

Free Component Library (FCL):
Reference guide.

Reference guide for FCL units.
Document version 2.1
August 2007

Michaël Van Canneyt

Contents

0.1	Overview	20
1	Reference for unit 'base64'	21
1.1	Used units	21
1.2	Overview	21
1.3	Constants, types and variables	21
1.3.1	Types	21
1.4	EBase64DecodingException	22
1.4.1	Description	22
1.5	TBase64DecodingStream	22
1.5.1	Description	22
1.5.2	Method overview	22
1.5.3	Property overview	22
1.5.4	TBase64DecodingStream.Create	22
1.5.5	TBase64DecodingStream.Reset	23
1.5.6	TBase64DecodingStream.Read	23
1.5.7	TBase64DecodingStream.Write	23
1.5.8	TBase64DecodingStream.Seek	24
1.5.9	TBase64DecodingStream.EOF	24
1.5.10	TBase64DecodingStream.Mode	24
1.6	TBase64EncodingStream	24
1.6.1	Description	24
1.6.2	Method overview	25
1.6.3	TBase64EncodingStream.Create	25
1.6.4	TBase64EncodingStream.Destroy	25
1.6.5	TBase64EncodingStream.Read	25
1.6.6	TBase64EncodingStream.Write	25
1.6.7	TBase64EncodingStream.Seek	26
2	Reference for unit 'bufstream'	27
2.1	Used units	27
2.2	Overview	27

2.3	Constants, types and variables	27
2.3.1	Constants	27
2.4	TBufStream	27
2.4.1	Description	27
2.4.2	Method overview	28
2.4.3	Property overview	28
2.4.4	TBufStream.Create	28
2.4.5	TBufStream.Destroy	28
2.4.6	TBufStream.Buffer	28
2.4.7	TBufStream.Capacity	29
2.4.8	TBufStream.BufferPos	29
2.4.9	TBufStream.BufferSize	29
2.5	TReadBufStream	30
2.5.1	Description	30
2.5.2	Method overview	30
2.5.3	TReadBufStream.Seek	30
2.5.4	TReadBufStream.Read	30
2.5.5	TReadBufStream.Write	30
2.6	TWriteBufStream	31
2.6.1	Description	31
2.6.2	Method overview	31
2.6.3	TWriteBufStream.Destroy	31
2.6.4	TWriteBufStream.Seek	31
2.6.5	TWriteBufStream.Read	32
2.6.6	TWriteBufStream.Write	32
3	Reference for unit 'CacheCls'	33
3.1	Used units	33
3.2	Overview	33
3.3	Constants, types and variables	33
3.3.1	Resource strings	33
3.3.2	Types	33
3.4	ECacheError	34
3.4.1	Description	34
3.5	TCache	34
3.5.1	Description	34
3.5.2	Method overview	35
3.5.3	Property overview	35
3.5.4	TCache.Create	35
3.5.5	TCache.Destroy	35

3.5.6	TCache.Add	35
3.5.7	TCache.AddNew	36
3.5.8	TCache.FindSlot	36
3.5.9	TCache.IndexOf	36
3.5.10	TCache.Remove	37
3.5.11	TCache.Data	37
3.5.12	TCache.MRUSlot	37
3.5.13	TCache.LRUSlot	38
3.5.14	TCache.SlotCount	38
3.5.15	TCache.Slots	38
3.5.16	TCache.OnIsDataEqual	38
3.5.17	TCache.OnFreeSlot	39
4	Reference for unit 'contnrs'	40
4.1	Used units	40
4.2	Overview	40
4.3	Constants, types and variables	40
4.3.1	Constants	40
4.3.2	Types	41
4.4	Procedures and functions	43
4.4.1	RSHash	43
4.5	EDuplicate	43
4.5.1	Description	43
4.6	EKeyNotFound	43
4.6.1	Description	43
4.7	TClassList	43
4.7.1	Description	43
4.7.2	Method overview	43
4.7.3	Property overview	44
4.7.4	TClassList.Add	44
4.7.5	TClassList.Extract	44
4.7.6	TClassList.Remove	44
4.7.7	TClassList.IndexOf	45
4.7.8	TClassList.First	45
4.7.9	TClassList.Last	45
4.7.10	TClassList.Insert	45
4.7.11	TClassList.Items	46
4.8	TComponentList	46
4.8.1	Description	46
4.8.2	Method overview	46

4.8.3	Property overview	46
4.8.4	TComponentList.Destroy	46
4.8.5	TComponentList.Add	47
4.8.6	TComponentList.Extract	47
4.8.7	TComponentList.Remove	47
4.8.8	TComponentList.IndexOf	47
4.8.9	TComponentList.First	48
4.8.10	TComponentList.Last	48
4.8.11	TComponentList.Insert	48
4.8.12	TComponentList.Items	49
4.9	TFPCustomHashTable	49
4.9.1	Description	49
4.9.2	Method overview	49
4.9.3	Property overview	49
4.9.4	TFPCustomHashTable.Create	50
4.9.5	TFPCustomHashTable.CreateWith	50
4.9.6	TFPCustomHashTable.Destroy	50
4.9.7	TFPCustomHashTable.ChangeTableSize	50
4.9.8	TFPCustomHashTable.Clear	51
4.9.9	TFPCustomHashTable.Delete	51
4.9.10	TFPCustomHashTable.Find	51
4.9.11	TFPCustomHashTable.IsEmpty	51
4.9.12	TFPCustomHashTable.HashFunction	52
4.9.13	TFPCustomHashTable.Count	52
4.9.14	TFPCustomHashTable.HashTableSize	52
4.9.15	TFPCustomHashTable.HashTable	52
4.9.16	TFPCustomHashTable.VoidSlots	53
4.9.17	TFPCustomHashTable.LoadFactor	53
4.9.18	TFPCustomHashTable.AVGChainLen	53
4.9.19	TFPCustomHashTable.MaxChainLength	53
4.9.20	TFPCustomHashTable.NumberOfCollisions	54
4.9.21	TFPCustomHashTable.Density	54
4.10	TFPDataHashTable	54
4.10.1	Description	54
4.10.2	Method overview	54
4.10.3	Property overview	55
4.10.4	TFPDataHashTable.Add	55
4.10.5	TFPDataHashTable.Items	55
4.11	TFPHashList	55
4.11.1	Description	55

4.11.2	Method overview	56
4.11.3	Property overview	56
4.11.4	TFPHashList.Create	56
4.11.5	TFPHashList.Destroy	56
4.11.6	TFPHashList.Add	57
4.11.7	TFPHashList.Clear	57
4.11.8	TFPHashList.NameOfIndex	57
4.11.9	TFPHashList.HashOfIndex	57
4.11.10	TFPHashList.Delete	58
4.11.11	TFPHashList.Error	58
4.11.12	TFPHashList.Expand	58
4.11.13	TFPHashList.Extract	58
4.11.14	TFPHashList.IndexOf	59
4.11.15	TFPHashList.Find	59
4.11.16	TFPHashList.FindIndexOf	59
4.11.17	TFPHashList.FindWithHash	59
4.11.18	TFPHashList.Rename	60
4.11.19	TFPHashList.Remove	60
4.11.20	TFPHashList.Pack	60
4.11.21	TFPHashList.ShowStatistics	60
4.11.22	TFPHashList.ForEachCall	61
4.11.23	TFPHashList.Capacity	61
4.11.24	TFPHashList.Count	61
4.11.25	TFPHashList.Items	61
4.11.26	TFPHashList.List	62
4.11.27	TFPHashList.Strs	62
4.12	TFPHashObject	62
4.12.1	Description	62
4.12.2	Method overview	62
4.12.3	Property overview	62
4.12.4	TFPHashObject.CreateNotOwned	63
4.12.5	TFPHashObject.Create	63
4.12.6	TFPHashObject.ChangeOwner	63
4.12.7	TFPHashObject.ChangeOwnerAndName	63
4.12.8	TFPHashObject.Rename	64
4.12.9	TFPHashObject.Name	64
4.12.10	TFPHashObject.Hash	64
4.13	TFPHashObjectList	65
4.13.1	Method overview	65
4.13.2	Property overview	65

4.13.3	TFPHashObjectList.Create	65
4.13.4	TFPHashObjectList.Destroy	65
4.13.5	TFPHashObjectList.Clear	66
4.13.6	TFPHashObjectList.Add	66
4.13.7	TFPHashObjectList.NameOfIndex	66
4.13.8	TFPHashObjectList.HashOfIndex	67
4.13.9	TFPHashObjectList.Delete	67
4.13.10	TFPHashObjectList.Expand	67
4.13.11	TFPHashObjectList.Extract	67
4.13.12	TFPHashObjectList.Remove	68
4.13.13	TFPHashObjectList.IndexOf	68
4.13.14	TFPHashObjectList.Find	68
4.13.15	TFPHashObjectList.FindIndexOf	68
4.13.16	TFPHashObjectList.FindWithHash	69
4.13.17	TFPHashObjectList.Rename	69
4.13.18	TFPHashObjectList.FindInstanceOf	69
4.13.19	TFPHashObjectList.Pack	69
4.13.20	TFPHashObjectList.ShowStatistics	70
4.13.21	TFPHashObjectList.ForEachCall	70
4.13.22	TFPHashObjectList.Capacity	70
4.13.23	TFPHashObjectList.Count	70
4.13.24	TFPHashObjectList.OwnsObjects	71
4.13.25	TFPHashObjectList.Items	71
4.13.26	TFPHashObjectList.List	71
4.14	TFPObjectHashTable	71
4.14.1	Description	71
4.14.2	Method overview	72
4.14.3	Property overview	72
4.14.4	TFPObjectHashTable.Create	72
4.14.5	TFPObjectHashTable.CreateWith	72
4.14.6	TFPObjectHashTable.Add	73
4.14.7	TFPObjectHashTable.Items	73
4.14.8	TFPObjectHashTable.OwnsObjects	73
4.15	TFPObjectList	73
4.15.1	Description	73
4.15.2	Method overview	74
4.15.3	Property overview	74
4.15.4	TFPObjectList.Create	74
4.15.5	TFPObjectList.Destroy	74
4.15.6	TFPObjectList.Clear	75

4.15.7	TFPObjectList.Add	75
4.15.8	TFPObjectList.Delete	75
4.15.9	TFPObjectList.Exchange	76
4.15.10	TFPObjectList.Expand	76
4.15.11	TFPObjectList.Extract	76
4.15.12	TFPObjectList.Remove	76
4.15.13	TFPObjectList.IndexOf	77
4.15.14	TFPObjectList.FindInstanceOf	77
4.15.15	TFPObjectList.Insert	77
4.15.16	TFPObjectList.First	78
4.15.17	TFPObjectList.Last	78
4.15.18	TFPObjectList.Move	78
4.15.19	TFPObjectList.Assign	78
4.15.20	TFPObjectList.Pack	79
4.15.21	TFPObjectList.Sort	79
4.15.22	TFPObjectList.ForEachCall	79
4.15.23	TFPObjectList.Capacity	80
4.15.24	TFPObjectList.Count	80
4.15.25	TFPObjectList.OwnsObjects	80
4.15.26	TFPObjectList.Items	80
4.15.27	TFPObjectList.List	81
4.16	TFPStringHashTable	81
4.16.1	Description	81
4.16.2	Method overview	81
4.16.3	Property overview	81
4.16.4	TFPStringHashTable.Add	81
4.16.5	TFPStringHashTable.Items	81
4.17	THTCustomNode	82
4.17.1	Description	82
4.17.2	Method overview	82
4.17.3	Property overview	82
4.17.4	THTCustomNode.CreateWith	82
4.17.5	THTCustomNode.HasKey	82
4.17.6	THTCustomNode.Key	83
4.18	THTDataNode	83
4.18.1	Description	83
4.18.2	Property overview	83
4.18.3	THTDataNode.Data	83
4.19	THTObjectNode	83
4.19.1	Description	83

4.19.2	Property overview	83
4.19.3	THObjectNode.Data	84
4.20	THOwnedObjectNode	84
4.20.1	Description	84
4.20.2	Method overview	84
4.20.3	THOwnedObjectNode.Destroy	84
4.21	THStringNode	84
4.21.1	Description	84
4.21.2	Property overview	84
4.21.3	THStringNode.Data	85
4.22	TObjectList	85
4.22.1	Description	85
4.22.2	Method overview	85
4.22.3	Property overview	85
4.22.4	TObjectList.create	85
4.22.5	TObjectList.Add	86
4.22.6	TObjectList.Extract	86
4.22.7	TObjectList.Remove	86
4.22.8	TObjectList.IndexOf	87
4.22.9	TObjectList.FindInstanceOf	87
4.22.10	TObjectList.Insert	87
4.22.11	TObjectList.First	87
4.22.12	TObjectList.Last	88
4.22.13	TObjectList.OwnsObjects	88
4.22.14	TObjectList.Items	88
4.23	TObjectQueue	88
4.23.1	Method overview	88
4.23.2	TObjectQueue.Push	89
4.23.3	TObjectQueue.Pop	89
4.23.4	TObjectQueue.Peek	89
4.24	TObjectStack	89
4.24.1	Description	89
4.24.2	Method overview	89
4.24.3	TObjectStack.Push	90
4.24.4	TObjectStack.Pop	90
4.24.5	TObjectStack.Peek	90
4.25	TOrderedList	90
4.25.1	Description	90
4.25.2	Method overview	91
4.25.3	TOrderedList.Create	91

4.25.4	TOrderedList.Destroy	91
4.25.5	TOrderedList.Count	91
4.25.6	TOrderedList.AtLeast	92
4.25.7	TOrderedList.Push	92
4.25.8	TOrderedList.Pop	92
4.25.9	TOrderedList.Peek	92
4.26	TQueue	93
4.26.1	Description	93
4.27	TStack	93
4.27.1	Description	93
5	Reference for unit 'dbugintf'	94
5.1	Writing a debug server	94
5.2	Overview	94
5.3	Constants, types and variables	94
5.3.1	Resource strings	94
5.3.2	Constants	95
5.3.3	Types	95
5.4	Procedures and functions	95
5.4.1	GetDebuggingEnabled	95
5.4.2	InitDebugClient	95
5.4.3	SendBoolean	96
5.4.4	SendDateTime	96
5.4.5	SendDebug	96
5.4.6	SendDebugEx	96
5.4.7	SendDebugFmt	97
5.4.8	SendDebugFmtEx	97
5.4.9	SendInteger	97
5.4.10	SendMethodEnter	98
5.4.11	SendMethodExit	98
5.4.12	SendPointer	98
5.4.13	SendSeparator	99
5.4.14	SetDebuggingEnabled	99
5.4.15	StartDebugServer	99
6	Reference for unit 'dbugmsg'	100
6.1	Used units	100
6.2	Overview	100
6.3	Constants, types and variables	100
6.3.1	Constants	100
6.3.2	Types	101

6.4	Procedures and functions	101
6.4.1	DebugMessageName	101
6.4.2	ReadDebugMessageFromStream	101
6.4.3	WriteDebugMessageToStream	102
7	Reference for unit 'eventlog'	103
7.1	Used units	103
7.2	Overview	103
7.3	Constants, types and variables	103
7.3.1	Resource strings	103
7.3.2	Types	104
7.4	ELogError	105
7.4.1	Description	105
7.5	TEventLog	105
7.5.1	Description	105
7.5.2	Method overview	105
7.5.3	Property overview	105
7.5.4	TEventLog.Destroy	105
7.5.5	TEventLog.EventTypeToString	106
7.5.6	TEventLog.RegisterMessageFile	106
7.5.7	TEventLog.Log	107
7.5.8	TEventLog.Warning	107
7.5.9	TEventLog.Error	107
7.5.10	TEventLog.Debug	108
7.5.11	TEventLog.Info	108
7.5.12	TEventLog.Identification	108
7.5.13	TEventLog.LogType	108
7.5.14	TEventLog.Active	109
7.5.15	TEventLog.DefaultEventType	109
7.5.16	TEventLog.FileName	109
7.5.17	TEventLog.TimeStampFormat	110
7.5.18	TEventLog.CustomLogType	110
7.5.19	TEventLog.EventIDOffset	110
7.5.20	TEventLog.OnGetCustomCategory	111
7.5.21	TEventLog.OnGetCustomEventID	111
7.5.22	TEventLog.OnGetCustomEvent	111
8	Reference for unit 'ezcgi'	112
8.1	Used units	112
8.2	Overview	112
8.3	Constants, types and variables	112

8.3.1	Constants	112
8.4	ECGIException	112
8.4.1	Description	112
8.5	TEZcgi	113
8.5.1	Description	113
8.5.2	Method overview	113
8.5.3	Property overview	113
8.5.4	TEZcgi.Create	113
8.5.5	TEZcgi.Destroy	113
8.5.6	TEZcgi.Run	114
8.5.7	TEZcgi.WriteContent	114
8.5.8	TEZcgi.PutLine	114
8.5.9	TEZcgi.GetValue	115
8.5.10	TEZcgi.DoPost	115
8.5.11	TEZcgi.DoGet	115
8.5.12	TEZcgi.Values	115
8.5.13	TEZcgi.Names	116
8.5.14	TEZcgi.Variables	116
8.5.15	TEZcgi.VariableCount	117
8.5.16	TEZcgi.Name	117
8.5.17	TEZcgi.Email	117
9	Reference for unit 'gettext'	118
9.1	Used units	118
9.2	Overview	118
9.3	Constants, types and variables	118
9.3.1	Constants	118
9.3.2	Types	118
9.4	Procedures and functions	119
9.4.1	GetLanguageIDs	119
9.4.2	TranslateResourceStrings	120
9.4.3	TranslateUnitResourceStrings	120
9.5	EMOFileError	120
9.5.1	Description	120
9.6	TMOFile	120
9.6.1	Description	120
9.6.2	Method overview	121
9.6.3	TMOFile.Create	121
9.6.4	TMOFile.Destroy	121
9.6.5	TMOFile.Translate	121

10 Reference for unit 'idea'	122
10.1 Used units	122
10.2 Overview	122
10.3 Constants, types and variables	122
10.3.1 Constants	122
10.3.2 Types	123
10.4 Procedures and functions	123
10.4.1 CipherIdea	123
10.4.2 DeKeyIdea	123
10.4.3 EnKeyIdea	124
10.5 EIDEAError	124
10.5.1 Description	124
10.6 TIDEADeCryptStream	124
10.6.1 Description	124
10.6.2 Method overview	124
10.6.3 TIDEADeCryptStream.Read	124
10.6.4 TIDEADeCryptStream.Write	125
10.6.5 TIDEADeCryptStream.Seek	125
10.7 TIDEAEncryptStream	125
10.7.1 Description	125
10.7.2 Method overview	126
10.7.3 TIDEAEncryptStream.Destroy	126
10.7.4 TIDEAEncryptStream.Read	126
10.7.5 TIDEAEncryptStream.Write	126
10.7.6 TIDEAEncryptStream.Seek	127
10.7.7 TIDEAEncryptStream.Flush	127
10.8 TIDEAStream	127
10.8.1 Description	127
10.8.2 Method overview	127
10.8.3 Property overview	127
10.8.4 TIDEAStream.Create	128
10.8.5 TIDEAStream.Key	128
11 Reference for unit 'inicol'	129
11.1 Used units	129
11.2 Overview	129
11.3 Constants, types and variables	129
11.3.1 Constants	129
11.4 EIniCol	130
11.4.1 Description	130

11.5	TIniCollection	130
11.5.1	Description	130
11.5.2	Method overview	130
11.5.3	Property overview	130
11.5.4	TIniCollection.Load	130
11.5.5	TIniCollection.Save	131
11.5.6	TIniCollection.SaveToIni	131
11.5.7	TIniCollection.SaveToFile	131
11.5.8	TIniCollection.LoadFromIni	132
11.5.9	TIniCollection.LoadFromFile	132
11.5.10	TIniCollection.Prefix	132
11.5.11	TIniCollection.SectionPrefix	133
11.5.12	TIniCollection.FileName	133
11.5.13	TIniCollection.GlobalSection	133
11.6	TIniCollectionItem	133
11.6.1	Description	133
11.6.2	Method overview	134
11.6.3	Property overview	134
11.6.4	TIniCollectionItem.SaveToIni	134
11.6.5	TIniCollectionItem.LoadFromIni	134
11.6.6	TIniCollectionItem.SaveToFile	134
11.6.7	TIniCollectionItem.LoadFromFile	135
11.6.8	TIniCollectionItem.SectionName	135
11.7	TNamedIniCollection	135
11.7.1	Method overview	135
11.7.2	Property overview	135
11.7.3	TNamedIniCollection.IndexOfUserData	136
11.7.4	TNamedIniCollection.IndexOfName	136
11.7.5	TNamedIniCollection.FindByName	136
11.7.6	TNamedIniCollection.FindByUserData	136
11.7.7	TNamedIniCollection.NamedItems	136
11.8	TNamedIniCollectionItem	136
11.8.1	Property overview	136
11.8.2	TNamedIniCollectionItem.UserData	136
11.8.3	TNamedIniCollectionItem.Name	136
12	Reference for unit 'iostream'	137
12.1	Used units	137
12.2	Overview	137
12.3	Constants, types and variables	137

12.3.1	Types	137
12.4	EIOStreamError	138
12.4.1	Description	138
12.5	TIOStream	138
12.5.1	Description	138
12.5.2	Method overview	138
12.5.3	TIOStream.Create	138
12.5.4	TIOStream.Read	138
12.5.5	TIOStream.Write	139
12.5.6	TIOStream.SetSize	139
12.5.7	TIOStream.Seek	139
13	Reference for unit 'Pipes'	140
13.1	Used units	140
13.2	Overview	140
13.3	Constants, types and variables	140
13.3.1	Constants	140
13.4	Procedures and functions	141
13.4.1	CreatePipeHandles	141
13.4.2	CreatePipeStreams	141
13.5	ENoReadPipe	141
13.5.1	Description	141
13.6	ENoWritePipe	141
13.6.1	Description	141
13.7	EPipeCreation	141
13.7.1	Description	141
13.8	EPipeError	142
13.8.1	Description	142
13.9	EPipeSeek	142
13.9.1	Description	142
13.10	TInputPipeStream	142
13.10.1	Description	142
13.10.2	Method overview	142
13.10.3	Property overview	142
13.10.4	TInputPipeStream.Write	142
13.10.5	TInputPipeStream.Seek	142
13.10.6	TInputPipeStream.Read	143
13.10.7	TInputPipeStream.NumBytesAvailable	143
13.11	TOutputPipeStream	143
13.11.1	Description	143

13.11.2 Method overview	144
13.11.3 TOutputPipeStream.Seek	144
13.11.4 TOutputPipeStream.Read	144
14 Reference for unit 'pooledmm'	145
14.1 Used units	145
14.2 Overview	145
14.3 Constants, types and variables	145
14.3.1 Types	145
14.4 TNonFreePooledMemManager	146
14.4.1 Description	146
14.4.2 Method overview	146
14.4.3 Property overview	146
14.4.4 TNonFreePooledMemManager.Clear	146
14.4.5 TNonFreePooledMemManager.Create	146
14.4.6 TNonFreePooledMemManager.Destroy	147
14.4.7 TNonFreePooledMemManager.NewItem	147
14.4.8 TNonFreePooledMemManager.EnumerateItems	147
14.4.9 TNonFreePooledMemManager.ItemSize	147
14.5 TPooledMemManager	148
14.5.1 Description	148
14.5.2 Method overview	148
14.5.3 Property overview	148
14.5.4 TPooledMemManager.Clear	148
14.5.5 TPooledMemManager.Create	148
14.5.6 TPooledMemManager.Destroy	148
14.5.7 TPooledMemManager.MinimumFreeCount	149
14.5.8 TPooledMemManager.MaximumFreeCountRatio	149
14.5.9 TPooledMemManager.Count	149
14.5.10 TPooledMemManager.FreeCount	150
14.5.11 TPooledMemManager.AllocatedCount	150
14.5.12 TPooledMemManager.FreedCount	150
15 Reference for unit 'process'	151
15.1 Used units	151
15.2 Overview	151
15.3 Constants, types and variables	151
15.3.1 Types	151
15.4 EProcess	153
15.4.1 Description	153
15.5 TProcess	153

15.5.1 Description	153
15.5.2 Method overview	154
15.5.3 Property overview	154
15.5.4 TProcess.Create	155
15.5.5 TProcess.Destroy	155
15.5.6 TProcess.Execute	155
15.5.7 TProcess.CloseInput	156
15.5.8 TProcess.CloseOutput	156
15.5.9 TProcess.CloseStderr	156
15.5.10 TProcess.Resume	156
15.5.11 TProcess.Suspend	157
15.5.12 TProcess.Terminate	157
15.5.13 TProcess.WaitOnExit	157
15.5.14 TProcess.WindowRect	158
15.5.15 TProcess.Handle	158
15.5.16 TProcess.ProcessHandle	158
15.5.17 TProcess.ThreadHandle	158
15.5.18 TProcess.ProcessID	159
15.5.19 TProcess.ThreadID	159
15.5.20 TProcess.Input	159
15.5.21 TProcess.Output	160
15.5.22 TProcess.Stderr	160
15.5.23 TProcess.ExitStatus	160
15.5.24 TProcess.InheritHandles	161
15.5.25 TProcess.Active	161
15.5.26 TProcess.ApplicationName	161
15.5.27 TProcess.CommandLine	161
15.5.28 TProcess.ConsoleTitle	162
15.5.29 TProcess.CurrentDirectory	162
15.5.30 TProcess.Desktop	162
15.5.31 TProcess.Environment	163
15.5.32 TProcess.Options	163
15.5.33 TProcess.Priority	164
15.5.34 TProcess.StartupOptions	164
15.5.35 TProcess.Running	165
15.5.36 TProcess.ShowWindow	165
15.5.37 TProcess.WindowColumns	166
15.5.38 TProcess.WindowHeight	166
15.5.39 TProcess.WindowLeft	166
15.5.40 TProcess.WindowRows	167

15.5.41 TProcess.WindowTop	167
15.5.42 TProcess.WindowWidth	167
15.5.43 TProcess.FillAttribute	168
16 Reference for unit 'streamcoll'	169
16.1 Used units	169
16.2 Overview	169
16.3 Procedures and functions	169
16.3.1 ColReadBoolean	169
16.3.2 ColReadCurrency	170
16.3.3 ColReadDateTime	170
16.3.4 ColReadFloat	170
16.3.5 ColReadInteger	170
16.3.6 ColReadString	171
16.3.7 ColWriteBoolean	171
16.3.8 ColWriteCurrency	171
16.3.9 ColWriteDateTime	171
16.3.10 ColWriteFloat	172
16.3.11 ColWriteInteger	172
16.3.12 ColWriteString	172
16.4 EStreamColl	172
16.4.1 Description	172
16.5 TStreamCollection	172
16.5.1 Description	172
16.5.2 Method overview	173
16.5.3 Property overview	173
16.5.4 TStreamCollection.LoadFromStream	173
16.5.5 TStreamCollection.SaveToStream	173
16.5.6 TStreamCollection.Streaming	173
16.6 TStreamCollectionItem	174
16.6.1 Description	174
17 Reference for unit 'streamex'	175
17.1 Used units	175
17.2 Overview	175
17.3 TBidirBinaryObjectReader	175
17.3.1 Description	175
17.3.2 Property overview	175
17.3.3 TBidirBinaryObjectReader.Position	175
17.4 TBidirBinaryObjectWriter	176
17.4.1 Description	176

17.4.2	Property overview	176
17.4.3	TBidirBinaryObjectWriter.Position	176
17.5	TDelphiReader	176
17.5.1	Description	176
17.5.2	Method overview	176
17.5.3	Property overview	176
17.5.4	TDelphiReader.GetDriver	177
17.5.5	TDelphiReader.ReadStr	177
17.5.6	TDelphiReader.Read	177
17.5.7	TDelphiReader.Position	177
17.6	TDelphiWriter	177
17.6.1	Description	177
17.6.2	Method overview	178
17.6.3	Property overview	178
17.6.4	TDelphiWriter.GetDriver	178
17.6.5	TDelphiWriter.FlushBuffer	178
17.6.6	TDelphiWriter.Write	178
17.6.7	TDelphiWriter.WriteStr	178
17.6.8	TDelphiWriter.WriteValue	179
17.6.9	TDelphiWriter.Position	179
18	Reference for unit 'StreamIO'	180
18.1	Used units	180
18.2	Overview	180
18.3	Procedures and functions	180
18.3.1	AssignStream	180
18.3.2	GetStream	181
19	Reference for unit 'zstream'	182
19.1	Used units	182
19.2	Overview	182
19.3	Constants, types and variables	182
19.3.1	Types	182
19.4	ECompressionError	183
19.4.1	Description	183
19.5	EDecompressionError	183
19.5.1	Description	183
19.6	EZlibError	183
19.6.1	Description	183
19.7	TCompressionStream	183
19.7.1	Description	183

19.7.2	Method overview	183
19.7.3	Property overview	184
19.7.4	TCompressionStream.Create	184
19.7.5	TCompressionStream.Destroy	184
19.7.6	TCompressionStream.Read	184
19.7.7	TCompressionStream.Write	185
19.7.8	TCompressionStream.Seek	185
19.7.9	TCompressionStream.CompressionRate	185
19.7.10	TCompressionStream.OnProgress	185
19.8	TCustomZlibStream	186
19.8.1	Description	186
19.8.2	Method overview	186
19.8.3	TCustomZlibStream.Create	186
19.9	TDecompressionStream	186
19.9.1	Description	186
19.9.2	Method overview	186
19.9.3	Property overview	186
19.9.4	TDecompressionStream.Create	187
19.9.5	TDecompressionStream.Destroy	187
19.9.6	TDecompressionStream.Read	187
19.9.7	TDecompressionStream.Write	187
19.9.8	TDecompressionStream.Seek	188
19.9.9	TDecompressionStream.OnProgress	188
19.10	TGZFileStream	188
19.10.1	Description	188
19.10.2	Method overview	188
19.10.3	TGZFileStream.Create	189
19.10.4	TGZFileStream.Destroy	189
19.10.5	TGZFileStream.Read	189
19.10.6	TGZFileStream.Write	190
19.10.7	TGZFileStream.Seek	190

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 'base64'

1.1 Used units

Table 1.1: Used units by unit 'base64'

Name	Page
Classes	??
sysutils	??

1.2 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` (24) for encoding, and `TBase64DecodingStream` (22) 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...

1.3 Constants, types and variables

1.3.1 Types

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

Table 1.2: Enumeration values for type `TBase64DecodingMode`

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

`TBase64DecodingMode` determines the decoding algorithm used by `TBase64DecodingStream` (22). 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` (22) 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.

1.4 EBase64DecodingException

1.4.1 Description

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

1.5 TBase64DecodingStream

1.5.1 Description

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

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

1.5.2 Method overview

Page	Property	Description
22	Create	Create a new instance of the <code>TBase64DecodingStream</code> class
23	Read	Read and decrypt data from the source stream
23	Reset	Reset the stream
24	Seek	Set stream position.
23	Write	Write data to the stream

1.5.3 Property overview

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

1.5.4 TBase64DecodingStream.Create

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

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

Visibility: public

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

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

See also: `TBase64EncodingStream.Create` (25), `TBase64DecodingMode` (21)

1.5.5 TBase64DecodingStream.Reset

Synopsis: Reset the stream

Declaration: `procedure Reset`

Visibility: `public`

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

Errors: None.

See also: `TBase64DecodingStream.EOF` (24), `TBase64DecodingStream.Read` (23)

1.5.6 TBase64DecodingStream.Read

Synopsis: Read and decrypt data from the source stream

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

Visibility: `public`

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

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

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

See also: `TBase64DecodingStream.Write` (23), `TBase64DecodingStream.Seek` (24), `#rtl.classes.TStream.Read` (??)

1.5.7 TBase64DecodingStream.Write

Synopsis: Write data to the stream

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

Visibility: `public`

Description: `Write` always raises an `EStreamError` exception, because the decoding stream is read-only. To write to an encrypted stream, use a `TBase64EncodingStream` (24) instance.

Errors:

See also: `TBase64DecodingStream.Read` (23), `TBase64DecodingStream.Seek` (24), `TBase64EncodingStream.Write` (25), `#rtl.classes.TStream.Write` (??)

1.5.8 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` (23), `TBase64DecodingStream.Write` (23), `TBase64EncodingStream.Seek` (26), `#rtl.classes.TStream.Seek` (??)

1.5.9 TBase64DecodingStream.EOF

Synopsis:

Declaration: `Property EOF : Boolean`

Visibility: `public`

Access: `Read`

Description:

1.5.10 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` (22)

1.6 TBase64EncodingStream

1.6.1 Description

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

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

1.6.2 Method overview

Page	Property	Description
25	Create	Create a new instance of the <code>TBase64EncodingStream</code> class.
25	Destroy	Remove a <code>TBase64EncodingStream</code> instance from memory
25	Read	Read data from the stream
26	Seek	Position the stream
25	Write	Write data to the stream.

1.6.3 TBase64EncodingStream.Create

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

Declaration: `constructor Create(AOutputStream: TStream)`

Visibility: `public`

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

See also: `TBase64EncodingStream.Destroy` ([25](#)), `TBase64DecodingStream.Create` ([22](#))

1.6.4 TBase64EncodingStream.Destroy

Synopsis: Remove a `TBase64EncodingStream` instance from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

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

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

See also: `TBase64EncodingStream.Create` ([25](#))

1.6.5 TBase64EncodingStream.Read

Synopsis: Read data from the stream

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

Visibility: `public`

Description: `Read` always raises an exception, because the encoding stream is write-only.

See also: `TBase64EncodingStream.Write` ([25](#)), `TBase64EncodingStream.Seek` ([26](#)), `TBase64DecodingStream.Read` ([23](#)), `#rtl.classes.TStream.Read` ([??](#))

1.6.6 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` (26), `TBase64EncodingStream.Read` (25), `TBase64DecodingStream.Write` (23), `#rtl.classes.TStream.Write` (??)

1.6.7 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` (25), `TBase64EncodingStream.Write` (25), `#rtl.classes.TStream.Seek` (??)

Chapter 2

Reference for unit 'bufstream'

2.1 Used units

Table 2.1: Used units by unit 'bufstream'

Name	Page
Classes	??
sysutils	??

2.2 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 (30) which is for reading only, and TWriteBufStream (31) 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.

2.3 Constants, types and variables

2.3.1 Constants

`DefaultBufferCapacity : Integer = 16`

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

2.4 TBufStream

2.4.1 Description

TBufStream is the common ancestor for the TReadBufStream (30) and TWriteBufStream (31) streams. It completely handles the buffer memory management and position management. An in-

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

2.4.2 Method overview

Page	Property	Description
28	Create	Create a new <code>TBufStream</code> instance.
28	Destroy	Destroys the <code>TBufStream</code> instance

2.4.3 Property overview

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

2.4.4 TBufStream.Create

Synopsis: Create a new `TBufStream` instance.

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

Visibility: public

Description: `Create` creates a new `TBufStream` instance. A buffer of size `ACapacity` is allocated, and the `ASource` source (or destination) stream is stored. If no capacity is specified, then `DefaultBufferCapacity` ([27](#)) is used as the capacity.

An instance of `TBufStream` should never be instantiated directly. Instead, an instance of `TReadBufStream` ([30](#)) or `TWriteBufStream` ([31](#)) should be created.

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

See also: `TBufStream.Destroy` ([28](#)), `TReadBufStream` ([30](#)), `TWriteBufStream` ([31](#))

2.4.5 TBufStream.Destroy

Synopsis: Destroys the `TBufStream` instance

Declaration: `destructor Destroy;` `Override`

Visibility: public

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

See also: `TBufStream.Create` ([28](#)), `TReadBufStream` ([30](#)), `TWriteBufStream` ([31](#))

2.4.6 TBufStream.Buffer

Synopsis: The current buffer

Declaration: `Property Buffer : Pointer`

Visibility: public

Access: Read

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

See also: `TBufStream.Create` (28), `TBufStream.Capacity` (29), `TBufStream.BufferSize` (29)

2.4.7 TBufStream.Capacity

Synopsis: Current buffer capacity

Declaration: `Property Capacity : Integer`

Visibility: public

Access: Read, Write

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

See also: `TBufStream.Create` (28), `TBufStream.Buffer` (28), `TBufStream.BufferSize` (29), `TBufStream.BufferPos` (29)

2.4.8 TBufStream.BufferPos

Synopsis: Current buffer position.

Declaration: `Property BufferPos : Integer`

Visibility: public

Access: Read

Description: `BufferPos` 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` (28), `TBufStream.Buffer` (28), `TBufStream.BufferSize` (29), `TBufStream.Capacity` (29)

2.4.9 TBufStream.BufferSize

Synopsis: Amount of data in the buffer

Declaration: `Property BufferSize : Integer`

Visibility: public

Access: Read

Description: `BufferSize` is the actual amount of data in the buffer. This is always less than or equal to the `Capacity` (29).

See also: `TBufStream.Create` (28), `TBufStream.Buffer` (28), `TBufStream.BufferPos` (29), `TBufStream.Capacity` (29)

2.5 TReadBufStream

2.5.1 Description

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

The stream provides limited forward-seek possibilities.

2.5.2 Method overview

Page	Property	Description
30	Read	Reads data from the stream
30	Seek	Set location in the buffer
30	Write	Writes data to the stream

2.5.3 TReadBufStream.Seek

Synopsis: Set location in the buffer

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

Visibility: public

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

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

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

See also: `TWriteBufStream.Seek` ([31](#)), `TReadBufStream.Read` ([30](#)), `TReadBufStream.Write` ([30](#))

2.5.4 TReadBufStream.Read

Synopsis: Reads data from the stream

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

Visibility: public

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

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

See also: `TReadBufStream.Seek` ([30](#)), `TReadBufStream.Read` ([30](#))

2.5.5 TReadBufStream.Write

Synopsis: Writes data to the stream

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

Visibility: public

Description: `Write` always raises an `EStreamError` exception, because the stream is read-only. A `TWriteBufStream` (31) write stream must be used to write data in a buffered way.

See also: `TReadBufStream.Seek` (30), `TReadBufStream.Read` (30)

2.6 TWriteBufStream

2.6.1 Description

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

2.6.2 Method overview

Page	Property	Description
31	<code>Destroy</code>	Remove the <code>TWriteBufStream</code> instance from memory
32	<code>Read</code>	Read data from the stream
31	<code>Seek</code>	Set stream position.
32	<code>Write</code>	Write data to the stream

2.6.3 TWriteBufStream.Destroy

Synopsis: Remove the `TWriteBufStream` instance from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` flushes the buffer and then calls the inherited `Destroy` (28).

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

See also: `TBufStream.Create` (28), `TBufStream.Destroy` (28)

2.6.4 TWriteBufStream.Seek

Synopsis: Set stream position.

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

Visibility: `public`

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

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

Errors:

See also: `TWriteBufStream.Write` (32), `TWriteBufStream.Read` (32), `TReadBufStream.Seek` (30)

2.6.5 TWriteBufStream.Read

Synopsis: Read data from the stream

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

Visibility: public

Description: `Read` always raises an `EStreamError` exception since `TWriteBufStream` is write-only. To read data in a buffered way, `TReadBufStream` (30) should be used.

See also: `TWriteBufStream.Seek` (31), `TWriteBufStream.Write` (32), `TReadBufStream.Read` (30)

2.6.6 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` (31), `TWriteBufStream.Read` (32), `TReadBufStream.Write` (30)

Chapter 3

Reference for unit 'CacheCls'

3.1 Used units

Table 3.1: Used units by unit 'CacheCls'

Name	Page
sysutils	??

3.2 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`

3.3 Constants, types and variables

3.3.1 Resource strings

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

Message shown when an invalid index is passed.

3.3.2 Types

```
PCacheSlot = ^TCacheSlot
```

Pointer to `TCacheSlot` (34) record.

```
PCacheSlotArray = ^TCacheSlotArray
```

Pointer to `TCacheSlotArray` (34) array

```
TCacheSlot = record
```

```

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

```

TCacheSlot is internally used by the TCache (34) class. It represents 1 element in the linked list.

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

TCacheSlotArray is an array of TCacheSlot items. Do not use TCacheSlotArray directly, instead, use PCacheSlotArray (33) 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 (35) call to determine whether the item to be added is a new item or not. The function returns True if the 2 data pointers AData1 and AData2 should be considered equal, or False when they are not.

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

3.4 ECacheError

3.4.1 Description

Exception class used in the cachecls unit.

3.5 TCache

3.5.1 Description

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

The Data (37) property offers indexed access to the array of items. When accessing the array through this property, the MRUSlot (37) property is updated.

3.5.2 Method overview

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

3.5.3 Property overview

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

3.5.4 TCache.Create

Synopsis: Create a new cache class.

Declaration: `constructor Create (ASlotCount: Integer)`

Visibility: `public`

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

See also: `TCache.SlotCount` ([38](#))

3.5.5 TCache.Destroy

Synopsis: Free the TCache class from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

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

See also: `TCache.Create` ([35](#))

3.5.6 TCache.Add

Synopsis: Add a data element to the list.

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

Visibility: `public`

Description: Add checks whether `AData` is already in the list. If so, the item is added to the top of the MRU list. If the item is not yet in the list, then the item is added to the list and placed at the top of the MRU list using the `AddNew` (36) 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` (36), `TCache.FindSlot` (36), `TCache.IndexOf` (36), `TCache.Data` (37), `TCache.MRUSlot` (37)

3.5.7 `TCache.AddNew`

Synopsis: Add a new item to the list.

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

Visibility: public

Description: `AddNew` adds a new item to the list: in difference with the `Add` (35) 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` (35), `TCache.FindSlot` (36), `TCache.IndexOf` (36), `TCache.Data` (37), `TCache.MRUSlot` (37)

3.5.8 `TCache.FindSlot`

Synopsis: Find data pointer in the list

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

Visibility: public

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

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

For this function to work correctly, the `OnIsDataEqual` (38) event must be set.

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

See also: `TCache.IndexOf` (36), `TCache.Add` (35), `TCache.OnIsDataEqual` (38)

3.5.9 `TCache.IndexOf`

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

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

Visibility: public

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

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

For this function to work correctly, the `OnIsDataEqual` (38) event must be set.

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

See also: `TCache.FindSlot` (36), `TCache.Add` (35), `TCache.OnIsDataEqual` (38)

3.5.10 TCache.Remove

Synopsis: Remove a data item from the list.

Declaration: `procedure Remove(AData: Pointer)`

Visibility: `public`

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

Errors: None.

See also: `TCache.FindSlot` (36)

3.5.11 TCache.Data

Synopsis: Indexed access to data items

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

Visibility: `public`

Access: Read,Write

Description: `Data` offers index-based access to the data pointers in the cache. By accessing an item in the list in this manner, the item is moved to the front of the MRU list, i.e. `MRUSlot` (37) will point to the accessed item. The access is both read and write.

The index is zero-based and can maximally be `SlotCount-1` (38). Providing an invalid index will result in an exception.

See also: `TCache.MRUSlot` (37)

3.5.12 TCache.MRUSlot

Synopsis: Most recent item slot.

Declaration: `Property MRUSlot : PCacheSlot`

Visibility: `public`

Access: Read,Write

Description: `MRUSlot` points to the most recent used slot. The most recent used slot is updated when the list is accessed through the `Data` (37) property, or when an item is added to the list with `Add` (35) or `AddNew` (36)

See also: `TCache.Add` (35), `TCache.AddNew` (36), `TCache.Data` (37), `TCache.LRUSlot` (38)

3.5.13 TCache.LRUSlot

Synopsis: Last used item

Declaration: `Property LRUSlot : PCacheSlot`

Visibility: public

Access: Read

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

See also: `TCache.Add` (35), `TCache.AddNew` (36), `TCache.Data` (37), `TCache.MRUSlot` (37)

3.5.14 TCache.SlotCount

Synopsis: Number of slots in the list

Declaration: `Property SlotCount : Integer`

Visibility: public

Access: Read, Write

Description: `SlotCount` is the number of slots in the list. Its initial value is set when the `TCache` instance is created, but this can be changed at any time. If items are added to the list and the list is full, then the number of slots is not increased, but the least used item is dropped from the list. In that case `OnFreeSlot` (39) is called.

See also: `TCache.Create` (35), `TCache.Data` (37), `TCache.Slots` (38)

3.5.15 TCache.Slots

Synopsis: Indexed array to the slots

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

Visibility: public

Access: Read

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

The index is zero-based and can maximally be `SlotCount-1` (38). Providing an invalid index will result in an exception.

See also: `TCache.Data` (37), `TCache.SlotCount` (38)

3.5.16 TCache.OnIsDataEqual

Synopsis: Event to compare 2 items.

Declaration: `Property OnIsDataEqual : TOnIsDataEqual`

Visibility: public

Access: Read, Write

Description: `OnIsDataEqual` is used by `FindSlot` (36) and `IndexOf` (36) to compare items when looking for a particular item. These functions are called by the `Add` (35) 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` (36), `TCache.IndexOf` (36), `TCache.Add` (35)

3.5.17 TCache.OnFreeSlot

Synopsis: Event called when a slot is freed

Declaration: `Property OnFreeSlot : TOnFreeSlot`

Visibility: `public`

Access: `Read,Write`

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

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

See also: `TCache.Add` (35), `TCache.AddNew` (36), `TCache.SlotCount` (38)

Chapter 4

Reference for unit 'contnrs'

4.1 Used units

Table 4.1: Used units by unit 'contnrs'

Name	Page
Classes	??
sysutils	??

4.2 Overview

The contnrs implements various general-purpose classes:

Stacks Stack classes to push/pop pointers or objects

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

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

Class lists lists that manage class pointers instead of pointers.

Stacks Stack classes to push/pop pointers or objects

Queues Classes to manage a FIFO list of pointers or objects

Hash lists General-purpose Hash lists.

4.3 Constants, types and variables

4.3.1 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.

4.3.2 Types

```
PHashItem = ^THashItem
```

PHashItem is a pointer type, pointing to the THashItem (42) record.

```
PHashItemList = ^THashItemList
```

PHashItemList is a pointer to the THashItemList (42). It's used in the TFPHashList (55) as a pointer to the memory area containing the hash item records.

```
PHashTable = ^THashTable
```

PHashTable is a pointer to the THashTable (42). It's used in the TFPHashList (55) as a pointer to the memory area containing the hash values.

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

TDataIteratorMethod is a callback prototype for the TDataHashTable.Iterate (40) method. It is called for each data pointer in the hash list, passing the key (key) and data pointer (item) for each item in the list. If Continue is set to false, the iteration stops.

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

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

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

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

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

THashItemList is an array type, primarily used to be able to define the PHashItemList (41) type. It's used in the TFPHashList (55) class.

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

THashTable defines an array of integers, used to hold hash values. It's mainly used to define the PHashTable (41) class.

```
THTCustomNodeClass = Class of THTCustomNode
```

THTCustomNodeClass is used by THTCustomHashTable (40) to decide which class should be created for elements in the list.

```
THTNode = THTDataNode
```

THTNode is provided for backwards compatibility.

```
TIteratorMethod = TDataIteratorMethod
```

TIteratorMethod is used in an internal TFPHashTable (40) method.

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

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

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

TObjectListCallback is used as the prototype for the TFPObjectList.ForEachCall (79) 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 (79) link call when a plain procedure should be called. The Data argument will contain each of the objects in the list in turn, and the Data argument will contain the data passed to the ForEachCall call.

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

TStringIteratorMethod is the callback prototype for the Iterate (49) method. It is called for each element in the hash table, with the string. If Continue is set to false, the iteration stops.

4.4 Procedures and functions

4.4.1 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` (49) hash class.
It's Robert Sedgwick's "Algorithms in C" hash function.

Errors: None.

See also: `TFPCustomHashTable` (49)

4.5 EDuplicate

4.5.1 Description

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

4.6 EKeyNotFound

4.6.1 Description

Exception raised when a key is not found.

4.7 TClassList

4.7.1 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.

4.7.2 Method overview

Page	Property	Description
44	Add	Add a new class pointer to the list.
44	Extract	Extract a class pointer from the list.
45	First	Return first non-nil class pointer
45	IndexOf	Search for a class pointer in the list.
45	Insert	Insert a new class pointer in the list.
45	Last	Return last non- <code>Nil</code> class pointer
44	Remove	Remove a class pointer from the list.

4.7.3 Property overview

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

4.7.4 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` ([44](#)), `#rtl.classes.tlist.add` (??)

4.7.5 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` ([44](#)), `#rtl.classes.Tlist.Extract` (??)

4.7.6 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` ([44](#)), `#rtl.classes.Tlist.Remove` (??)

4.7.7 TClassList.IndexOf

Synopsis: Search for a class pointer in the list.

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

Visibility: public

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

Errors: None.

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

4.7.8 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` (45), `TClassList.Pack` (43)

4.7.9 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` (45), `TClassList.Pack` (43)

4.7.10 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` (44), `TClassList.Remove` (44)

4.7.11 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` (??)

4.8 TComponentList

4.8.1 Description

`TComponentList` is a `TObjectList` (85) 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` (85), `TComponentList` removes any `TComponent` from the list if the `TComponent` instance was freed externally. It uses the `FreeNotification` mechanism for this.

4.8.2 Method overview

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

4.8.3 Property overview

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

4.8.4 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` (85), `#rtl.classes.TComponent` (??)

4.8.5 TComponentList.Add

Synopsis: Add a component to the list.

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

Visibility: public

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

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

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

See also: TObectList.Add ([40](#))

4.8.6 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 ([85](#)) 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 ([47](#)), TObjectList.Extract ([86](#))

4.8.7 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 ([85](#)) so it only accepts TComponent descendents. It introduces no new behaviour.

Errors: None.

See also: TComponentList.Extract ([47](#)), TObjectList.Remove ([86](#))

4.8.8 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` ([87](#))

4.8.9 `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` ([48](#)), `TObjectList.First` ([87](#))

4.8.10 `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` ([48](#)), `TObjectList.Last` ([88](#))

4.8.11 `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` ([87](#)), `TComponentList.Add` ([47](#)), `TComponentList.Remove` ([47](#))

4.8.12 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` (88)

4.9 TFPCustomHashTable

4.9.1 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` (43).

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

4.9.2 Method overview

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

4.9.3 Property overview

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

4.9.4 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` (43)

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

See also: `TFPCustomHashTable.CreateWith` (50)

4.9.5 TFPCustomHashTable.CreateWith

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

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

Visibility: `public`

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

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

See also: `TFPCustomHashTable.Create` (50)

4.9.6 TFPCustomHashTable.Destroy

Synopsis: Free the hash table.

Declaration: `destructor Destroy; Override`

Visibility: `public`

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

Errors: None.

See also: `TFPCustomHashTable.Destroy` (50), `TFPCustomHashTable.Create` (50), `TFPCustomHashTable.CreateWith` (50), `THTCustomNode.Data` (82)

4.9.7 TFPCustomHashTable.ChangeTableSize

Synopsis: Change the table size of the hash table.

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

Visibility: `public`

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

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

See also: `TFPCustomHashTable.HashTableSize` (52)

4.9.8 TFPCustomHashTable.Clear

Synopsis: Clear the hash table.

Declaration: `procedure Clear; Virtual`

Visibility: `public`

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

Errors: None.

See also: `TFPCustomHashTable.Destroy` (50)

4.9.9 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` (51), `TFPCustomHashTable.Add` (49)

4.9.10 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` (82) 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` (51) method should be used instead.

Errors: None.

See also: `TFPCustomHashTable.Add` (49), `TFPCustomHashTable.Delete` (51)

4.9.11 TFPCustomHashTable.IsEmpty

Synopsis: Check if the hash table is empty.

Declaration: `function IsEmpty : Boolean`

Visibility: `public`

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

Errors:

See also: `TFPCustomHashTable.Count` (52), `TFPCustomHashTable.HashTableSize` (52), `TFPCustomHashTable.AVGChainLen` (53), `TFPCustomHashTable.MaxChainLength` (53)

4.9.12 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 (49). Note that setting the hash function does NOT the hash value of all keys to be recomputed, so changing the value while there are still keys in the table is not a good idea.

See also: TFPCustomHashTable.SetHashFunction (49), TFPCustomHashTable.HashTableSize (52)

4.9.13 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 (51), TFPCustomHashTable.HashTableSize (52), TFPCustomHashTable.AVGChainLen (53), TFPCustomHashTable.MaxChainLength (53)

4.9.14 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 (51), TFPCustomHashTable.Count (52), TFPCustomHashTable.AVGChainLen (53), TFPCustomHashTable.MaxChainLength (53), TFPCustomHashTable.VoidSlots (53), TFPCustomHashTable.Density (54)

4.9.15 TFPCustomHashTable.HashTable

Synopsis: Hash table instance

Declaration: Property HashTable : TFPObjectList

Visibility: public

Access: Read

Description: TFPCustomHashTable is the internal list object (TFPObjectList (73) used for the hash table. Each element in this table is again a TFPObjectList (73) instance or Nil.

4.9.16 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` ([51](#)), `TFPCustomHashTable.Count` ([52](#)), `TFPCustomHashTable.AVGChainLen` ([53](#)), `TFPCustomHashTable.MaxChainLength` ([53](#)), `TFPCustomHashTable.LoadFactor` ([53](#)), `TFPCustomHashTable.Density` ([54](#)), `TFPCustomHashTable.NumberOfCollisions` ([54](#))

4.9.17 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` ([51](#)), `TFPCustomHashTable.Count` ([52](#)), `TFPCustomHashTable.AVGChainLen` ([53](#)), `TFPCustomHashTable.MaxChainLength` ([53](#)), `TFPCustomHashTable.VoidSlots` ([53](#)), `TFPCustomHashTable.Density` ([54](#)), `TFPCustomHashTable.NumberOfCollisions` ([54](#))

4.9.18 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` ([51](#)), `TFPCustomHashTable.Count` ([52](#)), `TFPCustomHashTable.LoadFactor` ([53](#)), `TFPCustomHashTable.MaxChainLength` ([53](#)), `TFPCustomHashTable.VoidSlots` ([53](#)), `TFPCustomHashTable.Density` ([54](#)), `TFPCustomHashTable.NumberOfCollisions` ([54](#))

4.9.19 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` (51), `TFPCustomHashTable.Count` (52), `TFPCustomHashTable.LoadFactor` (53), `TFPCustomHashTable.AvgChainLength` (49), `TFPCustomHashTable.VoidSlots` (53), `TFPCustomHashTable.Density` (54), `TFPCustomHashTable.NumberOfCollisions` (54)

4.9.20 `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` (51), `TFPCustomHashTable.Count` (52), `TFPCustomHashTable.LoadFactor` (53), `TFPCustomHashTable.AvgChainLength` (49), `TFPCustomHashTable.VoidSlots` (53), `TFPCustomHashTable.Density` (54)

4.9.21 `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` (51), `TFPCustomHashTable.Count` (52), `TFPCustomHashTable.LoadFactor` (53), `TFPCustomHashTable.AvgChainLength` (49), `TFPCustomHashTable.VoidSlots` (53), `TFPCustomHashTable.Density` (54)

4.10 `TFPDataHashTable`

4.10.1 Description

`TFPDataHashTable` is a `TFPCustomHashTable` (49) 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` (71), or for string data, `TFPStringHashTable` (81) is more suitable. The data pointers are exposed with their keys through the `Items` (55) property.

4.10.2 Method overview

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

4.10.3 Property overview

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

4.10.4 TFPDataHashTable.Add

Synopsis: Add a data pointer to the list.

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

Visibility: `public`

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

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

See also: TFPDataHashTable.Items ([55](#))

4.10.5 TFPDataHashTable.Items

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

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

Visibility: `public`

Access: Read, Write

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

See also: TFPStringHashTable.Add ([81](#))

4.11 TFPHashList

4.11.1 Description

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

if a base class for an own hash class is wanted, the TFPCustomHashTable ([49](#)) class can be used. If a hash class for objects is needed instead of pointers, the TFPHashObjectList ([65](#)) class can be used.

4.11.2 Method overview

Page	Property	Description
57	Add	Add a new key/data pair to the list
57	Clear	Clear the list
56	Create	Create a new instance of the hashlist
58	Delete	Delete an item from the list.
56	Destroy	Removes an instance of the hashlist from the heap
58	Error	Raise an error
58	Expand	Expand the list
58	Extract	Extract a pointer from the list
59	Find	Find data associated with key
59	FindIndexOf	Return index of named item.
59	FindWithHash	Find first element with given name and hash value
61	ForEachCall	Call a procedure for each element in the list
57	HashOfIndex	Return the hash value of an item by index
59	IndexOf	Return the index of the data pointer
57	NameOfIndex	Returns the key name of an item by index
60	Pack	Remove nil pointers from the list
60	Remove	Remove first instance of a pointer
60	Rename	Rename a key
60	ShowStatistics	Return some statistics for the list.

4.11.3 Property overview

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

4.11.4 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` ([56](#))

4.11.5 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` ([56](#)), `TFPHashList.Clear` ([57](#))

4.11.6 TFPHashList.Add

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

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

Visibility: public

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

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

See also: `TFPHashList.Extract` ([58](#)), `TFPHashList.Remove` ([60](#)), `TFPHashList.Delete` ([58](#))

4.11.7 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` ([58](#)), `TFPHashList.Remove` ([60](#)), `TFPHashList.Delete` ([58](#)), `TFPHashList.Add` ([57](#))

4.11.8 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` ([57](#)), `TFPHashList.Find` ([59](#)), `TFPHashList.FindIndexOf` ([59](#)), `TFPHashList.FindWithHash` ([59](#))

4.11.9 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` (55), `TFPHashList.Find` (59), `TFPHashList.FindIndexOf` (59), `TFPHashList.FindWithHash` (59)

4.11.10 TFPHashList.Delete

Synopsis: Delete an item from the list.

Declaration: `procedure Delete(Index: Integer)`

Visibility: `public`

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

Errors: `TFPHashList.Extract` (58) `TFPHashList.Remove` (60) `TFPHashList.Add` (57)

4.11.11 TFPHashList.Error

Synopsis: Raise an error

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

Visibility: `public`

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

4.11.12 TFPHashList.Expand

Synopsis: Expand the list

Declaration: `function Expand : TFPHashList`

Visibility: `public`

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

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

See also: `TFPHashList.Clear` (57)

4.11.13 TFPHashList.Extract

Synopsis: Extract a pointer from the list

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

Visibility: `public`

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

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

See also: `TFPHashList.Delete` (58), `TFPHashList.Remove` (60), `TFPHashList.Clear` (57)

4.11.14 TFPHashList.IndexOf

Synopsis: Return the index of the data pointer

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

Visibility: public

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

The performed search is linear, and not very efficient.

See also: `TFPHashList.HashOfIndex` (57), `TFPHashList.NameOfIndex` (57), `TFPHashList.Find` (59), `TFPHashList.FindIndexOf` (59), `TFPHashList.FindWithHash` (59)

4.11.15 TFPHashList.Find

Synopsis: Find data associated with key

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

Visibility: public

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

See also: `TFPHashList.HashOfIndex` (57), `TFPHashList.NameOfIndex` (57), `TFPHashList.IndexOf` (59), `TFPHashList.FindIndexOf` (59), `TFPHashList.FindWithHash` (59)

4.11.16 TFPHashList.FindIndexOf

Synopsis: Return index of named item.

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

Visibility: public

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

See also: `TFPHashList.HashOfIndex` (57), `TFPHashList.NameOfIndex` (57), `TFPHashList.IndexOf` (59), `TFPHashList.Find` (59), `TFPHashList.FindWithHash` (59)

4.11.17 TFPHashList.FindWithHash

Synopsis: Find first element with given name and hash value

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

Visibility: public

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

See also: `TFPHashList.HashOfIndex` (57), `TFPHashList.NameOfIndex` (57), `TFPHashList.IndexOf` (59), `TFPHashList.Find` (59), `TFPHashList.FindIndexOf` (59)

4.11.18 TFPHashList.Rename

Synopsis: Rename a key

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

Visibility: public

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

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

4.11.19 TFPHashList.Remove

Synopsis: Remove first instance of a pointer

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

Visibility: public

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

See also: `TFPHashList.Delete` (58), `TFPHashList.Clear` (57), `TFPHashList.Extract` (58)

4.11.20 TFPHashList.Pack

Synopsis: Remove nil pointers from the list

Declaration: `procedure Pack`

Visibility: public

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

See also: `TFPHashList.Clear` (57)

4.11.21 TFPHashList.ShowStatistics

Synopsis: Return some statistics for the list.

Declaration: `procedure ShowStatistics`

Visibility: public

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

HashSize Size of the hash table

HashMean Mean hash value

HashStdDev Standard deviation of hash values

ListSize Size and capacity of the list

StringSize Size and capacity of key strings

4.11.22 TFPHashList.ForEachCall

Synopsis: Call a procedure for each element in the list

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

Visibility: public

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

4.11.23 TFPHashList.Capacity

Synopsis: Capacity of the list.

Declaration: `Property Capacity : Integer`

Visibility: public

Access: Read,Write

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

See also: `TFPHashList.Count` ([61](#)), `TFPHashList.Items` ([61](#))

4.11.24 TFPHashList.Count

Synopsis: Current number of elements in the list.

Declaration: `Property Count : Integer`

Visibility: public

Access: Read,Write

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

See also: `TFPHashList.Capacity` ([61](#)), `TFPHashList.Items` ([61](#))

4.11.25 TFPHashList.Items

Synopsis: Indexed array with pointers

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

Visibility: public

Access: Read,Write

Description: `Items` provides indexed access to the pointers, the index runs from 0 to `Count-1` ([61](#)).

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

See also: `TFPHashList.Capacity` ([61](#)), `TFPHashList.Count` ([61](#))

4.11.26 TFPHashList.List

Synopsis: Low-level hash list

Declaration: `Property List : PHashItemList`

Visibility: public

Access: Read

Description: `List` exposes the low-level item list (42). It should not be used directly.

See also: `TFPHashList.Strs` (62), `THashItemList` (42)

4.11.27 TFPHashList.Strs

Synopsis: Low-level memory area with strings.

Declaration: `Property Strs : PChar`

Visibility: public

Access: Read

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

See also: `TFPHashList.List` (62)

4.12 TFPHashObject

4.12.1 Description

`TFPHashObject` is a `TObject` descendent which is aware of the `TFPHashObjectList` (65) 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.

4.12.2 Method overview

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

4.12.3 Property overview

Page	Property	Access	Description
64	<code>Hash</code>	r	Hash value
64	<code>Name</code>	r	Current name of the object

4.12.4 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` (65) hash list. It also has no name when created in this way.

See also: `TFPHashObject.Name` (64), `TFPHashObject.ChangeOwner` (63), `TFPHashObject.ChangeOwnerAndName` (63)

4.12.5 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` (65).

See also: `TFPHashObject.CreateNotOwned` (63), `TFPHashObject.ChangeOwner` (63), `TFPHashObject.Name` (64)

4.12.6 TFPHashObject.ChangeOwner

Synopsis: Change the list owning the object.

Declaration: `procedure ChangeOwner (HashObjectList: TFPHashObjectList)`

Visibility: `public`

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

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

See also: `TFPHashObject.ChangeOwnerAndName` (63), `TFPHashObject.Name` (64)

4.12.7 TFPHashObject.ChangeOwnerAndName

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

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

Visibility: `public`

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

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

See also: `TFPHashObject.ChangeOwner` (63), `TFPHashObject.Name` (64)

4.12.8 TFPHashObject.Rename

Synopsis: Rename the object

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

Visibility: public

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

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

See also: `TFPHashObject.ChangeOwner` (63), `TFPHashObject.ChangeOwnerAndName` (63), `TFPHashObject.Name` (64)

4.12.9 TFPHashObject.Name

Synopsis: Current name of the object

Declaration: `Property Name : shortstring`

Visibility: public

Access: Read

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

See also: `TFPHashObject.Rename` (64), `TFPHashObject.ChangeOwnerAndName` (63)

4.12.10 TFPHashObject.Hash

Synopsis: Hash value

Declaration: `Property Hash : LongWord`

Visibility: public

Access: Read

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

See also: `TFPHashObject.Name` (64)

4.13 TFPHashObjectList

4.13.1 Method overview

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

4.13.2 Property overview

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

4.13.3 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` ([65](#)), `TFPHashObjectList.OwnsObjects` ([71](#))

4.13.4 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` (65), `TFPHashObjectList.Clear` (66)

4.13.5 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` (71) is `True`. It always frees all memory needed to contain the objects.

Errors: None.

See also: `TFPHashObjectList.Extract` (67), `TFPHashObjectList.Remove` (68), `TFPHashObjectList.Delete` (67), `TFPHashObjectList.Add` (66)

4.13.6 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` (67), `TFPHashObjectList.Remove` (68), `TFPHashObjectList.Delete` (67)

4.13.7 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` (67), `TFPHashObjectList.Find` (68), `TFPHashObjectList.FindIndexOf` (68), `TFPHashObjectList.FindWithHash` (69)

4.13.8 TFPHashObjectList.HashOfIndex

Synopsis: Return the hash valye of an object by index

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

Visibility: public

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

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

See also: `TFPHashObjectList.HashOfName` (65), `TFPHashObjectList.Find` (68), `TFPHashObjectList.FindIndexOf` (68), `TFPHashObjectList.FindWithHash` (69)

4.13.9 TFPHashObjectList.Delete

Synopsis: Delete an object from the list.

Declaration: `procedure Delete(Index: Integer)`

Visibility: public

Description: `Delete` deletes the object at position `Index`. If `OwnsObjects` (71) is `True`, then the object itself is also freed from memory.

See also: `TFPHashObjectList.Extract` (67), `TFPHashObjectList.Remove` (68), `TFPHashObjectList.Add` (66), `TFPHashObjectList.OwnsObjects` (71)

4.13.10 TFPHashObjectList.Expand

Synopsis: Expand the list

Declaration: `function Expand : TFPHashObjectList`

Visibility: public

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

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

See also: `TFPHashObjectList.Clear` (66)

4.13.11 TFPHashObjectList.Extract

Synopsis: Extract a object instance from the list

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

Visibility: public

Description: `Extract` removes the data object from the list, if it is in the list. It returns the object instance if it was removed from the list, `Nil` otherwise. The object is *not* freed from memory, regardless of the value of `OwnsObjects` (71).

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

See also: `TFPHashObjectList.Delete` (67), `TFPHashObjectList.Remove` (68), `TFPHashObjectList.Clear` (66)

4.13.12 TFPHashObjectList.Remove

Synopsis: Remove first occurrence of a object instance

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

Visibility: public

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

If `OwnsObjects (71)` is `True`, then the object itself is also freed from memory.

See also: `TFPHashObjectList.Delete (67)`, `TFPHashObjectList.Clear (66)`, `TFPHashObjectList.Extract (67)`

4.13.13 TFPHashObjectList.IndexOf

Synopsis: Return the index of the object instance

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

Visibility: public

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

The performed search is linear, and not very efficient.

See also: `TFPHashObjectList.HashOfIndex (67)`, `TFPHashObjectList.NameOfIndex (66)`, `TFPHashObjectList.Find (68)`, `TFPHashObjectList.FindIndexOf (68)`, `TFPHashObjectList.FindWithHash (69)`

4.13.14 TFPHashObjectList.Find

Synopsis: Find data associated with key

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

Visibility: public

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

See also: `TFPHashObjectList.HashOfIndex (67)`, `TFPHashObjectList.NameOfIndex (66)`, `TFPHashObjectList.IndexOf (68)`, `TFPHashObjectList.FindIndexOf (68)`, `TFPHashObjectList.FindWithHash (69)`

4.13.15 TFPHashObjectList.FindIndexOf

Synopsis: Return index of named object.

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

Visibility: public

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

See also: `TFPHashObjectList.HashOfIndex (67)`, `TFPHashObjectList.NameOfIndex (66)`, `TFPHashObjectList.IndexOf (68)`, `TFPHashObjectList.Find (68)`, `TFPHashObjectList.FindWithHash (69)`

4.13.16 TFPHashObjectList.FindWithHash

Synopsis: Find first element with given name and hash value

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

Visibility: public

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

See also: `TFPHashObjectList.HashOfIndex` (67), `TFPHashObjectList.NameOfIndex` (66), `TFPHashObjectList.IndexOf` (68), `TFPHashObjectList.Find` (68), `TFPHashObjectList.FindIndexOf` (68)

4.13.17 TFPHashObjectList.Rename

Synopsis: Rename a key

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

Visibility: public

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

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

4.13.18 TFPHashObjectList.FindInstanceOf

Synopsis: Search an instance of a certain class

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

Visibility: public

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

4.13.19 TFPHashObjectList.Pack

Synopsis: Remove nil object instances from the list

Declaration: `procedure Pack`

Visibility: public

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

See also: `TFPHashObjectList.Clear` (66)

4.13.20 TFPHashObjectList.ShowStatistics

Synopsis: Return some statistics for the list.

Declaration: `procedure ShowStatistics`

Visibility: `public`

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

HashSizeSize of the hash table

HashMeanMean hash value

HashStdDevStandard deviation of hash values

ListSizeSize and capacity of the list

StringSizeSize and capacity of key strings

4.13.21 TFPHashObjectList.ForEachCall

Synopsis: Call a procedure for each object in the list

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

Visibility: `public`

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

4.13.22 TFPHashObjectList.Capacity

Synopsis: Capacity of the list.

Declaration: `Property Capacity : Integer`

Visibility: `public`

Access: Read,Write

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

See also: `TFPHashObjectList.Count` (70), `TFPHashObjectList.Items` (71)

4.13.23 TFPHashObjectList.Count

Synopsis: Current number of elements in the list.

Declaration: `Property Count : Integer`

Visibility: `public`

Access: Read,Write

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

See also: `TFPHashObjectList.Capacity` (70), `TFPHashObjectList.Items` (71)

4.13.24 TFPHashObjectList.OwnsObjects

Synopsis: Does the list own the objects it contains

Declaration: `Property OwnsObjects : Boolean`

Visibility: public

Access: Read,Write

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

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

See also: `TFPHashObjectList.Create` (65)

4.13.25 TFPHashObjectList.Items

Synopsis: Indexed array with object instances

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

Visibility: public

Access: Read,Write

Description: `Items` provides indexed access to the object instances, the index runs from 0 to `Count-1` (70).

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

See also: `TFPHashObjectList.Capacity` (70), `TFPHashObjectList.Count` (70)

4.13.26 TFPHashObjectList.List

Synopsis: Low-level hash list

Declaration: `Property List : TFPHashList`

Visibility: public

Access: Read

Description: `List` exposes the low-level hash list (55). It should not be used directly.

See also: `TFPHashList` (55)

4.14 TFPObjectHashTable

4.14.1 Description

`TFPStringHashTable` is a `TFPCustomHashTable` (49) 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` (81), or for arbitrary pointer data, `TFPDataHashTable` (54) is more suitable. The objects are exposed with their keys through the `Items` (73) property.

4.14.2 Method overview

Page	Property	Description
73	Add	Add a new object to the hash table
72	Create	Create a new instance of TFPOjectHashTable
72	CreateWith	Create a new hash table with given size and hash function

4.14.3 Property overview

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

4.14.4 TFPOjectHashTable.Create

Synopsis: Create a new instance of TFPOjectHashTable

Declaration: constructor Create(AOwnsObjects: Boolean)

Visibility: public

Description: Create creates a new instance of TFPOjectHashTable on the heap. It sets the OwnsObjects ([73](#)) property to AOwnsObjects, and then calls the inherited Create. If AOwnsObjects is set to True, then the hash table owns the objects: whenever an object is removed from the list, it is automatically freed.

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

See also: TFPOjectHashTable.OwnsObjects ([73](#)), TFPOjectHashTable.CreateWith ([72](#)), TFPOjectHashTable.Items ([73](#))

4.14.5 TFPOjectHashTable.CreateWith

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

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

Visibility: public

Description: CreateWith sets the OwnsObjects ([73](#)) property to AOwnsObjects, and then calls the inherited CreateWith. If AOwnsObjects is set to True, then the hash table owns the objects: whenever an object is removed from the list, it is automatically freed.

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

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

See also: TFPOjectHashTable.OwnsObjects ([73](#)), TFPOjectHashTable.Create ([72](#)), TFPOjectHashTable.Items ([73](#))

4.14.6 TFObjectHashTable.Add

Synopsis: Add a new object to the hash table

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

Visibility: public

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

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

See also: TFObjectHashTable.Items (73)

4.14.7 TFObjectHashTable.Items

Synopsis: Key-based access to the objects

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

Visibility: public

Access: Read, Write

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

See also: TFObjectHashTable.Add (73)

4.14.8 TFObjectHashTable.OwnsObjects

Synopsis: Does the hash table own the objects ?

Declaration: `Property OwnsObjects : Boolean`

Visibility: public

Access: Read, Write

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

See also: TFObjectHashTable.Create (72), TFObjectHashTable.Items (73)

4.15 TFObjectList

4.15.1 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 (85), 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).

4.15.2 Method overview

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

4.15.3 Property overview

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

4.15.4 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` ([74](#)), `TFObjectList.OwnsObjects` ([80](#)), `TObjectList` ([85](#))

4.15.5 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` (80) is `True`.

See also: `TFPObjectList.OwnsObjects` (80), `TObjectList.Create` (85)

4.15.6 TFPObjectList.Clear

Synopsis: Clear all elements in the list.

Declaration: `procedure Clear`

Visibility: public

Description: Removes all objects from the list, freeing all objects in the list if `OwnsObjects` (80) is `True`.

See also: `TObjectList.Destroy` (85)

4.15.7 TFPObjectList.Add

Synopsis: Add an object to the list.

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

Visibility: public

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

Note that when `OwnsObjects` (80) is `True`, an object should not be added twice to the list: this will result in memory corruption when the object is freed (as it will be freed twice). The `Add` method does not check this, however.

Errors: None.

See also: `TFPObjectList.OwnsObjects` (80), `TFPObjectList.Delete` (75)

4.15.8 TFPObjectList.Delete

Synopsis: Delete an element from the list.

Declaration: `procedure Delete(Index: Integer)`

Visibility: public

Description: `Delete` removes the object at index `Index` from the list. When `OwnsObjects` (80) is `True`, the object is also freed.

Errors: An access violation may occur when `OwnsObjects` (80) is `True` and either the object was freed externally, or when the same object is in the same list twice.

See also: `TTFPObjectList.Remove` (40), `TFPObjectList.Extract` (76), `TFPObjectList.OwnsObjects` (80), `TTFPObjectList.Add` (40), `TTFPObjectList.Clear` (40)

4.15.9 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` (40), `TFObjectList.Delete` (40)

4.15.10 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` (79), `TFObjectList.Clear` (75), `#rtl.classes.tfplist.expand (??)`

4.15.11 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` (79), `TFObjectList.Clear` (75), `TFObjectList.Remove` (76), `TFObjectList.Delete` (75)

4.15.12 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` (80) is `True`, and returns the index of the object that was found in the list, or -1 if the object was not found.

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

Errors: None.

See also: `TFPObjectList.Pack` ([79](#)), `TFPObjectList.Clear` ([75](#)), `TFPObjectList.Delete` ([75](#)), `TFPObjectList.Extract` ([76](#))

4.15.13 TFPObjectList.IndexOf

Synopsis: Search for an object in the list

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

Visibility: public

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

Errors: None.

See also: `TFPObjectList.Items` ([80](#)), `TFPObjectList.Remove` ([76](#)), `TFPObjectList.Extract` ([76](#))

4.15.14 TFPObjectList.FindInstanceOf

Synopsis: Search for an instance of a certain class

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

Visibility: public

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

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

Errors: None.

See also: `TFPObjectList.IndexOf` ([77](#))

4.15.15 TFPObjectList.Insert

Synopsis: Insert a new object in the list

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

Visibility: public

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

Errors: None.

See also: `TFPObjectList.Add` ([75](#)), `TFPObjectList.Delete` ([75](#))

4.15.16 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` (78), `TFObjectList.Pack` (79)

4.15.17 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` (78), `TFObjectList.Pack` (79)

4.15.18 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` (76), 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` (76), `TFObjectList.Delete` (75), `TFObjectList.Insert` (77)

4.15.19 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.

4.15.20 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` ([78](#)), `TFObjectList.Last` ([78](#))

4.15.21 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` ([??](#))

4.15.22 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` ([42](#)), `TObjectListCallback` ([42](#))

4.15.23 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` (80).

See also: `TFObjectList.Count` (80)

4.15.24 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` (80)

4.15.25 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` (74), `TFObjectList.Delete` (75), `TFObjectList.Remove` (76), `TFObjectList.Clear` (75)

4.15.26 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` (80)

4.15.27 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` (??)

4.16 TFPStringHashTable

4.16.1 Description

`TFPStringHashTable` is a `TFPCustomHashTable` (49) descendent which stores simple strings together with the keys. In case the data associated with the keys are objects, it's better to use `TFPObjectHashTable` (71), or for arbitrary pointer data, `TFPDataHashTable` (54) is more suitable. The strings are exposed with their keys through the `Items` (81) property.

4.16.2 Method overview

Page	Property	Description
81	<code>Add</code>	Add a new string to the hash list

4.16.3 Property overview

Page	Property	Access	Description
81	<code>Items</code>	rw	Key based access to the strings in the hash table

4.16.4 TFPStringHashTable.Add

Synopsis: Add a new string to the hash list

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

Visibility: public

Description: `Add` adds a new string `AItem` to the hash list with key `AKey`.

Errors: If a string with key `Akey` already exists in the hash table, an exception will be raised.

See also: `TFPStringHashTable.Items` (81)

4.16.5 TFPStringHashTable.Items

Synopsis: Key based access to the strings in the hash table

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

Visibility: public

Access: Read,Write

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

See also: `TFPStringHashTable.Add` ([81](#))

4.17 THTCustomNode

4.17.1 Description

`THTCustomNode` is used by the `TFPCustomHashTable` ([49](#)) class to store the keys and associated values.

4.17.2 Method overview

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

4.17.3 Property overview

Page	Property	Access	Description
83	<code>Key</code>	<code>r</code>	Key value associated with this hash item.

4.17.4 THTCustomNode.CreateWith

Synopsis: Create a new instance of `THTCustomNode`

Declaration: `constructor CreateWith(const AString: String)`

Visibility: `public`

Description: `CreateWith` creates a new instance of `THTCustomNode` and stores the string `AString` in it. It should never be necessary to call this method directly, it will be called by the `TFPHashTable` ([40](#)) class when needed.

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

See also: `TFPHashTable` ([40](#))

4.17.5 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` ([83](#))

4.17.6 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` ([82](#))

4.18 THTDataNode

4.18.1 Description

`THTDataNode` is used by `TDataHashTable` ([40](#)) 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`

4.18.2 Property overview

Page	Property	Access	Description
83	<code>Data</code>	rw	Data pointer

4.18.3 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.

4.19 THTObjectNode

4.19.1 Description

`THTObjectNode` is a `THTCustomNode` ([82](#)) descendent which holds the data in the `TFPObjectHashTable` ([71](#)) hash table. It exposes a data string.

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

4.19.2 Property overview

Page	Property	Access	Description
84	<code>Data</code>	rw	Object instance

4.19.3 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` (73)

See also: `TFPObjectHashTable` (71), `TFPObjectHashTable.Items` (73), `THTOwnedObjectNode` (84)

4.20 THTOwnedObjectNode

4.20.1 Description

`THTOwnedObjectNode` is used instead of `THTObjectNode` (83) in case `TFPObjectHashTable` (71) owns it's objects. When this object is destroyed, the associated data object is also destroyed.

4.20.2 Method overview

Page	Property	Description
84	<code>Destroy</code>	Destroys the node and the object.

4.20.3 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` (84), `TFPObjectHashTable.OwnsObjects` (73)

4.21 THTStringNode

4.21.1 Description

`THTStringNode` is a `THTCustomNode` (82) descendent which holds the data in the `TFPStringHashTable` (81) hash table. It exposes a data string.

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

4.21.2 Property overview

Page	Property	Access	Description
85	<code>Data</code>	<code>rw</code>	String data

4.21.3 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` (81)

See also: `TFPStringHashTable` (81)

4.22 TObjectList

4.22.1 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` (73), `TObjectList` offers a notification mechanism of list change operations: insert, delete. This slows down bulk operations, so if the notifications are not needed, `TObjectList` may be more appropriate.

4.22.2 Method overview

Page	Property	Description
86	Add	Add an object to the list.
85	create	Create a new object list.
86	Extract	Extract an object from the list.
87	FindInstanceOf	Search for an instance of a certain class
87	First	Return the first non-nil object in the list
87	IndexOf	Search for an object in the list
87	Insert	Insert an object in the list.
88	Last	Return the last non-nil object in the list.
86	Remove	Remove (and possibly free) an element from the list.

4.22.3 Property overview

Page	Property	Access	Description
88	Items	rw	Indexed access to the elements of the list.
88	OwnsObjects	rw	Should the list free elements when they are removed.

4.22.4 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` (85), `TObjectList.OwnsObjects` (88), `TFPObjectList` (73)

4.22.5 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` (87), `#rtl.classes.TList.Delete` (??), `TObjectList.Extract` (86), `TObjectList.Remove` (86)

4.22.6 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` (86), `Extract` does not free the extracted element if `OwnsObjects` (88) 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` (86)

4.22.7 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` (88) 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` (86)

4.22.8 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` (87)

4.22.9 TObjectList.FindInstanceOf

Synopsis: Search for an instance of a certain class

Declaration: `function FindInstanceOf(AClass: TClass; AExact: Boolean;
ASearchAt: 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` (87)

4.22.10 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` (86), `TObjectList.Remove` (86)

4.22.11 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` (88), `TObjectList.Pack` (85)

4.22.12 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` (87), `TObjectList.Pack` (85)

4.22.13 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` (85), `TObjectList.Delete` (85), `TObjectList.Remove` (86), `TObjectList.Clear` (85)

4.22.14 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` (??)

4.23 TObjectQueue

4.23.1 Method overview

Page	Property	Description
89	<code>Peek</code>	Look at the first object in the queue.
89	<code>Pop</code>	Pop the first element off the queue
89	<code>Push</code>	Push an object on the queue

4.23.2 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` ([89](#)), `TObjectQueue.Peek` ([89](#))

4.23.3 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` ([89](#)), `TObjectQueue.Peek` ([89](#))

4.23.4 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` ([89](#)), `TObjectQueue.Pop` ([89](#))

4.24 TObjectStack

4.24.1 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.

4.24.2 Method overview

Page	Property	Description
90	Peek	Look at the top object in the stack.
90	Pop	Pop the top object of the stack.
90	Push	Push an object on the stack.

4.24.3 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` (90), `TObjectStack.Peek` (90)

4.24.4 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` (90), `TObjectStack.Peek` (90)

4.24.5 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` (90), `TObjectStack.Pop` (90)

4.25 TOrderedList

4.25.1 Description

`TOrderedList` provides the base class for `TQueue` (93) and `TStack` (93). 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.

4.25.2 Method overview

Page	Property	Description
92	AtLeast	Check whether the list contains a certain number of elements.
91	Count	Number of elements on the list.
91	Create	Create a new ordered list
91	Destroy	Free an ordered list
92	Peek	Return the next element to be popped from the list.
92	Pop	Remove an element from the list.
92	Push	Push another element on the list.

4.25.3 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` ([91](#))

4.25.4 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` ([91](#))

4.25.5 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` ([92](#))

4.25.6 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` ([91](#))

4.25.7 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` ([92](#)), `TOrderedList.Peek` ([92](#))

4.25.8 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` ([92](#)), `TOrderedList.Push` ([92](#))

4.25.9 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` ([92](#)), without actually popping it from the list.

Errors: None.

See also: `TOrderedList.Pop` ([92](#)), `TOrderedList.Push` ([92](#))

4.26 TQueue

4.26.1 Description

TQueue is a descendent of TOrderedList (90) which implements Push (92) and Pop (92) 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 (90)

4.27 TStack

4.27.1 Description

TStack is a descendent of TOrderedList (90) which implements Push (92) and Pop (92) 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 (90)

Chapter 5

Reference for unit 'dbugintf'

5.1 Writing a debug server

Writing a debug server is relatively easy. It should instantiate a `TSimpleIPCTServer` class from the `SimpleIPC` (94) 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` (94) 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` (94) unit should also be functional.

5.2 Overview

Use `dbugintf` to add debug messages to your application. The messages are not sent to standard output, but are sent to a debug server process which collects messages from various clients and displays them somehow on screen.

The unit is transparant in its use: it does not need initialization, it will start the debug server by itself if it can find it: the program should be called `debugserver` and should be in the `PATH`. When the first debug message is sent, the unit will initialize itself.

The FCL contains a sample debug server (`dbugsrv`) which can be started in advance, and which writes debug message to the console (both on Windows and Linux). The Lazarus project contains a visual application which displays the messages in a GUI.

The `dbugintf` unit relies on the `SimpleIPC` (94) 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` (94) unit should also be functional.

5.3 Constants, types and variables

5.3.1 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.

5.3.2 Constants

`SendError : String = ''`

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

5.3.3 Types

`TDebugLevel = (dlInformation, dlWarning, dlError)`

Table 5.1: 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.

5.4 Procedures and functions

5.4.1 GetDebuggingEnabled

Declaration: `function GetDebuggingEnabled : Boolean`

Visibility: default

5.4.2 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` (96) call will initialize the debug client when it is first called.

Errors: None.

See also: `SendDebug` (96), `StartDebugServer` (99)

5.4.3 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` (96) which sends the name and value of a boolean value as an informational message.

Errors: None.

See also: `SendDebug` (96), `SendDateTime` (96), `SendInteger` (97), `SendPointer` (98)

5.4.4 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` (96) 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` (96), `SendBoolean` (96), `SendInteger` (97), `SendPointer` (98)

5.4.5 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` (95).

See also: `SendDebugEx` (96), `SendDebugFmt` (97), `SendDebugFmtEx` (97)

5.4.6 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` (96) uses informational messages.

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

Errors: None.

See also: `SendDebug` (96), `SendDebugFmt` (97), `SendDebugFmtEx` (97)

5.4.7 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` (96). It exists mainly to avoid the `Format` call in calling code.

Errors: None.

See also: `SendDebug` (96), `SendDebugEx` (96), `SendDebugFmtEx` (97), `#rtl.sysutils.format` (??)

5.4.8 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` (96) with Debug level `MType`. It exists mainly to avoid the `Format` call in calling code.

Errors: None.

See also: `SendDebug` (96), `SendDebugEx` (96), `SendDebugFmt` (97), `#rtl.sysutils.format` (??)

5.4.9 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` (96) 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` (96), `SendBoolean` (96), `SendDateTime` (96), `SendPointer` (98)

5.4.10 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` (98), 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` (96), `SendMethodExit` (98), `SendSeparator` (99)

5.4.11 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` (98), 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` (96), `SendMethodEnter` (98), `SendSeparator` (99)

5.4.12 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` (96) 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` (96), `SendBoolean` (96), `SendDateTime` (96), `SendInteger` (97)

5.4.13 SendSeparator

Synopsis: Send a separator message

Declaration: `procedure SendSeparator`

Visibility: `default`

Description: `SendSeparator` is a simple wrapper around `SendDebug` (96) 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` (96), `SendMethodEnter` (98), `SendMethodExit` (98)

5.4.14 SetDebuggingEnabled

Declaration: `procedure SetDebuggingEnabled(const AValue: Boolean)`

Visibility: `default`

5.4.15 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` (96) call will attempt to start the server by itself if it is not yet running.

Errors: On error, `False` is returned.

See also: `SendDebug` (96), `InitDebugClient` (95)

Chapter 6

Reference for unit 'dbugmsg'

6.1 Used units

Table 6.1: Used units by unit 'dbugmsg'

Name	Page
Classes	??

6.2 Overview

dbugmsg is an auxiliary unit used in the dbugintf ([94](#)) unit. It defines the message protocol used between the debug unit and the debug server.

6.3 Constants, types and variables

6.3.1 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.

6.3.2 Types

```
TDebugMessage = record
  MsgType : Integer;
  MsgTimeStamp : TDateTime;
  Msg : String;
end
```

`TDebugMessage` is a record that describes the message passed from the client to the server. It should not be passed directly in shared memory, as the string containing the message is allocated on the heap. Instead, the `WriteDebugMessageToStream` (102) and `ReadDebugMessageFromStream` (101) can be used to read or write the message from/to a stream.

6.4 Procedures and functions

6.4.1 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` (101) 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.

6.4.2 ReadDebugMessageFromStream

Synopsis: Read a message from stream

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

Visibility: default

Description: `ReadDebugMessageFromStream` reads a `TDebugMessage` (101) 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` (101), `WriteDebugMessageToStream` (102)

6.4.3 WriteDebugMessageToStream

Synopsis: Write a message to stream

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

Visibility: default

Description: `WriteDebugMessageFromStream` writes a `TDebugMessage` (101) 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` (101), `ReadDebugMessageToStream` (100)

Chapter 7

Reference for unit 'eventlog'

7.1 Used units

Table 7.1: Used units by unit 'eventlog'

Name	Page
Classes	??
sysutils	??

7.2 Overview

The EventLog unit implements the TEventLog ([105](#)) 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.

7.3 Constants, types and variables

7.3.1 Resource strings

`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

7.3.2 Types

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

Table 7.2: Enumeration values for type TEventType

Value	Explanation
etCustom	Custom event type.
etDebug	Debug event
etError	Error event
etInfo	Informational event
etWarning	Warning event

TEventType determines the type of event. Depending on the system logger, the log event may end up in different places, or may be displayed in a different manner. A suitable mapping is shown for each system. In the case of Windows, the formatting of the message is done differently, and a different icon is shown for each type of message.

```
TLogCategoryEvent = procedure(Sender: TObject; var Code: Word) of object
```

TLogCategoryEvent is the event type for the TEventLog.OnGetCustomCategory (111) 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 (111) and OnGetCustomEventID (111) 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 7.3: Enumeration values for type TLogType

Value	Explanation
ltFile	Write to file
ltSystem	Use the system log

TLogType determines where the log messages are written. It is the type of the TEventLog.LogType (108) 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.

7.4 ELogError

7.4.1 Description

ELogError is the exception used in the TEventLog (105) component to indicate errors.

7.5 TEventLog

7.5.1 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 (107) call, or the specialized Warning (107), Error (107), Info (108) or Debug (108) calls, which have the event type predefined.

7.5.2 Method overview

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

7.5.3 Property overview

Page	Property	Access	Description
109	Active	rw	Activate the log mechanism
110	CustomLogType	rw	Custom log type ID
109	DefaultEventType	rw	Default event type for the Log (107) call.
110	EventIDOffset	rw	Offset for event ID messages identifiers
109	FileName	rw	File name for log file
108	Identification	rw	Identification string for messages
108	LogType	rw	Log type
111	OnGetCustomCategory	rw	Event to retrieve custom message category
111	OnGetCustomEvent	rw	Event to retrieve custom event Code
111	OnGetCustomEventID	rw	Event to retrieve custom event ID
110	TimeStampFormat	rw	Format for the timestamp string

7.5.4 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 (109) property to False.

See also: `TEventLog.Active` ([109](#))

7.5.5 TEventLog.EventTypeToString

Synopsis: Create a string representation of an event type

Declaration: `function EventTypeToString(E: TEventType) : String`

Visibility: public

Description: `EventTypeToString` converts the event type `E` to a suitable string representation for logging purposes. It's mainly used when writing messages to file, as the system log usually has it's own mechanisms for displaying the various event types.

See also: `TEventType` ([104](#))

7.5.6 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` ([110](#)) (default is 1000) and each string should have 1 placeholder.

Free Pascal comes with a `fclel.res` resource file which contains default values for the 8 strings, in english. It can be linked in the application binary with the statement

```
{ $R fclel.res }
```

This file is generated from the `fclel.mc` and `fclel.rc` files that are distributed with the Free Pascal sources.

If the strings are not registered, windows will still display the event messages, but they will not be formatted nicely.

Note that while any messages logged with the event logger are displayed in the event viewern Windows locks the file registered here. This usually means that the binary is locked.

On non-windows operating systems, this call is ignored.

Errors: If `AFileName` is invalid, false is returned.

7.5.7 TEventLog.Log

Synopsis: Log a message to the system log.

Declaration: `procedure Log(EventType: TEventType;Msg: String); Overload`
`procedure Log(EventType: TEventType;Fmt: String;Args: Array of const)`
`; Overload`
`procedure Log(Msg: String); Overload`
`procedure Log(Fmt: String;Args: Array of const); Overload`

Visibility: public

Description: Log sends a log message to the system log. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters. If `EventType` is specified, then it is used as the message event type. If `EventType` is omitted, then the event type is determined from `Default-EventType` (109).

If `EventType` is `etCustom`, then the `OnGetCustomEvent` (111), `OnGetCustomEventID` (111) and `OnGetCustomCategory` (111).

The other logging calls: `Info` (108), `Warning` (107), `Error` (107) and `Debug` (108) use the `Log` call to do the actual work.

See also: `TEventLog.Info` (108), `TEventLog.Warning` (107), `TEventLog.Error` (107), `TEventLog.Debug` (108), `TEventLog.OnGetCustomEvent` (111), `TEventLog.OnGetCustomEventID` (111), `TEventLog.OnGetCustomCategory` (111)

7.5.8 TEventLog.Warning

Synopsis: Log a warning message.

Declaration: `procedure Warning(Msg: String); Overload`
`procedure Warning(Fmt: String;Args: Array of const); Overload`

Visibility: public

Description: `Warning` is a utility function which logs a message with the `etWarning` type. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters.

See also: `TEventLog.Log` (107), `TEventLog.Info` (108), `TEventLog.Error` (107), `TEventLog.Debug` (108)

7.5.9 TEventLog.Error

Synopsis: Log an error message to

Declaration: `procedure Error(Msg: String); Overload`
`procedure Error(Fmt: String;Args: Array of const); Overload`

Visibility: public

Description: `Error` is a utility function which logs a message with the `etError` type. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters.

See also: `TEventLog.Log` (107), `TEventLog.Info` (108), `TEventLog.Warning` (107), `TEventLog.Debug` (108)

7.5.10 TEventLog.Debug

Synopsis: Log a debug message

Declaration: `procedure Debug(Msg: String); Overload`
`procedure Debug(Fmt: String; Args: Array of const); Overload`

Visibility: public

Description: `Debug` is a utility function which logs a message with the `etDebug` type. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters.

See also: `TEventLog.Log` (107), `TEventLog.Info` (108), `TEventLog.Warning` (107), `TEventLog.Error` (107)

7.5.11 TEventLog.Info

Synopsis: Log an informational message

Declaration: `procedure Info(Msg: String); Overload`
`procedure Info(Fmt: String; Args: Array of const); Overload`

Visibility: public

Description: `Info` is a utility function which logs a message with the `etInfo` type. The message is either the parameter `Msg` as is, or is formatted from the `Fmt` and `Args` parameters.

See also: `TEventLog.Log` (107), `TEventLog.Warning` (107), `TEventLog.Error` (107), `TEventLog.Debug` (108)

7.5.12 TEventLog.Identification

Synopsis: Identification string for messages

Declaration: `Property Identification : String`

Visibility: published

Access: Read, Write

Description: `Identification` is used as a string identifying the source of the messages in the system log. If it is empty, the filename part of the application binary is used.

See also: `TEventLog.Active` (109), `TEventLog.TimeStampFormat` (110)

7.5.13 TEventLog.LogType

Synopsis: Log type

Declaration: `Property LogType : TLogType`

Visibility: published

Access: Read, Write

Description: `LogType` is the type of the log: if it is `ltSystem`, then the system log is used, if it is available. If it is `ltFile` or there is no system log available, then the log messages are written to a file. The name for the log file is taken from the `FileName` (109) property.

See also: `TEventLog.FileName` (109)

7.5.14 TEventLog.Active

Synopsis: Activate the log mechanism

Declaration: `Property Active : Boolean`

Visibility: published

Access: Read,Write

Description: `Active` determines whether the log mechanism is active: if set to `True`, the component connects to the system log mechanism, or opens the log file if needed. Any attempt to log a message while the log is not active will try to set this property to `True`. Disconnecting from the system log or closing the log file is done by setting the `Active` property to `False`.

If the connection to the system logger fails, or the log file cannot be opened, then setting this property may result in an exception.

See also: `TEventLog.Log` (107)

7.5.15 TEventLog.DefaultEventType

Synopsis: Default event type for the `Log` (107) call.

Declaration: `Property DefaultEventType : TEventType`

Visibility: published

Access: Read,Write

Description: `DefaultEventType` is the event type used by the `Log` (107) call if it's `EventType` parameter is omitted.

See also: `TEventLog.Log` (107)

7.5.16 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` (104) is `ltFile`. If none is specified, then the name of the application binary is used, with the extension replaced by `.log`. The file is then located in the `/tmp` directory on unix-like systems, or in the application directory for Dos/Windows like systems.

See also: `TEventType.LogType` (104)

7.5.17 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` ([108](#))

7.5.18 TEventLog.CustomLogType

Synopsis: Custom log type ID

Declaration: `Property CustomLogType : Word`

Visibility: published

Access: Read,Write

Description: `CustomLogType` is used in the `EventTypeToString` ([106](#)) to format the custom log event type string.

See also: `TEventLog.EventTypeToString` ([106](#))

7.5.19 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` ([106](#)) 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:

1000Custom event types

1001Information event type

1002Warning event type

1003Error event type

1004Debug event type

See also: `TEventLog.RegisterMessageFile` ([106](#))

7.5.20 TEventLog.OnGetCustomCategory

Synopsis: Event to retrieve custom message category

Declaration: Property OnGetCustomCategory : TLogCategoryEvent

Visibility: published

Access: Read,Write

Description: OnGetCustomCategory is called on the windows platform to determine the category of a custom event type. It should return an ID which will be used by windows to look up the string which describes the message category in the file containing the resource strings.

See also: TEventLog.OnGetCustomEventID ([111](#)), TEventLog.OnGetCustomEvent ([111](#))

7.5.21 TEventLog.OnGetCustomEventID

Synopsis: Event to retrieve custom event ID

Declaration: Property OnGetCustomEventID : TLogCodeEvent

Visibility: published

Access: Read,Write

Description: OnGetCustomEventID is called on the windows platform to determine the category of a custom event type. It should return an ID which will be used by windows to look up the string which formats the message, in the file containing the resource strings.

See also: TEventLog.OnGetCustomCategory ([111](#)), TEventLog.OnGetCustomEvent ([111](#))

7.5.22 TEventLog.OnGetCustomEvent

Synopsis: Event to retrieve custom event Code

Declaration: Property OnGetCustomEvent : TLogCodeEvent

Visibility: published

Access: Read,Write

Description: OnGetCustomEvent is called on the windows platform to determine the event code of a custom event type. It should return an ID.

See also: TEventLog.OnGetCustomCategory ([111](#)), TEventLog.OnGetCustomEventID ([111](#))

Chapter 8

Reference for unit 'ezcgi'

8.1 Used units

Table 8.1: Used units by unit 'ezcgi'

Name	Page
Classes	??
strings	112
sysutils	??

8.2 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` ([115](#)) or `DoGet` ([115](#)) methods should be overridden.

8.3 Constants, types and variables

8.3.1 Constants

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

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

8.4 ECGIException

8.4.1 Description

Exception raised by `TEZcgi` ([113](#))

8.5 TEZcgi

8.5.1 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` (115) or `DoGet` (115) methods. To run the program, an instance of this class must be created, and it's `Run` (114) 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.

8.5.2 Method overview

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

8.5.3 Property overview

Page	Property	Access	Description
117	Email	rw	Email of the server administrator
117	Name	rw	Name of the server administrator
116	Names	r	Indexed array with available variable names.
115	Values	r	Variables passed to the CGI script
117	VariableCount	r	Number of available variables.
116	Variables	r	Indexed array with variables as name=value pairs.

8.5.4 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` (112) property.

See also: `TZECGI.Variables` (112), `TZECGI.Names` (112), `TZECGI.Values` (112)

8.5.5 TEZcgi.Destroy

Synopsis: Removes the `TEZCGI` component from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` removes all variables from memory and then calls the inherited `destroy`, removing the `TEZCGI` instance from memory.

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

See also: `TEZcgi.Create` (113)

8.5.6 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` (112) property) and calls the `DoPost` (115) or `DoGet` (115) methods, depending on the method passed to the web server.

After creating the instance of `TEZCGI`, the `Run` method is the only method that should be called when using this component.

See also: `TEZCGI.Variables` (112), `TEZCGI.DoPost` (115), `TEZCGI.DoGet` (115)

8.5.7 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` (115) or `DoGet` (115) methods.

See also: `TEZCGI.DoPost` (115), `TEZCGI.DoGet` (115), `TEZcgi.PutLine` (114)

8.5.8 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` (114) was called with a content type of `text`. The sent text is not processed in any way, i.e. no HTML entities or so are inserted instead of special HTML characters. This should be done by the user.

Errors: No check is performed whether the content type is right.

See also: `TEZcgi.WriteContent` (114)

8.5.9 TEZcgi.GetValue

Synopsis: Return the value of a request variable.

Declaration: `function GetValue(Index: String; defaultValue: String) : String`

Visibility: public

Description: `GetValue` returns the value of the variable named `Index`, and returns `DefaultValue` if it is empty or does not exist.

See also: `TEZCGI.Values` ([115](#))

8.5.10 TEZcgi.DoPost

Synopsis: Method to handle POST requests

Declaration: `procedure DoPost; Virtual`

Visibility: public

Description: `DoPost` is called by the `Run` ([114](#)) method the POST method was used to invoke the CGI application. It should be overridden in descendents of `TEZcgi` to actually handle the request.

See also: `TEZcgi.Run` ([114](#)), `TEZcgi.DoGet` ([115](#))

8.5.11 TEZcgi.DoGet

Synopsis: Method to handle GET requests

Declaration: `procedure DoGet; Virtual`

Visibility: public

Description: `DoGet` is called by the `Run` ([114](#)) method the GET method was used to invoke the CGI application. It should be overridden in descendents of `TEZcgi` to actually handle the request.

See also: `TEZcgi.Run` ([114](#)), `TEZcgi.DoPost` ([115](#))

8.5.12 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_TYPEAuthorization type

CONTENT_LENGTHContent length

CONTENT_TYPEContent type

GATEWAY_INTERFACEUsed gateway interface
PATH_INFORequested URL
PATH_TRANSLATEDTransformed URL
QUERY_STRINGClient query string
REMOTE_ADDRAddress of remote client
REMOTE_HOSTDNS name of remote client
REMOTE_IDENTRemote identity.
REMOTE_USERRemote user
REQUEST_METHODRequest methods (POST or GET)
SCRIPT_NAMEScript name
SERVER_NAMEServer host name
SERVER_PORTServer port
SERVER_PROTOCOLServer protocol
SERVER_SOFTWAREWeb server software
HTTP_ACCEPTAccepted responses
HTTP_ACCEPT_CHARSETAccepted character sets
HTTP_ACCEPT_ENCODINGAccepted encodings
HTTP_IF_MODIFIED_SINCEProxy information
HTTP_REFERERReferring page
HTTP_USER_AGENTClient 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 \(116\)](#), [TEZCGI.Names \(116\)](#), [TEZCGI.GetValue \(115\)](#), [TEZcgi.VariableCount \(117\)](#)

8.5.13 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` ([117](#)). Any other value will result in an exception being raised.

See also: [TEZcgi.Variables \(116\)](#), [TEZcgi.Values \(115\)](#), [TEZcgi.GetValue \(115\)](#), [TEZcgi.VariableCount \(117\)](#)

8.5.14 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` (117). Any other value will result in an exception being raised.

See also: `TEZcgi.Names` (116), `TEZcgi.Values` (115), `TEZcgi.GetValue` (115), `TEZcgi.VariableCount` (117)

8.5.15 `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` (116) and `Variables` (116) properties.

See also: `TEZcgi.Names` (116), `TEZcgi.Variables` (116), `TEZcgi.Values` (115), `TEZcgi.GetValue` (115)

8.5.16 `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` (114) method.

See also: `TEZcgi.Run` (114), `TEZcgi.Email` (117)

8.5.17 `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` (114) method.

See also: `TEZcgi.Run` (114), `TEZcgi.Name` (117)

Chapter 9

Reference for unit 'gettext'

9.1 Used units

Table 9.1: Used units by unit 'gettext'

Name	Page
Classes	??
sysutils	??

9.2 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` ([120](#))) 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.

9.3 Constants, types and variables

9.3.1 Constants

```
MOFileHeaderMagic = $950412de
```

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

9.3.2 Types

```
PLongWordArray = ^TLongWordArray
```

Pointer to a `TLongWordArray` ([119](#)) array.

```
PMOStringTable = ^TMOStringTable
```

Pointer to a TMOStringTable (119) array.

```
PPCharArray = ^TPCharArray
```

Pointer to a TPCharArray (119) array.

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

TLongWordArray is an array used to define the PLongWordArray (118) pointer. A variable of type TLongWordArray should never be directly declared, as it would occupy too much memory. The PLongWordArray type can be used to allocate a dynamic number of elements.

```
TMOFileHeader = packed record
  magic : LongWord;
  revision : LongWord;
  nstrings : LongWord;
  OrigTabOffset : LongWord;
  TransTabOffset : LongWord;
  HashTabSize : LongWord;
  HashTabOffset : LongWord;
end
```

This structure describes the structure of a .mo file with string localizations.

```
TMOStringInfo = packed record
  length : LongWord;
  offset : LongWord;
end
```

This record is one element in the string tables describing the original and translated strings. It describes the position and length of the string. The location of these tables is stored in the TMOFileHeader (119) record at the start of the file.

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

TMOStringTable is an array type containing TMOStringInfo (119) records. It should never be used directly, as it would occupy too much memory.

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

TLongWordArray is an array used to define the PPCharArray (119) 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.

9.4 Procedures and functions

9.4.1 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.

9.4.2 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` (120) 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` (120), `TMOFile` (120)

9.4.3 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` (120), 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` (120), `TMOFile` (120)

9.5 EMOFileError

9.5.1 Description

`EMOFileError` is raised in case an `TMOFile` (120) instance is created with an invalid `.mo`.

9.6 TMOFile

9.6.1 Description

`TMOFile` is a class providing easy access to a `.mo` file. It can be used to translate any of the strings that reside in the `.mo` file. The internal structure of the `.mo` is completely hidden.

9.6.2 Method overview

Page	Property	Description
121	Create	Create a new instance of the <code>TMOFile</code> class.
121	Destroy	Removes the <code>TMOFile</code> instance from memory
121	Translate	Translate a string

9.6.3 TMOFile.Create

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

Declaration: `constructor Create(const AFilename: String)`
`constructor Create(AStream: TStream)`

Visibility: `public`

Description: `Create` creates a new instance of the `MOFile` class. It opens the file `AFilename` or the stream `AStream`. If a stream is provided, it should be seekable.

The whole contents of the file is read into memory during the `Create` call. This means that the stream is no longer needed after the `Create` call.

Errors: If the named file does not exist, then an exception may be raised. If the file does not contain a valid `TMOFileHeader` ([119](#)) structure, then an `EMOFileError` ([120](#)) exception is raised.

See also: `TMOFile.Destroy` ([121](#))

9.6.4 TMOFile.Destroy

Synopsis: Removes the `TMOFile` instance from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` cleans the internal structures with the contents of the `.mo`. After this the `TMOFile` instance is removed from memory.

See also: `TMOFile.Create` ([121](#))

9.6.5 TMOFile.Translate

Synopsis: Translate a string

Declaration: `function Translate(AOrig: PChar; ALen: Integer; AHash: LongWord) : String`
`function Translate(AOrig: String; AHash: LongWord) : String`
`function Translate(AOrig: String) : String`

Visibility: `public`

Description: `Translate` translates the string `AOrig`. The string should be in the `.mo` file as-is. The string can be given as a plain string, as a `PChar` (with length `ALen`). If the hash value (`AHash`) of the string is not given, it is calculated.

If the string is in the `.mo` file, the translated string is returned. If the string is not in the file, an empty string is returned.

Errors: None.

Chapter 10

Reference for unit 'idea'

10.1 Used units

Table 10.1: Used units by unit 'idea'

Name	Page
Classes	??
sysutils	??

10.2 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.

10.3 Constants, types and variables

10.3.1 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

10.3.2 Types

`IdeaCryptData = TIdeaCryptData`

Provided for backward functionality.

`IdeaCryptKey = TIdeaCryptKey`

Provided for backward functionality.

`IDEAkey = TIDEAKey`

Provided for backward functionality.

`TIdeaCryptData = Array[0..3] of Word`

`TIdeaCryptData` is an internal type, defined to hold data for encryption/decryption.

`TIdeaCryptKey = Array[0..7] of Word`

The actual encryption or decryption key for IDEA is 64-bit long. This type is used to hold such a key. It can be generated with the `EnKeyIDEA` (124) or `DeKeyIDEA` (123) algorithms depending on whether an encryption or decryption key is needed.

`TIDEAKey = Array[0..keylen-1] of Word`

The IDEA key should be filled by the user with some random data (say, a passphrase). This key is used to generate the actual encryption/decryption keys.

10.4 Procedures and functions

10.4.1 CipherIdea

Synopsis: Encrypt or decrypt a buffer.

Declaration: `procedure CipherIdea(Input: TIdeaCryptData; var outdata: TIdeaCryptData;
z: TIDEAKey)`

Visibility: default

Description: `CipherIdea` encrypts or decrypts a buffer with data (`Input`) using key `z`. The resulting encrypted or decrypted data is returned in `Output`.

Errors: None.

See also: `EnKeyIdea` (124), `DeKeyIdea` (123), `TIDEAEncryptStream` (125), `TIDEADecryptStream` (124)

10.4.2 DeKeyIdea

Synopsis: Create a decryption key from an encryption key.

Declaration: `procedure DeKeyIdea(z: TIDEAKey; var dk: TIDEAKey)`

Visibility: default

Description: `DeKeyIdea` creates a decryption key based on the encryption key `z`. The decryption key is returned in `dk`. Note that only a decryption key generated from the encryption key that was used to encrypt the data can be used to decrypt the data.

Errors: None.

See also: `EnKeyIdea` ([124](#)), `CipherIdea` ([123](#))

10.4.3 EnKeyIdea

Synopsis: Create an IDEA encryption key from a user key.

Declaration: `procedure EnKeyIdea (UserKey: TIDEACryptKey; var z: TIDEAKey)`

Visibility: default

Description: `EnKeyIdea` creates an IDEA encryption key from user-supplied data in `UserKey`. The Encryption key is stored in `z`.

Errors: None.

See also: `DeKeyIdea` ([123](#)), `CipherIdea` ([123](#))

10.5 EIDEAError

10.5.1 Description

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

10.6 TIDEADeCryptStream

10.6.1 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` ([123](#)) 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` ([125](#)) encryption stream.

10.6.2 Method overview

Page	Property	Description
124	Read	Reads data from the stream, decrypting it as needed
125	Seek	Set position on the stream
125	Write	Write data to the stream

10.6.3 TIDEADeCryptStream.Read

Synopsis: Reads data from the stream, decrypting it as needed

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

Visibility: public

Description: Read attempts to read `Count` bytes from the stream, placing them in `Buffer` the bytes are read from the source stream and decrypted as they are read. (bytes are read from the source stream in blocks of 8 bytes. The function returns the number of bytes actually read.

Errors: If an error occurs when reading data from the source stream, an exception may be raised.

See also: `TIDEADecryptStream.Write` (125), `TIDEADecryptStream.Seek` (125), `TIDEAEncryptStream` (125)

10.6.4 TIDEADeCryptStream.Write

Synopsis: Write data to the stream

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

Visibility: public

Description: `Write` always raises an `EIDEAError` (124) exception, because the decryption stream is read-only. To write to an encryption stream, use the `Write` (126) method of the `TIDEAEncryptStream` (125) encryption stream.

Errors: An `EIDEAError` (124) exception is raised when calling this method.

See also: `TIDEADecryptStream.Read` (124), `TIDEAEncryptStream` (125), `TIDEAEncryptStream.Write` (126)

10.6.5 TIDEADeCryptStream.Seek

Synopsis: Set position on the stream

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

Visibility: public

Description: `Seek` will only work on a forward seek. It emulates a forward seek by reading and discarding bytes from the input stream. The `TIDEADeCryptStream` stream tries to provide seek capabilities for the following limited number of cases:

Origin=soFromBeginningIf `Offset` is larger than the current position, then the remaining bytes are skipped by reading them from the stream and discarding them.

Origin=soFromCurrentIf `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` (124) exception is raised if the stream does not allow the requested seek operation.

See also: `TIDEADeCryptStream.Read` (124)

10.7 TIDEAEncryptStream

10.7.1 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` (123) 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` (124) decryption stream.

10.7.2 Method overview

Page	Property	Description
126	Destroy	Flush data buffers and free the stream instance.
127	Flush	Write remaining bytes from the stream
126	Read	Read data from the stream
127	Seek	Set stream position
126	Write	Write bytes to the stream to be encrypted

10.7.3 TIDEAEncryptStream.Destroy

Synopsis: Flush data buffers and free the stream instance.

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` flushes any data still remaining in the internal encryption buffer, and then calls the inherited `Destroy`

By default, the destination stream is not freed when the encryption stream is freed.

Errors: None.

See also: `TIDEAStream.Create` ([128](#))

10.7.4 TIDEAEncryptStream.Read

Synopsis: Read data from the stream

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

Visibility: `public`

Description: `Read` always raises an `EIDEAError` ([124](#)) exception, because the encryption stream is write-only. To read from an encrypted stream, use the `Read` ([124](#)) method of the `TIDEADecryptStream` ([124](#)) decryption stream.

Errors: An `EIDEAError` ([124](#)) exception is raised when calling this method.

See also: `TIDEAEncryptStream.Write` ([126](#)), `TIDEADecryptStream` ([124](#)), `TIDEADecryptStream.Read` ([124](#))

10.7.5 TIDEAEncryptStream.Write

Synopsis: Write bytes to the stream to be encrypted

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

Visibility: `public`

Description: `Write` writes `Count` bytes from `Buffer` to the stream, encrypting the bytes as they are written (encryption in blocks of 8 bytes).

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

See also: `TIDEADecryptStream.Read` ([124](#))

10.7.6 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` (124) will be raised unless called with 0 and `soFromCurrent` as arguments.

See also: `TIDEAEncryptStream.Write` (126), `EIDEAError` (124)

10.7.7 TIDEAEncryptStream.Flush

Synopsis: Write remaining bytes from the stream

Declaration: `procedure Flush`

Visibility: public

Description: `Flush` writes the current encryption buffer to the stream. Encryption always happens in blocks of 8 bytes, so if the buffer is not completely filled at the end of the writing operations, it must be flushed. It should never be called directly, unless at the end of all writing operations. It is called automatically when the stream is destroyed.

Errors: None.

See also: `TIDEAEncryptStream.Write` (126)

10.8 TIDEAStream

10.8.1 Description

Do not create instances of `TIDEAStream` directly. It implements no useful functionality: it serves as a common ancestor of the `TIDEAEncryptStream` (125) and `TIDEADeCryptStream` (124), 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.

10.8.2 Method overview

Page	Property	Description
128	Create	Creates a new instance of the <code>TIDEAStream</code> class

10.8.3 Property overview

Page	Property	Access	Description
128	Key	r	Key used when encrypting/decrypting

10.8.4 TIDEAStream.Create

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

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

Visibility: `public`

Description: `Create` stores the encryption/decryption key and then calls the inherited `Create` to store the `Dest` stream.

Errors: None.

See also: `TIDEAEncryptStream` ([125](#)), `TIDEADeCryptStream` ([124](#))

10.8.5 TIDEAStream.Key

Synopsis: Key used when encrypting/decrypting

Declaration: `Property Key : TIDEAKey`

Visibility: `public`

Access: `Read`

Description: `Key` is the key as it was passed to the constructor of the stream. It cannot be changed while data is read or written. It is the key as it is used when encrypting/decrypting.

See also: `CipherIdea` ([123](#))

Chapter 11

Reference for unit 'inicol'

11.1 Used units

Table 11.1: Used units by unit 'inicol'

Name	Page
Classes	??
Inifiles	129
sysutils	??

11.2 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` (??) class for this.

11.3 Constants, types and variables

11.3.1 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.

11.4 EIniCol

11.4.1 Description

EIniCol is used to report error conditions in the load and save methods of TIniCollection ([130](#)).

11.5 TIniCollection

11.5.1 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 ([133](#)).

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.

11.5.2 Method overview

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

11.5.3 Property overview

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

11.5.4 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 ([133](#)) property. It calls the LoadFromFile ([132](#)) 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 ([132](#)), TIniCollection.LoadFromIni ([132](#)), TIniCollection.Save ([131](#)), TIniCollection.FileName ([133](#))

11.5.5 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` (133) property, using `GlobalSection` (133) as the section. It calls the `SaveToFile` (131) 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` (131), `TIniCollection.SaveToIni` (131), `TIniCollection.Load` (130), `TIniCollection.FileName` (133)

11.5.6 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` (132) property for the key. After this it calls the `SaveToIni` (134) method of all `TIniCollectionItem` (133) 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` (133) property contains the value of `Section`, it will be used in the `Save` (133) method.

See also: `TIniCollectionItem.SaveToIni` (134)

11.5.7 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` (131) 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` (131), `TIniCollection.LoadFromFile` (132)

11.5.8 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` (134) 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` (131) method description.

See also: `TIniCollection.LoadFromFile` (132), `TIniCollectionItem.LoadFromIni` (134), `TIniCollection.SaveToIni` (131)

11.5.9 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` (132) using this instance and `Section` as the parameters.

See also: `TIniCollection.LoadFromIni` (132), `TIniCollection.Load` (130), `TIniCollection.SaveToIni` (131), `TIniCollection.SaveToFile` (131)

11.5.10 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` (133), `TIniCollection.GlobalSection` (133)

11.5.11 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` (135) property to return the exact section name. It can be empty.

See also: `TIniCollection.Section` (130), `TIniCollection.GlobalSection` (133)

11.5.12 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` (132) or `SaveToFile` (131) operation. It is used in the `Load` (130) or `Save` (131) calls.

See also: `TIniCollection.Save` (131), `TIniCollection.LoadFromFile` (132), `TIniCollection.SaveToFile` (131), `TIniCollection.Load` (130)

11.5.13 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` (132) or `SaveToIni` (131) calls. It's used in the `Load` (130) or `Save` (131) calls.

See also: `TIniCollection.Save` (131), `TIniCollection.LoadFromFile` (132), `TIniCollection.SaveToFile` (131), `TIniCollection.Load` (130)

11.6 TIniCollectionItem

11.6.1 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` (134) and `LoadFromIni` (134) methods should be overridden. They should implement the actual loading and saving. The loading and saving is always initiated by the methods in `TIniCollection` (130), `TIniCollection.LoadFromIni` (132) and `TIniCollection.SaveToIni` (131) respectively.

11.6.2 Method overview

Page	Property	Description
135	LoadFromFile	Load item from a file
134	LoadFromIni	Method called when the item must be loaded
134	SaveToFile	Save item to a file
134	SaveToIni	Method called when the item must be saved

11.6.3 Property overview

Page	Property	Access	Description
135	SectionName	rw	Default section name

11.6.4 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` ([131](#)) 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` ([131](#)), `TIniCollectionItem.LoadFromIni` ([134](#)), `TIniCollectionItem.SaveToFile` ([134](#)), `TIniCollectionItem.LoadFromFile` ([135](#))

11.6.5 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` ([132](#)) 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` ([132](#)), `TIniCollectionItem.SaveToIni` ([134](#)), `TIniCollectionItem.LoadFromFile` ([135](#)), `TIniCollectionItem.SaveToFile` ([134](#))

11.6.6 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` (134) to save the item to the indicated file in .ini format under the section `Section`

Errors: An exception can occur if the file is not writeable.

See also: `TIniCollectionItem.SaveToIni` (134), `TIniCollectionItem.LoadFromFile` (135)

11.6.7 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` (134) to load the item from the indicated file in .ini format from the section `Section`.

Errors: None.

See also: `TIniCollectionItem.SaveToFile` (134), `TIniCollectionItem.LoadFromIni` (134)

11.6.8 TIniCollectionItem.SectionName

Synopsis: Default section name

Declaration: `Property SectionName : String`

Visibility: public

Access: Read,Write

Description: `SectionName` is the section name under which the item will be saved or from which it should be read. The read/write functions should be overridden in descendents to determine a unique section name within the .ini file.

See also: `TIniCollectionItem.SaveToFile` (134), `TIniCollectionItem.LoadFromIni` (134)

11.7 TNamedIniCollection

11.7.1 Method overview

Page	Property	Description
136	<code>FindByName</code>	
136	<code>FindByUserData</code>	
136	<code>IndexOfName</code>	
136	<code>IndexOfUserData</code>	

11.7.2 Property overview

Page	Property	Access	Description
136	<code>NamedItems</code>	rw	

11.7.3 TNamedIniCollection.IndexOfUserData

Declaration: `function IndexOfUserData(UserData: TObject) : Integer`

Visibility: public

11.7.4 TNamedIniCollection.IndexOfName

Declaration: `function IndexOfName(const AName: String) : Integer`

Visibility: public

11.7.5 TNamedIniCollection.FindByName

Declaration: `function FindByName(const AName: String) : TNamedIniCollectionItem`

Visibility: public

11.7.6 TNamedIniCollection.FindByUserData

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

Visibility: public

11.7.7 TNamedIniCollection.NamedItems

Declaration: `Property NamedItems[Index: Integer]: TNamedIniCollectionItem; default`

Visibility: public

Access: Read,Write

11.8 TNamedIniCollectionItem

11.8.1 Property overview

Page	Property	Access	Description
136	Name	rw	
136	UserData	rw	

11.8.2 TNamedIniCollectionItem.UserData

Declaration: `Property UserData : TObject`

Visibility: public

Access: Read,Write

11.8.3 TNamedIniCollectionItem.Name

Declaration: `Property Name : String`

Visibility: published

Access: Read,Write

Chapter 12

Reference for unit 'iostream'

12.1 Used units

Table 12.1: Used units by unit 'iostream'

Name	Page
Classes	??

12.2 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`).

12.3 Constants, types and variables

12.3.1 Types

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

Table 12.2: 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` (138) constructor of `TIOStream` (138), it determines what kind of stream is created.

12.4 EIOStreamError

12.4.1 Description

Error thrown in case of an invalid operation on a TIOStream ([138](#)).

12.5 TIOStream

12.5.1 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 ([137](#)) 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 ([139](#)) behaviour based on this position.

12.5.2 Method overview

Page	Property	Description
138	Create	Construct a new instance of TIOStream (138)
138	Read	Read data from the stream.
139	Seek	Set the stream position
139	SetSize	Set the size of the stream
139	Write	Write data to the stream

12.5.3 TIOStream.Create

Synopsis: Construct a new instance of TIOStream ([138](#))

Declaration: `constructor Create(aIOSType: TIOSType)`

Visibility: public

Description: Create creates a new instance of TIOStream ([138](#)), 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 ([138](#)), TIOStream.Write ([139](#))

12.5.4 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 ([138](#)) 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 ([137](#)), TIOStream.Write ([139](#))

12.5.5 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` (138) exception. If the stream can be written to, it calls the inherited `Write` to actually read the data.

Errors: An `EIOStreamError` exception is raised if the stream does not allow writing.

See also: `TIOStreamType` (137), `TIOStream.Read` (138)

12.5.6 TIOStream.SetSize

Synopsis: Set the size of the stream

Declaration: `procedure SetSize(NewSize: LongInt); Override`

Visibility: public

Description: `SetSize` overrides the standard `SetSize` implementation. It always raises an exception, because the standard input, output and stderr files have no size.

Errors: An `EIOStreamError` exception is raised when this method is called.

See also: `EIOStreamError` (138)

12.5.7 TIOStream.Seek

Synopsis: Set the stream position

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

Visibility: public

Description: `Seek` overrides the standard `Seek` implementation. Normally, standard input, output and stderr are not seekable. The `TIOStream` stream tries to provide seek capabilities for the following limited number of cases:

Origin=soFromBeginning If `Offset` is larger than the current position, then the remaining bytes are skipped by reading them from the stream and discarding them, if the stream is of type `iosInput`.

Origin=soFromCurrent If `Offset` is zero, the current position is returned. If it is positive, then `Offset` bytes are skipped by reading them from the stream and discarding them, if the stream is of type `iosInput`.

All other cases will result in a `EIOStreamError` exception.

Errors: An `EIOStreamError` (138) exception is raised if the stream does not allow the requested seek operation.

See also: `EIOStreamError` (138)

Chapter 13

Reference for unit 'Pipes'

13.1 Used units

Table 13.1: Used units by unit 'Pipes'

Name	Page
Classes	??
sysutils	??

13.2 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.

13.3 Constants, types and variables

13.3.1 Constants

`ENoReadMsg = 'Cannot read from OutputPipeStream.'`

Constant used in `ENoReadPipe` (141) exception.

`ENoSeekMsg = 'Cannot seek on pipes'`

Constant used in `EPipeSeek` (142) exception.

`ENoWriteMsg = 'Cannot write to InputPipeStream.'`

Constant used in `ENoWritePipe` (141) exception.

`EPipeMsg = 'Failed to create pipe.'`

Constant used in `EPipeCreation` (141) exception.

13.4 Procedures and functions

13.4.1 CreatePipeHandles

Synopsis: Function to create a set of pipe handles

Declaration: `function CreatePipeHandles (var InHandle: THandle; var OutHandle: THandle)
: Boolean`

Visibility: default

Description: `CreatePipeHandles` provides an OS-independent way to create a set of pipe filehandles. These handles are inheritable to child processes. The reading end of the pipe is returned in `InHandle`, the writing end in `OutHandle`.

Errors: On error, `False` is returned.

See also: `CreatePipeStreams` ([141](#))

13.4.2 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` ([141](#)), and if that call is successful, a pair of streams is created: `InPipe` and `OutPipe`.

Errors: If no pipe handles could be created, an `EPipeCreation` ([141](#)) exception is raised.

See also: `CreatePipeHandles` ([141](#)), `TInputPipeStream` ([142](#)), `TOutputPipeStream` ([143](#))

13.5 ENoReadPipe

13.5.1 Description

Exception raised when a write operation is attempted on a write-only pipe.

13.6 ENoWritePipe

13.6.1 Description

Exception raised when a read operation is attempted on a read-only pipe.

13.7 EPipeCreation

13.7.1 Description

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

13.8 EPipeError

13.8.1 Description

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

13.9 EPipeSeek

13.9.1 Description

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

13.10 TInputPipeStream

13.10.1 Description

TInputPipeStream is created by the CreatePipeStreams (141) 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.

13.10.2 Method overview

Page	Property	Description
143	Read	Read data from the stream to a buffer.
142	Seek	Set the current position of the stream
142	Write	Write data to the stream.

13.10.3 Property overview

Page	Property	Access	Description
143	NumBytesAvailable	r	Number of bytes available for reading.

13.10.4 TInputPipeStream.Write

Synopsis: Write data to the stream.

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

Visibility: public

Description: Write overrides the parent implementation of Write. On a TInputPipeStream will always raise an exception, as the pipe is read-only.

Errors: An ENoWritePipe (141) exception is raised when this function is called.

See also: TInputPipeStream.Read (143), TInputPipeStream.Seek (142)

13.10.5 TInputPipeStream.Seek

Synopsis: Set the current position of the stream

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

Visibility: public

Description: `Seek` overrides the standard `Seek` implementation. Normally, pipe streams stderr are not seekable. The `TInputPipeStream` stream tries to provide seek capabilities for the following limited number of cases:

Origin=soFromBeginning If `Offset` is larger than the current position, then the remaining bytes are skipped by reading them from the stream and discarding them.

Origin=soFromCurrent If `Offset` is zero, the current position is returned. If it is positive, then `Offset` bytes are skipped by reading them from the stream and discarding them, if the stream is of type `iosInput`.

All other cases will result in a `EPipeSeek` exception.

Errors: An `EPipeSeek` (142) exception is raised if the stream does not allow the requested seek operation.

See also: `EPipeSeek` (142), `#rtl.classes.tstream.seek` (??)

13.10.6 TInputPipeStream.Read

Synopsis: Read data from the stream to a buffer.

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

Visibility: public

Description: `Read` calls the inherited `read` and adjusts the internal position pointer of the stream.

Errors: None.

See also: `TInputPipeStream.Write` (142), `TInputPipeStream.Seek` (142)

13.10.7 TInputPipeStream.NumBytesAvailable

Synopsis: Number of bytes available for reading.

Declaration: `Property NumBytesAvailable : DWord`

Visibility: public

Access: Read

Description: `NumBytesAvailable` is the number of bytes available for reading. This is the number of bytes in the OS buffer for the pipe. It is not a number of bytes in an internal buffer.

If this number is nonzero, then reading `NumBytesAvailable` bytes from the stream will not block the process. Reading more than `NumBytesAvailable` bytes will block the process, while it waits for the requested number of bytes to become available.

See also: `TInputPipeStream.Read` (143)

13.11 TOutputPipeStream

13.11.1 Description

`TOutputPipeStream` is created by the `CreatePipeStreams` (141) call to represent the writing end of a pipe. It is a `TStream` (??) descendent which does not allow reading.

13.11.2 Method overview

Page	Property	Description
144	Read	Read data from the stream.
144	Seek	Sets the position in the stream

13.11.3 TOutputPipeStream.Seek

Synopsis: Sets the position in the stream

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

Visibility: public

Description: `Seek` is overridden in `TOutputPipeStream`. Calling this method will always raise an exception: an output pipe is not seekable.

Errors: An `EPipeSeek` ([142](#)) exception is raised if this method is called.

13.11.4 TOutputPipeStream.Read

Synopsis: Read data from the stream.

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

Visibility: public

Description: `Read` overrides the parent `Read` implementation. It always raises an exception, because a output pipe is write-only.

Errors: An `ENoReadPipe` ([141](#)) exception is raised when this function is called.

See also: `TOutputPipeStream.Seek` ([144](#))

Chapter 14

Reference for unit 'pooledmm'

14.1 Used units

Table 14.1: Used units by unit 'pooledmm'

Name	Page
Classes	??

14.2 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.

14.3 Constants, types and variables

14.3.1 Types

```
PPooledMemManagerItem = ^TPooledMemManagerItem
```

`PPooledMemManagerItem` is a pointer type, pointing to a `TPooledMemManagerItem` (146) item, used in a linked list.

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

`TEnumItemsMethod` is a prototype for the callback used in the `TNonFreePooledMemManager.EnumerateItems` (147) call. The parameter `Item` will be set to each of the pointers in the item list of `TNonFreePooledMemManager` (146).

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

`TPooledMemManagerItem` is used internally by the `TPooledMemManager` (148) class to maintain the free list block. It simply points to the next free block.

14.4 TNonFreePooledMemManager

14.4.1 Description

`TNonFreePooledMemManager` keeps a list of fixed-size memory blocks in memory. Each block has the same size, making it suitable for storing a lot of records of the same type. It does not free the items stored in it, except when the list is cleared as a whole.

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

14.4.2 Method overview

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

14.4.3 Property overview

Page	Property	Access	Description
147	ItemSize	r	Size of an item in the list

14.4.4 TNonFreePooledMemManager.Clear

Synopsis: Clears the memory

Declaration: `procedure Clear`

Visibility: `public`

Description: `Clear` clears all blocks from memory, freeing the allocated memory blocks. None of the pointers returned by `NewItem` (147) is valid after a call to `Clear`

See also: `TNonFreePooledMemManager.NewItem` (147)

14.4.5 TNonFreePooledMemManager.Create

Synopsis: Creates a new instance of `TNonFreePooledMemManager`

Declaration: `constructor Create(TheItemSize: Integer)`

Visibility: `public`

Description: `Create` creates a new instance of `TNonFreePooledMemManager` and sets the item size to `TheItemSize`.

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

See also: `TNonFreePooledMemManager.ItemSize` (147)

14.4.6 TNonFreePooledMemManager.Destroy

Synopsis: Removes the `TNonFreePooledMemManager` instance from memory

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` clears the list, clears the internal structures, and then calls the inherited `Destroy`.

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

See also: `TNonFreePooledMemManager.Create` (146), `TNonFreePooledMemManager.Clear` (146)

14.4.7 TNonFreePooledMemManager.NewItem

Synopsis: Return a pointer to a new memory block

Declaration: `function NewItem : Pointer`

Visibility: `public`

Description: `NewItem` returns a pointer to an unused memory block of size `ItemSize` (147). 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` (146)

14.4.8 TNonFreePooledMemManager.EnumerateItems

Synopsis: Enumerate all items in the list

Declaration: `procedure EnumerateItems(const Method: TEnumItemsMethod)`

Visibility: `public`

Description: `EnumerateItems` will enumerate over all items in the list, passing the items to `Method`. This can be used to execute certain operations on all items in the list. (for example, simply list them)

14.4.9 TNonFreePooledMemManager.ItemSize

Synopsis: Size of an item in the list

Declaration: `Property ItemSize : Integer`

Visibility: `public`

Access: `Read`

Description: `ItemSize` is the size of a single block in the list. It's a fixed size determined when the list is created.

See also: `TNonFreePooledMemManager.Create` (146)

14.5 TPooledMemManager

14.5.1 Description

`TPooledMemManager` is a class which maintains a linked list of blocks, represented by the `TPooledMemManagerItem` (146) record. It should not be used directly, but should be descended from and the descendent should implement the actual memory manager.

14.5.2 Method overview

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

14.5.3 Property overview

Page	Property	Access	Description
150	<code>AllocatedCount</code>	r	Total number of allocated items in the list
149	<code>Count</code>	r	Number of items in the list
150	<code>FreeCount</code>	r	Number of free items in the list
150	<code>FreedCount</code>	r	Total number of freed items in the list.
149	<code>MaximumFreeCountRatio</code>	rw	Maximum ratio of free items over total items
149	<code>MinimumFreeCount</code>	rw	Minimum count of free items in the list

14.5.4 TPooledMemManager.Clear

Synopsis: Clears the list

Declaration: `procedure Clear`

Visibility: `public`

Description: `Clear` clears the list, it disposes all items in the list.

See also: `TPooledMemManager.FreedCount` ([150](#))

14.5.5 TPooledMemManager.Create

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

Declaration: `constructor Create`

Visibility: `public`

Description: `Create` initializes all necessary properties and then calls the inherited `create`.

See also: `TPooledMemManager.Destroy` ([148](#))

14.5.6 TPooledMemManager.Destroy

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

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` calls `Clear` ([148](#)) and then calls the inherited `destroy`.

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

See also: `TPooledMemManager.Create` ([148](#))

14.5.7 `TPooledMemManager.MinimumFreeCount`

Synopsis: Minimum count of free items in the list

Declaration: `Property MinimumFreeCount : Integer`

Visibility: public

Access: Read,Write

Description: `MinimumFreeCount` is the minimum number of free items in the linked list. When disposing an item in the list, the number of items is checked, and only if the required number of free items is present, the item is actually freed.

The default value is 100000

See also: `TPooledMemManager.MaximumFreeCountRatio` ([149](#))

14.5.8 `TPooledMemManager.MaximumFreeCountRatio`

Synopsis: Maximum ratio of free items over total items

Declaration: `Property MaximumFreeCountRatio : Integer`

Visibility: public

Access: Read,Write

Description: `MaximumFreeCountRatio` is the maximum ratio (divided by 8) of free elements over the total amount of elements: When disposing an item in the list, if the number of free items is higher than this ratio, the item is freed.

The default value is 8.

See also: `TPooledMemManager.MinimumFreeCount` ([149](#))

14.5.9 `TPooledMemManager.Count`

Synopsis: Number of items in the list

Declaration: `Property Count : Integer`

Visibility: public

Access: Read

Description: `Count` is the total number of items allocated from the list.

See also: `TPooledMemManager.FreeCount` ([150](#)), `TPooledMemManager.AllocatedCount` ([150](#)), `TPooledMemManager.FreedCount` ([150](#))

14.5.10 TPooledMemManager.FreeCount

Synopsis: Number of free items in the list

Declaration: `Property FreeCount : Integer`

Visibility: `public`

Access: `Read`

Description: `FreeCount` is the current total number of free items in the list.

See also: `TPooledMemManager.Count` ([149](#)), `TPooledMemManager.AllocatedCount` ([150](#)), `TPooledMemManager.FreedCount` ([150](#))

14.5.11 TPooledMemManager.AllocatedCount

Synopsis: Total number of allocated items in the list

Declaration: `Property AllocatedCount : Int64`

Visibility: `public`

Access: `Read`

Description: `AllocatedCount` is the total number of newly allocated items on the list.

See also: `TPooledMemManager.Count` ([149](#)), `TPooledMemManager.FreeCount` ([150](#)), `TPooledMemManager.FreedCount` ([150](#))

14.5.12 TPooledMemManager.FreedCount

Synopsis: Total number of freed items in the list.

Declaration: `Property FreedCount : Int64`

Visibility: `public`

Access: `Read`

Description: `FreedCount` is the total number of elements actually freed in the list.

See also: `TPooledMemManager.Count` ([149](#)), `TPooledMemManager.FreeCount` ([150](#)), `TPooledMemManager.AllocatedCount` ([150](#))

Chapter 15

Reference for unit 'process'

15.1 Used units

Table 15.1: Used units by unit 'process'

Name	Page
Classes	??
Pipes	140
sysutils	??

15.2 Overview

The `Process` unit contains the code for the `TProcess` ([153](#)) 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.

15.3 Constants, types and variables

15.3.1 Types

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

When a new process is started using `TProcess.Execute` ([155](#)), these options control the way the process is started. Note that not all options are supported on all platforms.

```
TProcessOptions= Set of (poDebugOnlyThisProcess, poDebugProcess,  
                          poDefaultErrorMode, poNewConsole,
```


Table 15.2: Enumeration values for type TProcessOption

Value	Explanation
poDebugOnlyThisProcess	Do not follow processes started by this process (Win32 only)
poDebugProcess	Allow debugging of the process (Win32 only)
poDefaultErrorMode	Use default error handling.
poNewConsole	Start a new console window for the process (Win32 only)
poNewProcessGroup	Start the process in a new process group (Win32 only)
poNoConsole	Do not allow access to the console window for the process (Win32 only)
poRunSuspended	Start the process in suspended state.
poStderrToOutPut	Redirect standard error to the standard output stream.
poUsePipes	Use pipes to redirect standard input and output.
poWaitOnExit	Wait for the process to terminate before returning.

```
poNewProcessGroup, poNoConsole, poRunSuspended,
poStderrToOutPut, poUsePipes, poWaitOnExit)
```

Set of TProcessOption (151).

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

Table 15.3: Enumeration values for type TProcessPriority

Value	Explanation
ppHigh	The process runs at higher than normal priority.
ppIdle	The process only runs when the system is idle (i.e. has nothing else to do)
ppNormal	The process runs at normal priority.
ppRealTime	The process runs at real-time priority.

This enumerated type determines the priority of the newly started process. It translates to default platform specific constants. If finer control is needed, then platform-dependent mechanism need to be used to set the priority.

```
TShowWindowOptions = (swoNone, swoHIDE, swoMaximize, swoMinimize,
swoRestore, swoShow, swoShowDefault,
swoShowMaximized, swoShowMinimized,
swoshowMinNOActive, swoShowNA, swoShowNoActivate,
swoShowNormal)
```

This type describes what the new process' main window should look like. Most of these have only effect on Windows. They are ignored on other systems.

```
TStartupOption = (suoUseShowWindow, suoUseSize, suoUsePosition,
suoUseCountChars, suoUseFillAttribute)
```

These options are mainly for Win32, and determine what should be done with the application once it's started.

Table 15.4: Enumeration values for type TShowWindowOptions

Value	Explanation
swoHIDE	The main window is hidden.
swoMaximize	The main window is maximized.
swoMinimize	The main window is minimized.
swoNone	Allow system to position the window.
swoRestore	Restore the previous position.
swoShow	Show the main window.
swoShowDefault	When showing Show the main window on
swoShowMaximized	The main window is shown maximized
swoShowMinimized	The main window is shown minimized
swoshowMinNOActive	The main window is shown minimized but not activated
swoShowNA	The main window is shown but not activated
swoShowNoActivate	The main window is shown but not activated
swoShowNormal	The main window is shown normally

Table 15.5: Enumeration values for type TStartupOption

Value	Explanation
suoUseCountChars	Use the console character width as specified in TProcess (153).
suoUseFillAttribute	Use the console fill attribute as specified in TProcess (153).
suoUsePosition	Use the window sizes as specified in TProcess (153).
suoUseShowWindow	Use the Show Window options specified in TShowWindowOption (152)
suoUseSize	Use the window sizes as specified in TProcess (153)

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

Set of TStartUpOption (152).

15.4 EProcess

15.4.1 Description

Exception raised when an error occurs in a TProcess routine.

15.5 TProcess

15.5.1 Description

TProcess is a component that can be used to start and control other processes (programs/binaries). It contains a lot of options that control how the process is started. Many of these are Win32 specific, and have no effect on other platforms, so they should be used with care.

The simplest way to use this component is to create an instance, set the CommandLine (161) property to the full pathname of the program that should be executed, and call Execute (155). To determine whether the process is still running (i.e. has not stopped executing), the Running (165) property can be checked.

More advanced techniques can be used with the Options (163) settings.

15.5.2 Method overview

Page	Property	Description
156	CloseInput	Close the input stream of the process
156	CloseOutput	Close the output stream of the process
156	CloseStderr	Close the error stream of the process
155	Create	Create a new instance of the <code>TProcess</code> class.
155	Destroy	Destroy this instance of <code>TProcess</code>
155	Execute	Execute the program with the given options
156	Resume	Resume execution of a suspended process
157	Suspend	Suspend a running process
157	Terminate	Terminate a running process
157	WaitOnExit	Wait for the program to stop executing.

15.5.3 Property overview

Page	Property	Access	Description
161	Active	rw	Start or stop the process.
161	ApplicationName	rw	Name of the application to start
161	CommandLine	rw	Command-line to execute
162	ConsoleTitle	rw	Title of the console window
162	CurrentDirectory	rw	Working directory of the process.
162	Desktop	rw	Desktop on which to start the process.
163	Environment	rw	Environment variables for the new process
160	ExitStatus	r	Exit status of the process.
168	FillAttribute	rw	Color attributes of the characters in the console window (Windows only)
158	Handle	r	Handle of the process
161	InheritHandles	rw	Should the created process inherit the open handles of the current process.
159	Input	r	Stream connected to standard input of the process.
163	Options	rw	Options to be used when starting the process.
160	Output	r	Stream connected to standard output of the process.
164	Priority	rw	Priority at which the process is running.
158	ProcessHandle	r	Alias for Handle (158)
159	ProcessID	r	ID of the process.
165	Running	r	Determines wheter the process is still running.
165	ShowWindow	rw	Determines how the process main window is shown (Windows only)
164	StartupOptions	rw	Additional (Windows) startup options
160	Stderr	r	Stream connected to standard diagnostic output of the process.
158	ThreadHandle	r	Main process thread handle
159	ThreadID	r	ID of the main process thread
166	WindowColumns	rw	Number of columns in console window (windows only)
166	WindowHeight	rw	Height of the process main window
166	WindowLeft	rw	X-coordinate of the initial window (Windows only)
158	WindowRect	rw	Positions for the main program window.
167	WindowRows	rw	Number of rows in console window (Windows only)
167	WindowTop	rw	Y-coordinate of the initial window (Windows only)
167	WindowWidth	rw	Height of the process main window (Windows only)

15.5.4 TProcess.Create

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

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

Visibility: `public`

Description: `Create` creates a new instance of the `TProcess` class. After calling the inherited constructor, it simply sets some default values.

15.5.5 TProcess.Destroy

Synopsis: Destroy this instance of `TProcess`

Declaration: `destructor Destroy; Override`

Visibility: `public`

Description: `Destroy` cleans up this instance of `TProcess`. Prior to calling the inherited destructor, it cleans up any streams that may have been created. If a process was started and is still executed, it is *not* stopped, but the standard input/output/stderr streams are no longer available, because they have been destroyed.

Errors: None.

See also: `TProcess.Create` ([155](#))

15.5.6 TProcess.Execute

Synopsis: Execute the program with the given options

Declaration: `procedure Execute; Virtual`

Visibility: `public`

Description: `Execute` actually executes the program as specified in `CommandLine` ([161](#)), applying as much as of the specified options as supported on the current platform.

If the `poWaitOnExit` option is specified in `Options` ([163](#)), 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` ([157](#)) call can be used to wait for it to close, or the `Running` ([165](#)) call can be used to check whether it is still running.

The `TProcess.Terminate` ([157](#)) call can be used to terminate the program if it is still running, or the `Suspend` ([157](#)) call can be used to temporarily stop the program's execution.

The `ExitStatus` ([160](#)) function can be used to check the program's exit status, after it has stopped executing.

Errors: On error a `EProcess` ([153](#)) exception is raised.

See also: `TProcess.Running` ([165](#)), `TProcess.WaitOnExit` ([157](#)), `TProcess.Terminate` ([157](#)), `TProcess.Suspend` ([157](#)), `TProcess.Resume` ([156](#)), `TProcess.ExitStatus` ([160](#))

15.5.7 TProcess.CloseInput

Synopsis: Close the input stream of the process

Declaration: `procedure CloseInput; Virtual`

Visibility: `public`

Description: `CloseInput` closes the input file descriptor of the process, that is, it closes the handle of the pipe to standard input of the process.

See also: `TProcess.Input` (159), `TProcess.StdErr` (160), `TProcess.Output` (160), `TProcess.CloseOutput` (156), `TProcess.CloseStdErr` (156)

15.5.8 TProcess.CloseOutput

Synopsis: Close the output stream of the process

Declaration: `procedure CloseOutput; Virtual`

Visibility: `public`

Description: `CloseOutput` closes the output file descriptor of the process, that is, it closes the handle of the pipe to standard output of the process.

See also: `TProcess.Output` (160), `TProcess.Input` (159), `TProcess.StdErr` (160), `TProcess.CloseInput` (156), `TProcess.CloseStdErr` (156)

15.5.9 TProcess.CloseStderr

Synopsis: Close the error stream of the process

Declaration: `procedure CloseStderr; Virtual`

Visibility: `public`

Description: `CloseStdErr` closes the standard error file descriptor of the process, that is, it closes the handle of the pipe to standard error output of the process.

See also: `TProcess.Output` (160), `TProcess.Input` (159), `TProcess.StdErr` (160), `TProcess.CloseInput` (156), `TProcess.CloseStdErr` (156)

15.5.10 TProcess.Resume

Synopsis: Resume execution of a suspended process

Declaration: `function Resume : Integer; Virtual`

Visibility: `public`

Description: `Resume` should be used to let a suspended process resume its execution. It should be called in particular when the `poRunSuspended` flag is set in `Options` (163).

Errors: None.

See also: `TProcess.Suspend` (157), `TProcess.Options` (163), `TProcess.Execute` (155), `TProcess.Terminate` (157)

15.5.11 TProcess.Suspend

Synopsis: Suspend a running process

Declaration: `function Suspend : Integer; Virtual`

Visibility: `public`

Description: `Suspend` suspends a running process. If the call is successful, the process is suspended: it stops running, but can be made to execute again using the `Resume` (156) call.

`Suspend` is fundamentally different from `TProcess.Terminate` (157) which actually stops the process.

Errors: On error, a nonzero result is returned.

See also: `TProcess.Options` (163), `TProcess.Resume` (156), `TProcess.Terminate` (157), `TProcess.Execute` (155)

15.5.12 TProcess.Terminate

Synopsis: Terminate a running process

Declaration: `function Terminate(AExitCode: Integer) : Boolean; Virtual`

Visibility: `public`

Description: `Terminate` stops the execution of the running program. It effectively stops the program.

On Windows, the program will report an exit code of `AExitCode`, on other systems, this value is ignored.

Errors: On error, a nonzero value is returned.

See also: `TProcess.ExitStatus` (160), `TProcess.Suspend` (157), `TProcess.Execute` (155), `TProcess.WaitOnExit` (157)

15.5.13 TProcess.WaitOnExit

Synopsis: Wait for the program to stop executing.

Declaration: `function WaitOnExit : Boolean`

Visibility: `public`

Description: `WaitOnExit` waits for the running program to exit. It returns `True` if the wait was successful, or `False` if there was some error waiting for the program to exit.

Note that the return value of this function has changed. The old return value was a `DWord` with a platform dependent error code. To make things consistent and cross-platform, a boolean return type was used.

Errors: On error, `False` is returned. No extended error information is available, as it is highly system dependent.

See also: `TProcess.ExitStatus` (160), `TProcess.Terminate` (157), `TProcess.Running` (165)

15.5.14 TProcess.WindowRect

Synopsis: Positions for the main program window.

Declaration: `Property WindowRect : Trect`

Visibility: `public`

Access: `Read,Write`

Description: `WindowRect` can be used to specify the position of

15.5.15 TProcess.Handle

Synopsis: Handle of the process

Declaration: `Property Handle : THandle`

Visibility: `public`

Access: `Read`

Description: `Handle` identifies the process. In Unix systems, this is the process ID. On windows, this is the process handle. It can be used to signal the process.

The handle is only valid after `TProcess.Execute` (155) has been called. It is not reset after the process stopped.

See also: `TProcess.ThreadHandle` (158), `TProcess.ProcessID` (159), `TProcess.ThreadID` (159)

15.5.16 TProcess.ProcessHandle

Synopsis: Alias for `Handle` (158)

Declaration: `Property ProcessHandle : THandle`

Visibility: `public`

Access: `Read`

Description: `ProcessHandle` equals `Handle` (158) and is provided for completeness only.

See also: `TProcess.Handle` (158), `TProcess.ThreadHandle` (158), `TProcess.ProcessID` (159), `TProcess.ThreadID` (159)

15.5.17 TProcess.ThreadHandle

Synopsis: Main process thread handle

Declaration: `Property ThreadHandle : THandle`

Visibility: `public`

Access: `Read`

Description: `ThreadHandle` is the main process thread handle. On Unix, this is the same as the process ID, on Windows, this may be a different handle than the process handle.

The handle is only valid after `TProcess.Execute` (155) has been called. It is not reset after the process stopped.

See also: `TProcess.Handle` (158), `TProcess.ProcessID` (159), `TProcess.ThreadID` (159)

15.5.18 TProcess.ProcessID

Synopsis: ID of the process.

Declaration: `Property ProcessID : Integer`

Visibility: `public`

Access: `Read`

Description: `ProcessID` is the ID of the process. It is the same as the handle of the process on Unix systems, but on Windows it is different from the process Handle.

The ID is only valid after `TProcess.Execute` (155) has been called. It is not reset after the process stopped.

See also: `TProcess.Handle` (158), `TProcess.ThreadHandle` (158), `TProcess.ThreadID` (159)

15.5.19 TProcess.ThreadID

Synopsis: ID of the main process thread

Declaration: `Property ThreadID : Integer`

Visibility: `public`

Access: `Read`

Description: `ProcessID` is the ID of the main process thread. It is the same as the handle of the main process thread (or the process itself) on Unix systems, but on Windows it is different from the thread Handle.

The ID is only valid after `TProcess.Execute` (155) has been called. It is not reset after the process stopped.

See also: `TProcess.ProcessID` (159), `TProcess.Handle` (158), `TProcess.ThreadHandle` (158)

15.5.20 TProcess.Input

Synopsis: Stream connected to standard input of the process.

Declaration: `Property Input : TOutputPipeStream`

Visibility: `public`

Access: `Read`

Description: `Input` is a stream which is connected to the process' standard input file handle. Anything written to this stream can be read by the process.

The `Input` stream is only instantiated when the `poUsePipes` flag is used in `Options` (163).

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` (160), `TProcess.StdErr` (160), `TProcess.Options` (163), `TProcessOption` (151)

15.5.21 TProcess.Output

Synopsis: Stream connected to standard output of the process.

Declaration: `Property Output : TInputPipeStream`

Visibility: `public`

Access: `Read`

Description: `Output` is a stream which is connected to the process' standard output file handle. Anything written to standard output by the created process can be read from this stream.

The `Output` stream is only instantiated when the `poUsePipes` flag is used in `Options` (163).

The `Output` stream also contains any data written to standard diagnostic output (`stderr`) when the `poStdErrToOutPut` flag is used in `Options` (163).

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` (159), `TProcess.StdErr` (160), `TProcess.Options` (163), `TProcessOption` (151)

15.5.22 TProcess.Stderr

Synopsis: Stream connected to standard diagnostic output of the process.

Declaration: `Property Stderr : TInputPipeStream`

Visibility: `public`

Access: `Read`

Description: `StdErr` is a stream which is connected to the process' standard diagnostic output file handle (`StdErr`). Anything written to standard diagnostic output by the created process can be read from this stream.

The `StdErr` stream is only instantiated when the `poUsePipes` flag is used in `Options` (163).

The `Output` stream equals the `Output` (160) when the `poStdErrToOutPut` flag is used in `Options` (163).

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` (159), `TProcess.Output` (160), `TProcess.Options` (163), `TProcessOption` (151)

15.5.23 TProcess.ExitStatus

Synopsis: Exit status of the process.

Declaration: `Property ExitStatus : Integer`

Visibility: `public`

Access: `Read`

Description: `ExitStatus` contains the exit status as reported by the process when it stopped executing. The value of this property is only meaningful when the process is no longer running. If it is not running then the value is zero.

See also: `TProcess.Running` (165), `TProcess.Terminate` (157)

15.5.24 TProcess.InheritHandles

Synopsis: Should the created process inherit the open handles of the current process.

Declaration: `Property InheritHandles : Boolean`

Visibility: `public`

Access: `Read,Write`

Description: `InheritHandles` determines whether the created process inherits the open handles of the current process (value `True`) or not (`False`).

On Unix, setting this variable has no effect.

See also: `TProcess.InPut` ([159](#)), `TProcess.Output` ([160](#)), `TProcess.StdErr` ([160](#))

15.5.25 TProcess.Active

Synopsis: Start or stop the process.

Declaration: `Property Active : Boolean`

Visibility: `published`

Access: `Read,Write`

Description: `Active` starts the process if it is set to `True`, or terminates the process if set to `False`. It's mostly intended for use in an IDE.

See also: `TProcess.Execute` ([155](#)), `TProcess.Terminate` ([157](#))

15.5.26 TProcess.ApplicationName

Synopsis: Name of the application to start

Declaration: `Property ApplicationName : String`

Visibility: `published`

Access: `Read,Write`

Description: `ApplicationName` is an alias for `TProcess.CommandLine` ([161](#)). It's mostly for use in the Windows `CreateProcess` call. If `CommandLine` is not set, then `ApplicationName` will be used instead.

Note that either `CommandLine` or `ApplicationName` must be set prior to calling `Execute`.

See also: `TProcess.CommandLine` ([161](#))

15.5.27 TProcess.CommandLine

Synopsis: Command-line to execute

Declaration: `Property CommandLine : String`

Visibility: `published`

Access: `Read,Write`

Description: `CommandLine` is the command-line to be executed: this is the name of the program to be executed, followed by any options it should be passed.

If the command to be executed or any of the arguments contains whitespace (space, tab character, linefeed character) it should be enclosed in single or double quotes.

If no absolute pathname is given for the command to be executed, it is searched for in the `PATH` environment variable. On Windows, the current directory always will be searched first. On other platforms, this is not so.

Note that either `CommandLine` or `ApplicationName` must be set prior to calling `Execute`.

See also: `TProcess.ApplicationName` ([161](#))

15.5.28 TProcess.ConsoleTitle

Synopsis: Title of the console window

Declaration: `Property ConsoleTitle : String`

Visibility: published

Access: Read,Write

Description: `ConsoleTitle` is used on Windows when executing a console application: it specifies the title caption of the console window. On other platforms, this property is currently ignored.

Changing this property after the process was started has no effect.

See also: `TProcess.WindowColumns` ([166](#)), `TProcess.WindowRows` ([167](#))

15.5.29 TProcess.CurrentDirectory

Synopsis: Working directory of the process.

Declaration: `Property CurrentDirectory : String`

Visibility: published

Access: Read,Write

Description: `CurrentDirectory` specifies the working directory of the newly started process.

Changing this property after the process was started has no effect.

See also: `TProcess.Environment` ([163](#))

15.5.30 TProcess.Desktop

Synopsis: Desktop on which to start the process.

Declaration: `Property Desktop : String`

Visibility: published

Access: Read,Write

Description: `Desktop` is used on Windows to determine on which desktop the process' main window should be shown. Leaving this empty means the process is started on the same desktop as the currently running process.

Changing this property after the process was started has no effect.

On unix, this parameter is ignored.

See also: [TProcess.Input \(159\)](#), [TProcess.Output \(160\)](#), [TProcess.StdErr \(160\)](#)

15.5.31 TProcess.Environment

Synopsis: Environment variables for the new process

Declaration: `Property Environment : TStrings`

Visibility: published

Access: Read,Write

Description: `Environment` contains the environment for the new process; it's a list of `Name=Value` pairs, one per line.

If it is empty, the environment of the current process is passed on to the new process.

See also: [TProcess.Options \(163\)](#)

15.5.32 TProcess.Options

Synopsis: Options to be used when starting the process.

Declaration: `Property Options : TProcessOptions`

Visibility: published

Access: Read,Write

Description: `Options` determine how the process is started. They should be set before the [Execute \(155\)](#) call is made.

Table 15.6:

option	Meaning
<code>poRunSuspended</code>	Start the process in suspended state.
<code>poWaitOnExit</code>	Wait for the process to terminate before returning.
<code>poUsePipes</code>	Use pipes to redirect standard input and output.
<code>poStderrToOutPut</code>	Redirect standard error to the standard output stream.
<code>poNoConsole</code>	Do not allow access to the console window for the process (Win32 only)
<code>poNewConsole</code>	Start a new console window for the process (Win32 only)
<code>poDefaultErrorMode</code>	Use default error handling.
<code>poNewProcessGroup</code>	Start the process in a new process group (Win32 only)
<code>poDebugProcess</code>	Allow debugging of the process (Win32 only)
<code>poDebugOnlyThisProcess</code>	Do not follow processes started by this process (Win32 only)

See also: [TProcessOption \(151\)](#), [TProcessOptions \(152\)](#), [TProcess.Priority \(164\)](#), [TProcess.StartUpOptions \(164\)](#)

15.5.33 TProcess.Priority

Synopsis: Priority at which the process is running.

Declaration: `Property Priority : TProcessPriority`

Visibility: published

Access: Read,Write

Description: `Priority` determines the priority at which the process is running.

Table 15.7:

Priority	Meaning
<code>ppHigh</code>	The process runs at higher than normal priority.
<code>ppIdle</code>	The process only runs when the system is idle (i.e. has nothing else to do)
<code>ppNormal</code>	The process runs at normal priority.
<code>ppRealTime</code>	The process runs at real-time priority.

Note that not all priorities can be set by any user. Usually, only users with administrative rights (the root user on Unix) can set a higher process priority.

On unix, the process priority is mapped on `Nice` values as follows:

Table 15.8:

Priority	Nice value
<code>ppHigh</code>	20
<code>ppIdle</code>	20
<code>ppNormal</code>	0
<code>ppRealTime</code>	-20

See also: `TProcessPriority` ([152](#))

15.5.34 TProcess.StartupOptions

Synopsis: Additional (Windows) startup options

Declaration: `Property StartupOptions : TStartupOptions`

Visibility: published

Access: Read,Write

Description: `StartupOptions` contains additional startup options, used mostly on Windows system. They determine which other window layout properties are taken into account when starting the new process.

See also: `TProcess.ShowWindow` ([165](#)), `TProcess.WindowHeight` ([166](#)), `TProcess.WindowWidth` ([167](#)), `TProcess.WindowLeft` ([166](#)), `TProcess.WindowTop` ([167](#)), `TProcess.WindowColumns` ([166](#)), `TProcess.WindowRows` ([167](#)), `TProcess.FillAttribute` ([168](#))

Table 15.9:

Priority	Meaning
<code>suoUseShowWindow</code>	Use the Show Window options specified in <code>ShowWindow</code> (165)
<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> (168).

15.5.35 TProcess.Running

Synopsis: Determines wheter the process is still running.

Declaration: `Property Running : Boolean`

Visibility: published

Access: Read

Description: `Running` can be read to determine whether the process is still running.

See also: `TProcess.Terminate` (157), `TProcess.Active` (161), `TProcess.ExitStatus` (160)

15.5.36 TProcess.ShowWindow

Synopsis: Determines how the process main window is shown (Windows only)

Declaration: `Property ShowWindow : TShowWindowOptions`

Visibility: published

Access: Read,Write

Description: `ShowWindow` determines how the process' main window is shown. It is useful only on Windows.

Table 15.10:

Option	Meaning
<code>swoNone</code>	Allow system to position the window.
<code>swoHIDE</code>	The main window is hidden.
<code>swoMaximize</code>	The main window is maximized.
<code>swoMinimize</code>	The main window is minimized.
<code>swoRestore</code>	Restore the previous position.
<code>swoShow</code>	Show the main window.
<code>swoShowDefault</code>	When showing Show the main window on a default position
<code>swoShowMaximized</code>	The main window is shown maximized
<code>swoShowMinimized</code>	The main window is shown minimized
<code>swoshowMinNOActive</code>	The main window is shown minimized but not activated
<code>swoShowNA</code>	The main window is shown but not activated
<code>swoShowNoActivate</code>	The main window is shown but not activated
<code>swoShowNormal</code>	The main window is shown normally

15.5.37 TProcess.WindowColumns

Synopsis: Number of columns in console window (windows only)

Declaration: `Property WindowColumns : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowColumns` is the number of columns in the console window, used to run the command in. This property is only effective if `suoUseCountChars` is specified in `StartupOptions` (164)

See also: `TProcess.WindowHeight` (166), `TProcess.WindowWidth` (167), `TProcess.WindowLeft` (166), `TProcess.WindowTop` (167), `TProcess.WindowRows` (167), `TProcess.FillAttribute` (168), `TProcess.StartupOptions` (164)

15.5.38 TProcess.WindowHeight

Synopsis: Height of the process main window

Declaration: `Property WindowHeight : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowHeight` is the initial height (in pixels) of the process' main window. This property is only effective if `suoUseSize` is specified in `StartupOptions` (164)

See also: `TProcess.WindowWidth` (167), `TProcess.WindowLeft` (166), `TProcess.WindowTop` (167), `TProcess.WindowColumns` (166), `TProcess.WindowRows` (167), `TProcess.FillAttribute` (168), `TProcess.StartupOptions` (164)

15.5.39 TProcess.WindowLeft

Synopsis: X-coordinate of the initial window (Windows only)

Declaration: `Property WindowLeft : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowLeft` is the initial X coordinate (in pixels) of the process' main window, relative to the left border of the desktop. This property is only effective if `suoUsePosition` is specified in `StartupOptions` (164)

See also: `TProcess.WindowHeight` (166), `TProcess.WindowWidth` (167), `TProcess.WindowTop` (167), `TProcess.WindowColumns` (166), `TProcess.WindowRows` (167), `TProcess.FillAttribute` (168), `TProcess.StartupOptions` (164)

15.5.40 TProcess.WindowRows

Synopsis: Number of rows in console window (Windows only)

Declaration: `Property WindowRows : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowRows` is the number of rows in the console window, used to run the command in. This property is only effective if `suoUseCountChars` is specified in `StartupOptions` (164)

See also: `TProcess.WindowHeight` (166), `TProcess.WindowWidth` (167), `TProcess.WindowLeft` (166), `TProcess.WindowTop` (167), `TProcess.WindowColumns` (166), `TProcess.FillAttribute` (168), `TProcess.StartupOptions` (164)

15.5.41 TProcess.WindowTop

Synopsis: Y-coordinate of the initial window (Windows only)

Declaration: `Property WindowTop : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowTop` is the initial Y coordinate (in pixels) of the process' main window, relative to the top border of the desktop. This property is only effective if `suoUsePosition` is specified in `StartupOptions` (164)

See also: `TProcess.WindowHeight` (166), `TProcess.WindowWidth` (167), `TProcess.WindowLeft` (166), `TProcess.WindowColumns` (166), `TProcess.WindowRows` (167), `TProcess.FillAttribute` (168), `TProcess.StartupOptions` (164)

15.5.42 TProcess.WindowWidth

Synopsis: Height of the process main window (Windows only)

Declaration: `Property WindowWidth : Cardinal`

Visibility: published

Access: Read,Write

Description: `WindowWidth` is the initial width (in pixels) of the process' main window. This property is only effective if `suoUseSize` is specified in `StartupOptions` (164)

See also: `TProcess.WindowHeight` (166), `TProcess.WindowLeft` (166), `TProcess.WindowTop` (167), `TProcess.WindowColumns` (166), `TProcess.WindowRows` (167), `TProcess.FillAttribute` (168), `TProcess.StartupOptions` (164)

15.5.43 TProcess.FillAttribute

Synopsis: Color attributes of the characters in the console window (Windows only)

Declaration: `Property FillAttribute : Cardinal`

Visibility: `published`

Access: `Read, Write`

Description: `FillAttribute` is a `WORD` value which specifies the background and foreground colors of the console window.

See also: `TProcess.WindowHeight` ([166](#)), `TProcess.WindowWidth` ([167](#)), `TProcess.WindowLeft` ([166](#)), `TProcess.WindowTop` ([167](#)), `TProcess.WindowColumns` ([166](#)), `TProcess.WindowRows` ([167](#)), `TProcess.StartupOptions` ([164](#))

Chapter 16

Reference for unit 'streamcoll'

16.1 Used units

Table 16.1: Used units by unit 'streamcoll'

Name	Page
Classes	??
sysutils	??

16.2 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` (172) 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.

16.3 Procedures and functions

16.3.1 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` (171) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColReadDateTime \(170\)](#), [ColWriteBoolean \(171\)](#), [ColReadString \(171\)](#), [ColReadInteger \(170\)](#), [ColReadFloat \(170\)](#), [ColReadCurrency \(170\)](#)

16.3.2 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 (171)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColReadDateTime \(170\)](#), [ColReadBoolean \(169\)](#), [ColReadString \(171\)](#), [ColReadInteger \(170\)](#), [ColReadFloat \(170\)](#), [ColWriteCurrency \(171\)](#)

16.3.3 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 (171)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColWriteDateTime \(171\)](#), [ColReadBoolean \(169\)](#), [ColReadString \(171\)](#), [ColReadInteger \(170\)](#), [ColReadFloat \(170\)](#), [ColReadCurrency \(170\)](#)

16.3.4 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 (172)` and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: [ColReadDateTime \(170\)](#), [ColReadBoolean \(169\)](#), [ColReadString \(171\)](#), [ColReadInteger \(170\)](#), [ColWriteFloat \(172\)](#), [ColReadCurrency \(170\)](#)

16.3.5 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` (172) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: `ColReadDateTime` (170), `ColReadBoolean` (169), `ColReadString` (171), `ColWriteInteger` (172), `ColReadFloat` (170), `ColReadCurrency` (170)

16.3.6 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` (172) and returns the read value. The value cannot be read and written across systems that have different endian values.

See also: `ColReadDateTime` (170), `ColReadBoolean` (169), `ColWriteString` (172), `ColReadInteger` (170), `ColReadFloat` (170), `ColReadCurrency` (170)

16.3.7 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` (169), `ColWriteString` (172), `ColWriteInteger` (172), `ColWriteCurrency` (171), `ColWriteDateTime` (171), `ColWriteFloat` (172)

16.3.8 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` (171), `ColWriteString` (172), `ColWriteInteger` (172), `ColWriteDateTime` (171), `ColWriteFloat` (172), `ColReadCurrency` (170)

16.3.9 ColWriteDateTime

Synopsis: Write a `TDateTime` value to stream

Declaration: `procedure ColWriteDateTime(S: TStream; AValue: TDateTime)`

Visibility: default

Description: `ColWriteDateTime` writes the `TDateTime` `AValue` to the stream `S`.

See also: `ColReadDateTime` (170), `ColWriteBoolean` (171), `ColWriteString` (172), `ColWriteInteger` (172), `ColWriteFloat` (172), `ColWriteCurrency` (171)

16.3.10 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` (171), `ColWriteBoolean` (171), `ColWriteString` (172), `ColWriteInteger` (172), `ColReadFloat` (170), `ColWriteCurrency` (171)

16.3.11 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` (171), `ColWriteString` (172), `ColReadInteger` (170), `ColWriteCurrency` (171), `ColWriteDateTime` (171)

16.3.12 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` (171), `ColReadString` (171), `ColWriteInteger` (172), `ColWriteCurrency` (171), `ColWriteDateTime` (171), `ColWriteFloat` (172)

16.4 EStreamColl

16.4.1 Description

Exception raised when an error occurs when streaming the collection.

16.5 TStreamCollection

16.5.1 Description

`TStreamCollection` is a `TCollection` (??) descendent which implements 2 calls `LoadFromStream` (173) and `SaveToStream` (173) which load and save the contents of the collection to a stream.

The collection items must be descendents of the `TStreamCollectionItem` (174) class for the streaming to work correctly.

Note that the stream must be used to load collections of the same type.

16.5.2 Method overview

Page	Property	Description
173	LoadFromStream	Load the collection from a stream
173	SaveToStream	Load the collection from the stream.

16.5.3 Property overview

Page	Property	Access	Description
173	Streaming	r	Indicates whether the collection is currently being written to stream

16.5.4 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` ([173](#)). 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` ([173](#))

16.5.5 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` ([173](#)). 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` ([173](#))

16.5.6 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` ([173](#)), `TStreamCollection.SaveToStream` ([173](#))

16.6 TStreamCollectionItem

16.6.1 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 ([172](#)) to save or load the item from the stream.

Chapter 17

Reference for unit 'streamex'

17.1 Used units

Table 17.1: Used units by unit 'streamex'

Name	Page
Classes	??

17.2 Overview

streamex implements some extensions to be used together with streams from the classes unit.

17.3 TBidirBinaryObjectReader

17.3.1 Description

`TBidirBinaryObjectReader` is a class descendent from `TBinaryObjectReader` (??), which implements the necessary support for BiDi data: the position in the stream (not available in the standard streaming) is emulated.

17.3.2 Property overview

Page	Property	Access	Description
175	Position	rw	Position in the stream

17.3.3 TBidirBinaryObjectReader.Position

Synopsis: Position in the stream

Declaration: `Property Position : LongInt`

Visibility: public

Access: Read,Write

Description: `Position` exposes the position of the stream in the reader for use in the `TDelphiReader` (176) class.

See also: `TDelphiReader` (176)

17.4 TBidirBinaryObjectWriter

17.4.1 Description

`TBidirBinaryObjectReader` is a class descendent from `TBinaryObjectWriter` (??), which implements the necessary support for BiDi data.

17.4.2 Property overview

Page	Property	Access	Description
176	<code>Position</code>	rw	Position in the stream

17.4.3 TBidirBinaryObjectWriter.Position

Synopsis: Position in the stream

Declaration: `Property Position : LongInt`

Visibility: public

Access: Read,Write

Description: `Position` exposes the position of the stream in the writer for use in the `TDelphiWriter` (177) class.

See also: `TDelphiWriter` (177)

17.5 TDelphiReader

17.5.1 Description

`TDelphiReader` is a descendent of `TReader` which has support for BiDi Streaming. It overrides the stream reading methods for strings, and makes sure the stream can be positioned in the case of strings. For this purpose, it makes use of the `TBidirBinaryObjectReader` (175) driver class.

17.5.2 Method overview

Page	Property	Description
177	<code>GetDriver</code>	Return the driver class as a <code>TBidirBinaryObjectReader</code> (175) class
177	<code>Read</code>	Read data from stream
177	<code>ReadStr</code>	Overrides the standard <code>ReadStr</code> method

17.5.3 Property overview

Page	Property	Access	Description
177	<code>Position</code>	rw	Position in the stream

17.5.4 TDelphiReader.GetDriver

Synopsis: Return the driver class as a `TBidirBinaryObjectReader` (175) class

Declaration: `function GetDriver : TBidirBinaryObjectReader`

Visibility: public

Description: `GetDriver` simply returns the used driver and typecasts it as `TBidirBinaryObjectReader` (175) class.

See also: `TBidirBinaryObjectReader` (175)

17.5.5 TDelphiReader.ReadStr

Synopsis: Overrides the standard `ReadStr` method

Declaration: `function ReadStr : String`

Visibility: public

Description: `ReadStr` makes sure the `TBidirBinaryObjectReader` (175) methods are used, to store additional information about the stream position when reading the strings.

See also: `TBidirBinaryObjectReader` (175)

17.5.6 TDelphiReader.Read

Synopsis: Read data from stream

Declaration: `procedure Read(var Buf; Count: LongInt); Override`

Visibility: public

Description: `Read` reads raw data from the stream. It reads `Count` bytes from the stream and places them in `Buf`. It forces the use of the `TBidirBinaryObjectReader` (175) class when reading.

See also: `TBidirBinaryObjectReader` (175), `TDelphiReader.Position` (177)

17.5.7 TDelphiReader.Position

Synopsis: Position in the stream

Declaration: `Property Position : LongInt`

Visibility: public

Access: Read, Write

Description: Position in the stream.

See also: `TDelphiReader.Read` (177)

17.6 TDelphiWriter

17.6.1 Description

`TDelphiWriter` is a descendent of `TWriter` which has support for BiDi Streaming. It overrides the stream writing methods for strings, and makes sure the stream can be positioned in the case of strings. For this purpose, it makes use of the `TBidirBinaryObjectWriter` (176) driver class.

17.6.2 Method overview

Page	Property	Description
178	FlushBuffer	Flushes the stream buffer
178	GetDriver	Return the driver class as a TBidirBinaryObjectWriter (176) class
178	Write	Write raw data to the stream
178	WriteStr	Write a string to the stream
179	WriteValue	Write value type

17.6.3 Property overview

Page	Property	Access	Description
179	Position	rw	Position in the stream

17.6.4 TDelphiWriter.GetDriver

Synopsis: Return the driver class as a [TBidirBinaryObjectWriter \(176\)](#) class

Declaration: `function GetDriver : TBidirBinaryObjectWriter`

Visibility: public

Description: `GetDriver` simply returns the used driver and typecasts it as [TBidirBinaryObjectWriter \(176\)](#) class.

See also: [TBidirBinaryObjectWriter \(176\)](#)

17.6.5 TDelphiWriter.FlushBuffer

Synopsis: Flushes the stream buffer

Declaration: `procedure FlushBuffer`

Visibility: public

Description: `FlushBuffer` flushes the internal buffer of the writer. It simply calls the `FlushBuffer` method of the driver class.

17.6.6 TDelphiWriter.Write

Synopsis: Write raw data to the stream

Declaration: `procedure Write(const Buf; Count: LongInt); Override`

Visibility: public

Description: `Write` writes `Count` bytes from `Buf` to the buffer, updating the position as needed.

17.6.7 TDelphiWriter.WriteStr

Synopsis: Write a string to the stream

Declaration: `procedure WriteStr(const Value: String)`

Visibility: public

Description: `WriteStr` writes a string to the stream, forcing the use of the `TBidirBinaryObjectWriter` (176) class methods, which update the position of the stream.

See also: `TBidirBinaryObjectWriter` (176)

17.6.8 `TDelphiWriter.WriteValue`

Synopsis: Write value type

Declaration: `procedure WriteValue(Value: TValueType)`

Visibility: public

Description: `WriteValue` overrides the same method in `TWriter` to force the use of the `TBidirBinaryObjectWriter` (176) methods, which update the position of the stream.

See also: `TBidirBinaryObjectWriter` (176)

17.6.9 `TDelphiWriter.Position`

Synopsis: Position in the stream

Declaration: `Property Position : LongInt`

Visibility: public

Access: Read, Write

Description: `Position` exposes the position in the stream as exposed by the `TBidirBinaryObjectWriter` (176) instance used when streaming.

See also: `TBidirBinaryObjectWriter` (176)

Chapter 18

Reference for unit 'StreamIO'

18.1 Used units

Table 18.1: Used units by unit 'StreamIO'

Name	Page
Classes	??
sysutils	??

18.2 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.

18.3 Procedures and functions

18.3.1 AssignStream

Synopsis: Assign a text file to a stream.

Declaration: `procedure AssignStream(var F: Textfile; Stream: TStream)`

Visibility: default

Description: `AssignStream` assigns the stream `Stream` to file `F`. The file can subsequently be used to write to the stream, using the standard `Write` (??) calls.

Before writing, call `Rewrite` (??) on the stream. Before reading, call `Reset` (??).

Errors: if `Stream` is `Nil`, an exception will be raised.

See also: `#rtl.classes.TStream` (??), `GetStream` ([181](#))

18.3.2 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` ([180](#)).

Errors: An invalid class reference will be returned if the file was not associated with a stream.

See also: `AssignStream` ([180](#)), `#rtl.classes.TStream` (??)

Chapter 19

Reference for unit 'zstream'

19.1 Used units

Table 19.1: Used units by unit 'zstream'

Name	Page
Classes	??
paszlib	182
sysutils	??
zbase	182

19.2 Overview

The `ZStream` unit implements a `TStream` (??) descendent (`TCompressionStream` ([183](#))) 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` ([186](#))) and decompresses it with the inflate algorithm.

19.3 Constants, types and variables

19.3.1 Types

`TCompressionLevel = (clNone, clFastest, clDefault, clMax)`

Compression level for the deflate algorithm

`TGZOpenMode = (gzOpenRead, gzOpenWrite)`

Open mode for gzip file.

Table 19.2: Enumeration values for type `TCompressionLevel`

Value	Explanation
<code>clDefault</code>	Use default compression
<code>clFastest</code>	Use fast (but less) compression.
<code>clMax</code>	Use maximum compression
<code>clNone</code>	Do not use compression, just copy data.

Table 19.3: Enumeration values for type `TGZOpenMode`

Value	Explanation
<code>gzOpenRead</code>	Open file for reading
<code>gzOpenWrite</code>	Open file for writing

19.4 `ECompressionError`

19.4.1 Description

`ECompressionError` is the exception class used by the `TCompressionStream` (183) class.

19.5 `EDecompressionError`

19.5.1 Description

`EDecompressionError` is the exception class used by the `TDecompressionStream` (186) class.

19.6 `EZlibError`

19.6.1 Description

Errors which occur in the `zstream` unit are signaled by raising an `EZlibError` exception descendant.

19.7 `TCompressionStream`

19.7.1 Description

`TCompressionStream`

19.7.2 Method overview

Page	Property	Description
184	Create	Create a new instance of the compression stream.
184	Destroy	Flush data to the output stream and destroys the compression stream.
184	Read	Overridden to raise an exception.
185	Seek	Overrides seek to raise an exception.
185	Write	Write data to the stream

19.7.3 Property overview

Page	Property	Access	Description
185	CompressionRate	r	Running compression rate of compression stream
185	OnProgress		Progress handler

19.7.4 TCompressionStream.Create

Synopsis: Create a new instance of the compression stream.

Declaration: `constructor Create(CompressionLevel: TCompressionLevel; Dest: TStream; ASkipHeader: Boolean)`

Visibility: public

Description: `Create` creates a new instance of the compression stream. It merely calls the inherited constructor with the destination stream `Dest` and stores the compression level.

If `ASkipHeader` is set to `True`, the method will not write the block header to the stream. This is required for deflated data in a zip file.

Note that the compressed data is only completely written after the compression stream is destroyed.

See also: `TCompressionStream.Destroy` ([184](#))

19.7.5 TCompressionStream.Destroy

Synopsis: Flush data to the output stream and destroys the compression stream.

Declaration: `destructor Destroy; Override`

Visibility: public

Description: `Destroy` flushes the output stream: any compressed data not yet written to the output stream are written, and the deflate structures are cleaned up.

Errors: None.

See also: `TCompressionStream.Create` ([184](#))

19.7.6 TCompressionStream.Read

Synopsis: Overridden to raise an exception.

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

Visibility: public

Description: The `Read` method of `TStream` is overridden, and always raises an exception, because `TCompressionStream` is write-only.

Errors: An `ECompressionError` ([183](#)) exception is raised.

See also: `ECompressionError` ([183](#)), `TCompressionStream.Write` ([185](#))

19.7.7 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` (183) exception is raised.

See also: `TCompressionStream.Read` (184), `TCompressionStream.Seek` (185)

19.7.8 TCompressionStream.Seek

Synopsis: Overrides seek to raise an exception.

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

Visibility: public

Description: The `Seek` method of `TStream` is overridden, and always raises an exception, because `TCompressionStream` is write-only, and cannot seek.

Errors: An `ECompressionError` (183) exception is raised.

See also: `ECompressionError` (183), `TCompressionStream.Read` (184), `TCompressionStream.Write` (185)

19.7.9 TCompressionStream.CompressionRate

Synopsis: Running compression rate of compression stream

Declaration: `Property CompressionRate : extended`

Visibility: public

Access: Read

Description: The `CompressionRate` is updated as more data is written to the stream and represents the ratio of outputted data versus written data.

See also: `TCompressionStream.Write` (185)

19.7.10 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.

19.8 TCustomZlibStream

19.8.1 Description

TCustomZlibStream serves as the ancestor class for the TCompressionStream (183) and TDecompressionStream (186) classes.

It introduces support for a progress handler, and stores the input or output stream.

19.8.2 Method overview

Page	Property	Description
186	Create	Create a new instance of TCustomZlibStream

19.8.3 TCustomZlibStream.Create

Synopsis: Create a new instance of TCustomZlibStream

Declaration: constructor Create(Strm: 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 (183), TDecompressionStream (186)

19.9 TDecompressionStream

19.9.1 Description

TDecompressionStream performs the inverse operation of TCompressionStream (183). A read operation reads data from an input stream and decompresses (inflates) the data it as it goes along.

The decompression stream reads it's compressed data from a stream with deflated data. This data can be created e.g. with a TCompressionStream (183) compression stream.

19.9.2 Method overview

Page	Property	Description
187	Create	Creates a new instance of the TDecompressionStream stream
187	Destroy	Destroys the TDecompressionStream instance
187	Read	Read data from the compressed stream
188	Seek	Move stream position to a certain location in the stream.
187	Write	Write data to the stream

19.9.3 Property overview

Page	Property	Access	Description
188	OnProgress		Progress handler

19.9.4 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: `TDecompressionStream.Destroy` ([187](#))

19.9.5 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: `TDecompressionStream.Create` ([187](#))

19.9.6 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` ([183](#)) exception is raised.

See also: `TCompressionStream.Write` ([185](#))

19.9.7 TDecompressionStream.Write

Synopsis: Write data to the stream

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

Visibility: `public`

Description: `Write` will raise a `EDeCompressionError` ([183](#)) exception, because the `TDecompressionStream` class is read-only.

Errors: An `EDeCompressionError` ([183](#)) exception is always raised.

See also: `TDecompressionStream.Read` ([187](#)), `EDeCompressionError` ([183](#))

19.9.8 TDecompressionStream.Seek

Synopsis: Move stream position to a certain location in the stream.

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

Visibility: `public`

Description: `Seek` overrides the standard `Seek` implementation. Normally, pipe streams `stderr` are not seekable. The `TDecompressionStream` 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 `EDecompressionError` (183) exception is raised if the stream does not allow the requested seek operation.

See also: `TDecompressionStream.Read` (187)

19.9.9 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.

19.10 TGZFileStream

19.10.1 Description

`TGZFileStream` can be used to read data from a gzip file, or to write data to a gzip file.

19.10.2 Method overview

Page	Property	Description
189	Create	Create a new instance of <code>TGZFileStream</code>
189	Destroy	Removes <code>TGZFileStream</code> instance
189	Read	Read data from the compressed file
190	Seek	Set the position in the compressed stream.
190	Write	Write data to be compressed

19.10.3 TGZFileStream.Create

Synopsis: Create a new instance of `TGZFileStream`

Declaration: constructor `Create(FileName: String; 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` (189) 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` (190) method will be stored in the file in compressed (deflated) form.

Errors: If the file is not found, an `EZlibError` (183) exception is raised.

See also: `TGZFileStream.Destroy` (189), `TGZOpenMode` (182)

19.10.4 TGZFileStream.Destroy

Synopsis: Removes `TGZFileStream` instance

Declaration: destructor `Destroy`; Override

Visibility: public

Description: `Destroy` closes the file and releases the `TGZFileStream` instance from memory.

See also: `TGZFileStream.Create` (189)

19.10.5 TGZFileStream.Read

Synopsis: Read data from the compressed file

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

Visibility: public

Description: `Read` overrides the `Read` method of `TStream` to read the data from the compressed file. The `Buffer` parameter indicates where the read data should be stored. The `Count` parameter specifies the number of bytes (*uncompressed*) that should be read from the compressed file. Note that it is not possible to read from the stream if it was opened in write mode.

The function returns the number of uncompressed bytes actually read.

Errors: If `Buffer` points to an invalid location, or does not have enough room for `Count` bytes, an exception will be raised.

See also: `TGZFileStream.Create` (189), `TGZFileStream.Write` (190), `TGZFileStream.Seek` (190)

19.10.6 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` ([183](#)) exception is raised.

See also: `TGZFileStream.Create` ([189](#)), `TGZFileStream.Read` ([189](#)), `TGZFileStream.Seek` ([190](#))

19.10.7 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` ([188](#)) for a list of possibilities.

Errors: In case an impossible combination is asked, an `EZlibError` ([183](#)) exception is raised.

See also: `TDecompressionStream.Seek` ([188](#))