** `Property OnProgress :`

**Visibility:** `public`

**Access:**

**Description:** `OnProgress` is called whenever output data is written to the output stream. It can be used to update a progress bar or so. The `Sender` argument to the progress handler is the compression stream instance.

## 42.9 `Tcustomzlibstream`

### 42.9.1 `Description`

`TCustomZlibStream` serves as the ancestor class for the `TCompressionStream` (918) and `TDecompressionStream` (921) classes.

It introduces support for a progress handler, and stores the input or output stream.

### 42.9.2 Method overview

Page	Method	Description
<a href="#">921</a>	create	Create a new instance of TCustomZlibStream
<a href="#">921</a>	destroy	Clear up instance

### 42.9.3 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 ([918](#)), TDecompressionStream ([921](#))

### 42.9.4 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 ([921](#))

## 42.10 Tdecompressionstream

### 42.10.1 Description

TDecompressionStream performs the inverse operation of TCompressionStream ([918](#)). A read operation reads data from an input stream and decompresses (inflates) the data as it goes along.

The decompression stream reads its compressed data from a stream with deflated data. This data can be created e.g. with a TCompressionStream ([918](#)) compression stream.

See also: TCompressionStream ([918](#))

### 42.10.2 Method overview

Page	Method	Description
<a href="#">922</a>	create	Creates a new instance of the TDecompressionStream stream
<a href="#">922</a>	destroy	Destroys the TDecompressionStream instance
<a href="#">923</a>	get\_compressionrate	Get the current compression rate
<a href="#">922</a>	read	Read data from the compressed stream
<a href="#">923</a>	Seek	Move stream position to a certain location in the stream.

### 42.10.3 Property overview

Page	Properties	Access	Description
<a href="#">923</a>	OnProgress		Progress handler

### 42.10.4 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: `Destroy` ([922](#))

### 42.10.5 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: `Create` ([922](#))

### 42.10.6 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` ([918](#)) exception is raised.

See also: `Write` ([919](#))

### 42.10.7 Tdecompressionstream.Seek

Synopsis: Move stream position to a certain location in the stream.

Declaration: `function Seek(const Offset: Int64; Origin: TSeekOrigin) : Int64; 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` (918) exception is raised if the stream does not allow the requested seek operation.

See also: `Read` (922)

### 42.10.8 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.

### 42.10.9 Tdecompressionstream.OnProgress

Synopsis: Progress handler

Declaration: `Property OnProgress :`

Visibility: public

Access:

Description: `OnProgress` is called whenever input data is read from the source stream. It can be used to update a progress bar or so. The `Sender` argument to the progress handler is the decompression stream instance.

## 42.11 TGZFileStream

### 42.11.1 Description

`TGZFileStream` can be used to read data from a gzip file, or to write data to a gzip file.

See also: `TCompressionStream` (918), `TDeCompressionStream` (921)

### 42.11.2 Method overview

Page	Method	Description
<a href="#">924</a>	<code>create</code>	Create a new instance of <code>TGZFileStream</code>
<a href="#">925</a>	<code>destroy</code>	Removes <code>TGZFileStream</code> instance
<a href="#">924</a>	<code>read</code>	Read data from the compressed file
<a href="#">925</a>	<code>seek</code>	Set the position in the compressed stream.
<a href="#">925</a>	<code>write</code>	Write data to be compressed

### 42.11.3 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` (924) 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` (925) method will be stored in the file in compressed (deflated) form.

Errors: If the file is not found, an `EZlibError` (918) exception is raised.

See also: `Destroy` (925), `TGZOpenMode` (918)

### 42.11.4 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: `Create` (924), `Write` (925), `Seek` (925)

### 42.11.5 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` (918) exception is raised.

See also: `Create` (924), `Read` (924), `Seek` (925)

### 42.11.6 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` (923) for a list of possibilities.

Errors: In case an impossible combination is asked, an `EZlibError` (918) exception is raised.

See also: `TDecompressionStream.Seek` (923)

### 42.11.7 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: `Create` (924)