PureBasic Reference Manual
4.50 (Windows)

http://www.purebasic.com/

June 8, 2010

Contents

I General 27

1 Introduction 28

2 Terms And Conditions 29

3 System requirements 30

4 Installation 31

5 Order 32

6 Contact 34

7 Acknowledgements 36

II The PureBasic Editor 38

8 Getting Started 39

9 Working with source files 41

10 Editing features 43

11 Managing projects 49

12 Compiling your programs 55

13 Using the debugger 65

14 Included debugging tools 73

15 Using the buildin Tools 82

16 Using external tools 87

17 Getting Help 92

18 Customizing the IDE 95

19 Commandline options for the IDE 112

III Language Reference 113

20 Working with different number bases 114

21 Break : Continue 119

22 Using the command line compiler 121

23 Compiler Directives 123

24 Compiler Functions 127
25 Data 132
26 Debugger keywords in PureBasic 135
27 Define 137
28 Dim 139
29 Building a DLL 141
30 Enumerations 143
31 For : Next 145
32 ForEach : Next 147
33 General Rules 149
34 Global 152
35 Gosub : Return 154
36 Handles and Numbers 156
37 If : Else : EndIf 158
38 Import : EndImport 159
39 Includes Functions 161
40 Inline x86 ASM 163
41 Interfaces 165
42 Macros 167
43 Pointers and memory access 169
44 NewList 173
45 NewMap 175
46 Others Commands 177
47 Procedures 179
48 Protected 182
49 Prototypes 184
50 Pseudotypes 186
51 PureBasic objects 188
52 Repeat : Until 190
53 Select : EndSelect 191
54 Using several PureBasic versions on Windows 193
55 Shared 194

56 Static 195

57 Structures 197

58 Threaded 201

59 Unicode 202

60 Variables and Types 204

61 We start programming... 213

62 While : Wend 216

63 Windows Message Handling 217

64 With : EndWith 219

IV Library Reference 221

65 2DDrawing 222
65.1 BackColor . . . 222
65.2 Box . . . 222
65.3 RoundBox . . . 223
65.4 Circle . . . 224
65.5 DrawImage . . . 225
65.6 DrawAlphaImage . . . 225
65.7 DrawingBuffer . . . 226
65.8 DrawingBufferPitch . . . 226
65.9 DrawingBufferPixelFormat . . . 226
65.10 DrawingFont . . . 227
65.11 DrawingMode . . . 227
65.12 DrawRotatedText . . . 229
65.13 FillArea . . . 230
65.14 GrabDrawingImage . . . 231
65.15 StartDrawing . . . 231
65.16 DrawText . . . 232
65.17 Ellipse . . . 233
65.18 FrontColor . . . 234
65.19 Line . . . 234
65.20 LineXY . . . 235
65.21 Plot . . . 235
65.22 Point . . . 236
65.23 StopDrawing . . . 236
65.24 TextHeight . . . 237
65.25 TextWidth . . . 237
65.26 OutputDepth . . . 237
65.27 OutputWidth . . . 237
65.28 OutputHeight . . . 238
65.29 CustomFilterCallback . . . 238
65.30 GradientColor . . . 239
65.31 ResetGradientColors . . . 240
65.32 LinearGradient . . . 241
65.33 CircularGradient . . . 242
65.34 EllipticalGradient . . . 243
65.35 BoxedGradient . . . 244
65.36 ConicalGradient . . . 245
65.37 CustomGradient . . . 246

66 Array 248

66.1 ArraySize . . . 248
66.2 CopyArray . . . 249
66.3 FreeArray . . . 249

67 AudioCD **250**
67.1 AudioCDLength . . . 250
67.2 AudioCDName . . . 250
67.3 AudioCDTrackLength . . . 251
67.4 AudioCDStatus . . . 251
67.5 AudioCDTracks . . . 251
67.6 AudioCDTrackSeconds . . . 251
67.7 EjectAudioCD . . . 252
67.8 InitAudioCD . . . 252
67.9 PauseAudioCD . . . 252
67.10 PlayAudioCD . . . 252
67.11 ResumeAudioCD . . . 253
67.12 StopAudioCD . . . 253
67.13 UseAudioCD . . . 253

68 Billboard **254**
68.1 AddBillboard . . . 254
68.2 BillboardGroupID . . . 255
68.3 BillboardGroupLocate . . . 255
68.4 BillboardGroupMaterial . . . 255
68.5 BillboardGroupX . . . 256
68.6 BillboardGroupY . . . 256
68.7 BillboardGroupZ . . . 257
68.8 BillboardHeight . . . 257
68.9 BillboardLocate . . . 257
68.10 BillboardWidth . . . 258
68.11 BillboardX . . . 258
68.12 BillboardY . . . 259
68.13 BillboardZ . . . 259
68.14 ClearBillboards . . . 260
68.15 CountBillboards . . . 260
68.16 CreateBillboardGroup . . . 261
68.17 FreeBillboardGroup . . . 261
68.18 HideBillboardGroup . . . 262
68.19 IsBillboardGroup . . . 262
68.20 MoveBillboard . . . 263
68.21 MoveBillboardGroup . . . 263
68.22 RemoveBillboard . . . 264
68.23 ResizeBillboard . . . 264
68.24 RotateBillboardGroup . . . 265

69 Camera **266**
69.1 CameraBackColor . . . 266
69.2 CameraFOV . . . 267
69.3 CameraID . . . 267
69.4 CameraLocate . . . 267
69.5 CameraLookAt . . . 268
69.6 CameraProjectionMode . . . 268
69.7 CameraProjectionX . . . 269
69.8 CameraProjectionY . . . 269
69.9 CameraRange . . . 269
69.10 CameraRenderMode . . . 270
69.11 CameraX . . . 271
69.12 CameraY . . . 271
69.13 CameraZ . . . 271

69.14 CreateCamera . . . 272
69.15 FreeCamera . . . 273
69.16 IsCamera . . . 273
69.17 MoveCamera . . . 274
69.18 RotateCamera . . . 274

70 Cipher **275**
70.1 AddCipherBuffer . . . 275
70.2 AESEncoder . . . 275
70.3 AESDecoder . . . 277
70.4 DESFingerprint . . . 277
70.5 ExamineMD5Fingerprint . . . 278
70.6 ExamineSHA1Fingerprint . . . 279
70.7 FinishCipher . . . 279
70.8 NextFingerprint . . . 280
70.9 FinishFingerprint . . . 280
70.10 IsFingerprint . . . 280
70.11 MD5FileFingerprint . . . 281
70.12 MD5Fingerprint . . . 281
70.13 SHA1Fingerprint . . . 282
70.14 SHA1FileFingerprint . . . 282
70.15 CRC32Fingerprint . . . 283
70.16 CRC32FileFingerprint . . . 283
70.17 Base64Decoder . . . 284
70.18 Base64Encoder . . . 284
70.19 StartAESCipher . . . 285
70.20 OpenCryptRandom . . . 285
70.21 CloseCryptRandom . . . 286
70.22 CryptRandom . . . 286
70.23 CryptRandomData . . . 286

71 Clipboard **288**
71.1 ClearClipboard . . . 288
71.2 GetClipboardImage . . . 289
71.3 GetClipboardText . . . 289
71.4 SetClipboardImage . . . 289
71.5 SetClipboardText . . . 290

72 Console **291**
72.1 ClearConsole . . . 291
72.2 CloseConsole . . . 292
72.3 ConsoleError . . . 292
72.4 ConsoleTitle . . . 292
72.5 ConsoleColor . . . 293
72.6 EnableGraphicalConsole . . . 294
72.7 Inkey . . . 295
72.8 Input . . . 296
72.9 ConsoleLocate . . . 296
72.10 ConsoleCursor . . . 297
72.11 Print . . . 298
72.12 PrintN . . . 298
72.13 OpenConsole . . . 299
72.14 ReadConsoleData . . . 299
72.15 RawKey . . . 300
72.16 WriteConsoleData . . . 301

73 Database **303**
73.1 CloseDatabase . . . 303
73.2 DatabaseColumns . . . 303
73.3 DatabaseColumnName . . . 304

73.4 DatabaseColumnSize . . . 304
73.5 DatabaseColumnType . . . 304
73.6 DatabaseDriverDescription . . . 305
73.7 DatabaseDriverName . . . 305
73.8 DatabaseError . . . 305
73.9 DatabaseID . . . 306
73.10 DatabaseQuery . . . 306
73.11 DatabaseUpdate . . . 306
73.12 ExamineDatabaseDrivers . . . 307
73.13 FinishDatabaseQuery . . . 307
73.14 FirstDatabaseRow . . . 308
73.15 GetDatabaseBlob . . . 308
73.16 GetDatabaseDouble . . . 308
73.17 GetDatabaseFloat . . . 309
73.18 GetDatabaseLong . . . 309
73.19 GetDatabaseQuad . . . 309
73.20 GetDatabaseString . . . 310
73.21 IsDatabase . . . 310
73.22 NextDatabaseDriver . . . 310
73.23 NextDatabaseRow . . . 310
73.24 OpenDatabase . . . 311
73.25 OpenDatabaseRequester . . . 311
73.26 PreviousDatabaseRow . . . 311
73.27 SetDatabaseBlob . . . 312
73.28 UsePostgreSQLDatabase . . . 313
73.29 UseSQLiteDatabase . . . 313
73.30 UseODBCDatabase . . . 314

74 Date **315**
74.1 AddDate . . . 315
74.2 Date . . . 316
74.3 Day . . . 316
74.4 DayOfWeek . . . 316
74.5 DayOfYear . . . 317
74.6 Month . . . 317
74.7 Year . . . 317
74.8 Hour . . . 318
74.9 Minute . . . 318
74.10 Second . . . 318
74.11 FormatDate . . . 318
74.12 ParseDate . . . 319

75 Debugger **321**
75.1 ShowDebugOutput . . . 321
75.2 ClearDebugOutput . . . 321
75.3 SaveDebugOutput . . . 322
75.4 ShowProfiler . . . 322
75.5 ResetProfiler . . . 322
75.6 StartProfiler . . . 323
75.7 StopProfiler . . . 323
75.8 ShowMemoryViewer . . . 323
75.9 ShowLibraryViewer . . . 324
75.10 ShowWatchlist . . . 324
75.11 ShowVariableViewer . . . 324
75.12 ShowCallstack . . . 325
75.13 ShowAssemblyViewer . . . 325
75.14 PurifierGranularity . . . 325

76 Desktop **326**

76.1 ExamineDesktops . . . 326
76.2 DesktopDepth . . . 326
76.3 DesktopFrequency . . . 327
76.4 DesktopHeight . . . 327
76.5 DesktopX . . . 327
76.6 DesktopY . . . 328
76.7 DesktopMouseX . . . 328
76.8 DesktopMouseY . . . 328
76.9 DesktopName . . . 329
76.10 DesktopWidth . . . 329

77 DragDrop **330**
77.1 DragText . . . 330
77.2 DragImage . . . 331
77.3 DragFiles . . . 332
77.4 DragPrivate . . . 333
77.5 DragOSFormats . . . 334
77.6 EnableGadgetDrop . . . 335
77.7 EnableWindowDrop . . . 336
77.8 EventDropAction . . . 337
77.9 EventDropType . . . 337
77.10 EventDropText . . . 338
77.11 EventDropImage . . . 338
77.12 EventDropFiles . . . 339
77.13 EventDropPrivate . . . 339
77.14 EventDropBuffer . . . 339
77.15 EventDropSize . . . 340
77.16 EventDropX . . . 340
77.17 EventDropY . . . 340
77.18 SetDragCallback . . . 340
77.19 SetDropCallback . . . 341

78 Engine3D **343**
78.1 Add3DArchive . . . 343
78.2 AmbientColor . . . 344
78.3 AntialiasingMode . . . 344
78.4 CheckEntityCollision . . . 344
78.5 CountRenderedTriangles . . . 344
78.6 Engine3DFrameRate . . . 345
78.7 EnableWorldCollisions . . . 345
78.8 EnableWorldPhysics . . . 345
78.9 ExamineWorldCollisions . . . 345
78.10 FirstWorldCollisionEntity . . . 346
78.11 Fog . . . 346
78.12 InitEngine3D . . . 346
78.13 InputEvent3D . . . 346
78.14 LoadWorld . . . 347
78.15 MousePick . . . 347
78.16 PointPick . . . 347
78.17 PickX . . . 348
78.18 PickY . . . 348
78.19 PickZ . . . 348
78.20 ShowGUI . . . 348
78.21 SetGUITheme3D . . . 348
78.22 NextWorldCollision . . . 349
78.23 Parse3DScripts . . . 349
78.24 RenderWorld . . . 349
78.25 SecondWorldCollisionEntity . . . 350
78.26 SkyBox . . . 350

78.27 SkyDome . . . 350
78.28 CreateWater . . . 350
78.29 Sun . . . 351
78.30 WorldShadows . . . 351
78.31 WorldGravity . . . 352
78.32 WorldDebug . . . 352

79 Entity **353**
79.1 AnimateEntity . . . 353
79.2 CopyEntity . . . 353
79.3 CreateEntity . . . 353
79.4 EntityID . . . 354
79.5 EntityLocate . . . 354
79.6 EntityLookAt . . . 354
79.7 EntityMaterial . . . 354
79.8 EntityPhysicBody . . . 355
79.9 EntityRenderMode . . . 355
79.10 EntityX . . . 356
79.11 EntityY . . . 356
79.12 EntityZ . . . 356
79.13 FreeEntity . . . 356
79.14 GetEntityAnimationTime . . . 356
79.15 GetEntityFriction . . . 357
79.16 GetEntityMass . . . 357
79.17 EntityAnimationLength . . . 357
79.18 HideEntity . . . 357
79.19 IsEntity . . . 358
79.20 MoveEntity . . . 358
79.21 ResizeEntity . . . 358
79.22 RotateEntity . . . 358
79.23 ScaleEntity . . . 359
79.24 SetEntityAnimationTime . . . 359
79.25 SetEntityMass . . . 359
79.26 SetEntityFriction . . . 359

80 File **360**
80.1 CloseFile . . . 360
80.2 CreateFile . . . 360
80.3 Eof . . . 361
80.4 FileBuffersSize . . . 361
80.5 FileID . . . 362
80.6 FileSeek . . . 362
80.7 FlushFileBuffers . . . 362
80.8 IsFile . . . 363
80.9 Loc . . . 363
80.10 Lof . . . 363
80.11 OpenFile . . . 364
80.12 TruncateFile . . . 364
80.13 ReadAsciiCharacter . . . 364
80.14 ReadByte . . . 365
80.15 ReadCharacter . . . 365
80.16 ReadDouble . . . 365
80.17 ReadFile . . . 365
80.18 ReadFloat . . . 366
80.19 ReadInteger . . . 366
80.20 ReadLong . . . 366
80.21 ReadQuad . . . 367
80.22 ReadData . . . 367
80.23 ReadString . . . 367

80.24 ReadStringFormat . . . 368
80.25 ReadUnicodeCharacter . . . 368
80.26 ReadWord . . . 368
80.27 WriteAsciiCharacter . . . 369
80.28 WriteByte . . . 369
80.29 WriteCharacter . . . 369
80.30 WriteDouble . . . 369
80.31 WriteFloat . . . 370
80.32 WriteInteger . . . 370
80.33 WriteLong . . . 370
80.34 WriteData . . . 370
80.35 WriteQuad . . . 371
80.36 WriteString . . . 371
80.37 WriteStringFormat . . . 372
80.38 WriteStringN . . . 372
80.39 WriteUnicodeCharacter . . . 373
80.40 WriteWord . . . 373

81 FileSystem **374**
81.1 CopyDirectory . . . 374
81.2 CopyFile . . . 374
81.3 CreateDirectory . . . 375
81.4 DeleteDirectory . . . 375
81.5 DeleteFile . . . 375
81.6 DirectoryEntryAttributes . . . 375
81.7 DirectoryEntryDate . . . 376
81.8 DirectoryEntryName . . . 377
81.9 DirectoryEntryType . . . 377
81.10 DirectoryEntrySize . . . 377
81.11 ExamineDirectory . . . 377
81.12 FinishDirectory . . . 378
81.13 GetExtensionPart . . . 378
81.14 GetFilePart . . . 378
81.15 GetPathPart . . . 379
81.16 IsDirectory . . . 379
81.17 CheckFilename . . . 379
81.18 FileSize . . . 379
81.19 GetCurrentDirectory . . . 380
81.20 GetHomeDirectory . . . 380
81.21 GetTemporaryDirectory . . . 380
81.22 GetFileDate . . . 380
81.23 GetFileAttributes . . . 381
81.24 NextDirectoryEntry . . . 382
81.25 RenameFile . . . 382
81.26 SetFileDate . . . 382
81.27 SetFileAttributes . . . 383
81.28 SetCurrentDirectory . . . 383

82 Font **384**
82.1 FreeFont . . . 384
82.2 FontID . . . 384
82.3 IsFont . . . 385
82.4 LoadFont . . . 385

83 Ftp **387**
83.1 AbortFTPFile . . . 387
83.2 CheckFTPConnection . . . 387
83.3 CloseFTP . . . 387
83.4 CreateFTPDirectory . . . 388

83.5 DeleteFTPDirectory . . . 388
83.6 DeleteFTPFile . . . 388
83.7 ExamineFTPDirectory . . . 388
83.8 GetFTPDirectory . . . 389
83.9 FinishFTPDirectory . . . 389
83.10 FTPDirectoryEntryAttributes . . . 389
83.11 FTPDirectoryEntryDate . . . 390
83.12 FTPDirectoryEntryName . . . 390
83.13 FTPDirectoryEntryType . . . 390
83.14 FTPDirectoryEntryRaw . . . 391
83.15 FTPDirectoryEntrySize . . . 391
83.16 FTPProgress . . . 391
83.17 IsFtp . . . 392
83.18 NextFTPDirectoryEntry . . . 392
83.19 OpenFTP . . . 392
83.20 ReceiveFTPFile . . . 393
83.21 RenameFTPFile . . . 393
83.22 SendFTPFile . . . 393
83.23 SetFTPDirectory . . . 393

84 Gadget **395**
84.1 AddGadgetColumn . . . 395
84.2 AddGadgetItem . . . 396
84.3 ButtonImageGadget . . . 397
84.4 ButtonGadget . . . 398
84.5 CalendarGadget . . . 399
84.6 CheckBoxGadget . . . 400
84.7 ClearGadgetItems . . . 401
84.8 CloseGadgetList . . . 402
84.9 ComboBoxGadget . . . 402
84.10 ContainerGadget . . . 403
84.11 CountGadgetItems . . . 404
84.12 DateGadget . . . 405
84.13 DisableGadget . . . 406
84.14 EditorGadget . . . 406
84.15 ExplorerComboGadget . . . 408
84.16 ExplorerListGadget . . . 408
84.17 ExplorerTreeGadget . . . 410
84.18 Frame3DGadget . . . 412
84.19 FreeGadget . . . 413
84.20 GadgetID . . . 413
84.21 GadgetItemID . . . 413
84.22 GadgetToolTip . . . 414
84.23 GadgetX . . . 414
84.24 GadgetY . . . 415
84.25 GadgetHeight . . . 415
84.26 GadgetType . . . 415
84.27 GadgetWidth . . . 416
84.28 GetActiveGadget . . . 417
84.29 GetGadgetAttribute . . . 417
84.30 GetGadgetColor . . . 418
84.31 GetGadgetData . . . 418
84.32 GetGadgetFont . . . 419
84.33 GetGadgetItemAttribute . . . 419
84.34 GetGadgetItemColor . . . 419
84.35 GetGadgetItemData . . . 420
84.36 GetGadgetState . . . 420
84.37 GetGadgetItemText . . . 421
84.38 GetGadgetItemState . . . 421

84.39 GetGadgetText . . . 422
84.40 HideGadget . . . 423
84.41 HyperLinkGadget . . . 423
84.42 ImageGadget . . . 424
84.43 IPAddressGadget . . . 425
84.44 IsGadget . . . 426
84.45 ListIconGadget . . . 426
84.46 ListViewGadget . . . 429
84.47 MDIGadget . . . 430
84.48 OpenGadgetList . . . 432
84.49 OptionGadget . . . 432
84.50 PanelGadget . . . 433
84.51 ProgressBarGadget . . . 434
84.52 RemoveGadgetColumn . . . 435
84.53 RemoveGadgetItem . . . 436
84.54 ResizeGadget . . . 436
84.55 ScrollBarGadget . . . 437
84.56 ScrollAreaGadget . . . 438
84.57 SetActiveGadget . . . 439
84.58 SetGadgetAttribute . . . 440
84.59 SetGadgetColor . . . 440
84.60 SetGadgetData . . . 442
84.61 SetGadgetFont . . . 443
84.62 SetGadgetItemAttribute . . . 443
84.63 SetGadgetItemColor . . . 444
84.64 SetGadgetItemData . . . 445
84.65 SetGadgetItemState . . . 446
84.66 SetGadgetItemText . . . 446
84.67 SetGadgetState . . . 447
84.68 SetGadgetText . . . 448
84.69 ShortcutGadget . . . 448
84.70 SpinGadget . . . 449
84.71 SplitterGadget . . . 450
84.72 StringGadget . . . 452
84.73 TextGadget . . . 453
84.74 TrackBarGadget . . . 454
84.75 TreeGadget . . . 455
84.76 UseGadgetList . . . 457
84.77 WebGadget . . . 458
84.78 WebGadgetPath . . . 461

85 Gadget3D **462**
85.1 AddGadgetItem3D . . . 462
85.2 ButtonGadget3D . . . 462
85.3 CheckBoxGadget3D . . . 463
85.4 ClearGadgetItems3D . . . 463
85.5 CloseGadgetList3D . . . 464
85.6 ComboBoxGadget3D . . . 464
85.7 ContainerGadget3D . . . 464
85.8 CountGadgetItems3D . . . 465
85.9 DisableGadget3D . . . 465
85.10 EditorGadget3D . . . 465
85.11 Frame3DGadget3D . . . 466
85.12 FreeGadget3D . . . 466
85.13 GadgetID3D . . . 466
85.14 GadgetToolTip3D . . . 467
85.15 GadgetX3D . . . 467
85.16 GadgetY3D . . . 467
85.17 GadgetHeight3D . . . 467

85.18 GadgetType3D . . . 468
85.19 GadgetWidth3D . . . 468
85.20 GetActiveGadget3D . . . 469
85.21 GetGadgetAttribute3D . . . 469
85.22 GetGadgetData3D . . . 469
85.23 GetGadgetItemData3D . . . 469
85.24 GetGadgetState3D . . . 470
85.25 GetGadgetItemText3D . . . 470
85.26 GetGadgetItemState3D . . . 470
85.27 GetGadgetText3D . . . 471
85.28 HideGadget3D . . . 471
85.29 ImageGadget3D . . . 471
85.30 IsGadget3D . . . 472
85.31 ListViewGadget3D . . . 472
85.32 OpenGadgetList3D . . . 472
85.33 OptionGadget3D . . . 473
85.34 PanelGadget3D . . . 473
85.35 ProgressBarGadget3D . . . 473
85.36 RemoveGadgetItem3D . . . 474
85.37 ResizeGadget3D . . . 474
85.38 ScrollBarGadget3D . . . 475
85.39 ScrollAreaGadget3D . . . 475
85.40 SetActiveGadget3D . . . 476
85.41 SetGadgetAttribute3D . . . 476
85.42 SetGadgetData3D . . . 477
85.43 SetGadgetItemData3D . . . 477
85.44 SetGadgetItemState3D . . . 477
85.45 SetGadgetItemText3D . . . 477
85.46 SetGadgetState3D . . . 478
85.47 SetGadgetText3D . . . 478
85.48 SpinGadget3D . . . 478
85.49 StringGadget3D . . . 479
85.50 TextGadget3D . . . 480

86 Help **481**
86.1 CloseHelp . . . 481
86.2 OpenHelp . . . 481

87 Http **482**
87.1 GetHTTPHeader . . . 482
87.2 GetURLPart . . . 483
87.3 ReceiveHTTPFile . . . 483
87.4 URLDecoder . . . 484
87.5 URLEncoder . . . 484
87.6 SetURLPart . . . 485

88 Image **486**
88.1 CatchImage . . . 486
88.2 CopyImage . . . 487
88.3 CreateImage . . . 487
88.4 FreeImage . . . 488
88.5 GrabImage . . . 488
88.6 ImageDepth . . . 488
88.7 ImageHeight . . . 489
88.8 ImageID . . . 489
88.9 ImageOutput . . . 489
88.10 ImageWidth . . . 489
88.11 IsImage . . . 489
88.12 LoadImage . . . 490

88.13 ResizeImage . . . 490
88.14 SaveImage . . . 491

89 ImagePlugin **492**
89.1 UseJPEGImageDecoder . . . 492
89.2 UseJPEGImageEncoder . . . 492
89.3 UseJPEG2000ImageDecoder . . . 493
89.4 UseJPEG2000ImageEncoder . . . 493
89.5 UsePNGImageDecoder . . . 493
89.6 UsePNGImageEncoder . . . 493
89.7 UseTGAImageDecoder . . . 494
89.8 UseTIFFImageDecoder . . . 494

90 Joystick **495**
90.1 InitJoystick . . . 495
90.2 ExamineJoystick . . . 495
90.3 JoystickAxisX . . . 495
90.4 JoystickAxisY . . . 496
90.5 JoystickButton . . . 496

91 Keyboard **497**
91.1 InitKeyboard . . . 497
91.2 ExamineKeyboard . . . 497
91.3 KeyboardInkey . . . 497
91.4 KeyboardMode . . . 498
91.5 KeyboardPushed . . . 499
91.6 KeyboardReleased . . . 501

92 Library **503**
92.1 CloseLibrary . . . 503
92.2 CallCFunction . . . 504
92.3 CallCFunctionFast . . . 504
92.4 CallFunction . . . 505
92.5 CallFunctionFast . . . 506
92.6 CountLibraryFunctions . . . 506
92.7 ExamineLibraryFunctions . . . 507
92.8 GetFunction . . . 507
92.9 GetFunctionEntry . . . 508
92.10 IsLibrary . . . 508
92.11 LibraryFunctionAddress . . . 509
92.12 LibraryFunctionName . . . 509
92.13 LibraryID . . . 510
92.14 NextLibraryFunction . . . 510
92.15 OpenLibrary . . . 510

93 Light **512**
93.1 CopyLight . . . 512
93.2 CreateLight . . . 512
93.3 FreeLight . . . 513
93.4 HideLight . . . 513
93.5 IsLight . . . 513
93.6 LightColor . . . 513
93.7 LightLocate . . . 513
93.8 LightSpecularColor . . . 514
93.9 MoveLight . . . 514

94 LinkedList **515**
94.1 AddElement . . . 515
94.2 ChangeCurrentElement . . . 516
94.3 ClearList . . . 517

94.4 CopyList . . . 518
94.5 FreeList . . . 518
94.6 ListSize . . . 518
94.7 DeleteElement . . . 519
94.8 FirstElement . . . 520
94.9 InsertElement . . . 521
94.10 LastElement . . . 522
94.11 ListIndex . . . 523
94.12 NextElement . . . 524
94.13 PreviousElement . . . 525
94.14 ResetList . . . 526
94.15 SelectElement . . . 526
94.16 SwapElements . . . 527

95 Mail **529**
95.1 AddMailAttachment . . . 529
95.2 AddMailAttachmentData . . . 531
95.3 AddMailRecipient . . . 532
95.4 CreateMail . . . 532
95.5 FreeMail . . . 533
95.6 GetMailAttribute . . . 533
95.7 GetMailBody . . . 534
95.8 IsMail . . . 534
95.9 MailProgress . . . 534
95.10 RemoveMailRecipient . . . 535
95.11 SendMail . . . 535
95.12 SetMailAttribute . . . 536
95.13 SetMailBody . . . 536

96 Map **538**
96.1 AddMapElement . . . 538
96.2 ClearMap . . . 539
96.3 CopyMap . . . 539
96.4 FreeMap . . . 540
96.5 MapSize . . . 540
96.6 DeleteMapElement . . . 541
96.7 FindMapElement . . . 541
96.8 MapKey . . . 542
96.9 NextMapElement . . . 542
96.10 ResetMap . . . 543

97 Material **544**
97.1 AddMaterialLayer . . . 544
97.2 CopyMaterial . . . 544
97.3 CountMaterialLayers . . . 545
97.4 CreateMaterial . . . 545
97.5 DisableMaterialLighting . . . 545
97.6 FreeMaterial . . . 545
97.7 IsMaterial . . . 546
97.8 MaterialAmbientColor . . . 546
97.9 MaterialBlendingMode . . . 546
97.10 MaterialDiffuseColor . . . 546
97.11 MaterialFilteringMode . . . 547
97.12 MaterialID . . . 547
97.13 MaterialShadingMode . . . 547
97.14 MaterialSpecularColor . . . 548
97.15 ScrollMaterial . . . 548
97.16 RemoveMaterialLayer . . . 548
97.17 RotateMaterial . . . 548

98 Math **549**
98.1 Abs . . . 549
98.2 ACos . . . 549
98.3 ACosH . . . 550
98.4 ASin . . . 550
98.5 ASinH . . . 550
98.6 ATan . . . 551
98.7 ATan2 . . . 551
98.8 ATanH . . . 552
98.9 Cos . . . 552
98.10 CosH . . . 552
98.11 Degree . . . 553
98.12 Exp . . . 553
98.13 Infinity . . . 553
98.14 Int . . . 553
98.15 IntQ . . . 554
98.16 IsInfinity . . . 554
98.17 IsNaN . . . 554
98.18 Pow . . . 555
98.19 Log . . . 555
98.20 Log10 . . . 555
98.21 NaN . . . 555
98.22 Radian . . . 556
98.23 Round . . . 556
98.24 Sign . . . 556
98.25 Sin . . . 557
98.26 SinH . . . 557
98.27 Sqr . . . 557
98.28 Tan . . . 558
98.29 TanH . . . 558

99 Memory **559**
99.1 AllocateMemory . . . 559
99.2 CompareMemory . . . 559
99.3 CompareMemoryString . . . 560
99.4 CopyMemory . . . 560
99.5 CopyMemoryString . . . 560
99.6 FillMemory . . . 561
99.7 FreeMemory . . . 561
99.8 MemorySize . . . 562
99.9 MemoryStringLength . . . 562
99.10 MoveMemory . . . 562
99.11 ReAllocateMemory . . . 562
99.12 PeekA . . . 563
99.13 PeekB . . . 563
99.14 PeekC . . . 564
99.15 PeekD . . . 564
99.16 PeekI . . . 564
99.17 PeekL . . . 564
99.18 PeekW . . . 564
99.19 PeekF . . . 565
99.20 PeekQ . . . 565
99.21 PeekS . . . 565
99.22 PeekU . . . 565
99.23 PokeA . . . 566
99.24 PokeB . . . 566
99.25 PokeC . . . 566
99.26 PokeD . . . 566
99.27 PokeI . . . 567

99.28 PokeL . . . 567
99.29 PokeQ . . . 567
99.30 PokeW . . . 567
99.31 PokeF . . . 567
99.32 PokeS . . . 568
99.33 PokeU . . . 568

100 Menu **569**
100.1 CloseSubMenu . . . 569
100.2 CreateMenu . . . 569
100.3 CreateImageMenu . . . 570
100.4 CreatePopupMenu . . . 571
100.5 CreatePopupImageMenu . . . 572
100.6 DisplayPopupMenu . . . 573
100.7 DisableMenuItem . . . 573
100.8 FreeMenu . . . 574
100.9 GetMenuItemState . . . 574
100.10 GetMenuItemText . . . 575
100.11 GetMenuTitleText . . . 575
100.12 HideMenu . . . 575
100.13 IsMenu . . . 575
100.14 MenuBar . . . 576
100.15 MenuHeight . . . 576
100.16 MenuItem . . . 576
100.17 MenuID . . . 577
100.18 MenuTitle . . . 577
100.19 OpenSubMenu . . . 578
100.20 SetMenuItemState . . . 579
100.21 SetMenuItemText . . . 579
100.22 SetMenuTitleText . . . 579

101 Mesh **581**
101.1 CreateMesh . . . 581
101.2 CopyMesh . . . 581
101.3 FreeMesh . . . 582
101.4 IsMesh . . . 582
101.5 LoadMesh . . . 582
101.6 MeshID . . . 582
101.7 SetMeshData . . . 583

102 Misc **584**
102.1 Delay . . . 584
102.2 ElapsedMilliseconds . . . 584
102.3 OSVersion . . . 585
102.4 Random . . . 585
102.5 RandomData . . . 586
102.6 RandomSeed . . . 587
102.7 Red . . . 587
102.8 Green . . . 587
102.9 Blue . . . 587
102.10 Alpha . . . 588
102.11 RGB . . . 588
102.12 RGBA . . . 588
102.13 AlphaBlend . . . 588

103 Module **589**
103.1 CatchModule . . . 589
103.2 FreeModule . . . 590
103.3 GetModulePosition . . . 590
103.4 GetModuleRow . . . 590

103.5 IsModule . . . 590
103.6 LoadModule . . . 591
103.7 ModuleVolume . . . 591
103.8 PlayModule . . . 591
103.9 SetModulePosition . . . 591
103.10 StopModule . . . 592

104 Mouse **593**
104.1 InitMouse . . . 593
104.2 ExamineMouse . . . 593
104.3 MouseButton . . . 593
104.4 MouseDeltaX . . . 594
104.5 MouseDeltaY . . . 594
104.6 MouseLocate . . . 594
104.7 MouseWheel . . . 595
104.8 MouseX . . . 595
104.9 MouseY . . . 595
104.10 ReleaseMouse . . . 595

105 Movie **596**
105.1 FreeMovie . . . 596
105.2 InitMovie . . . 596
105.3 IsMovie . . . 596
105.4 LoadMovie . . . 597
105.5 MovieAudio . . . 597
105.6 MovieHeight . . . 597
105.7 MovieInfo . . . 598
105.8 MovieLength . . . 598
105.9 MovieSeek . . . 598
105.10 MovieStatus . . . 599
105.11 MovieWidth . . . 599
105.12 PauseMovie . . . 599
105.13 PlayMovie . . . 599
105.14 RenderMovieFrame . . . 600
105.15 ResizeMovie . . . 600
105.16 ResumeMovie . . . 600
105.17 StopMovie . . . 600

106 Network **601**
106.1 CloseNetworkConnection . . . 601
106.2 ConnectionID . . . 601
106.3 ServerID . . . 601
106.4 CloseNetworkServer . . . 602
106.5 CreateNetworkServer . . . 602
106.6 ExamineIPAddresses . . . 602
106.7 HostName . . . 603
106.8 InitNetwork . . . 603
106.9 IPString . . . 603
106.10 IPAddressField . . . 603
106.11 MakeIPAddress . . . 604
106.12 EventServer . . . 604
106.13 EventClient . . . 604
106.14 GetClientIP . . . 604
106.15 GetClientPort . . . 605
106.16 NetworkClientEvent . . . 605
106.17 NetworkServerEvent . . . 605
106.18 NextIPAddress . . . 606
106.19 OpenNetworkConnection . . . 606
106.20 ReceiveNetworkData . . . 606

106.21 ReceiveNetworkFile . . . 607
106.22 SendNetworkData . . . 607
106.23 SendNetworkFile . . . 607
106.24 SendNetworkString . . . 608

107 Node **609**
107.1 AttachNodeObject . . . 609
107.2 DetachNodeObject . . . 609
107.3 CreateNode . . . 610
107.4 NodeID . . . 610
107.5 NodeLocate . . . 610
107.6 NodeLookAt . . . 611
107.7 NodeX . . . 611
107.8 NodeY . . . 611
107.9 NodeZ . . . 611
107.10 FreeNode . . . 611
107.11 IsNode . . . 612
107.12 MoveNode . . . 612
107.13 RotateNode . . . 612

108 OnError **613**
108.1 OnErrorExit . . . 613
108.2 OnErrorCall . . . 614
108.3 OnErrorGoto . . . 614
108.4 OnErrorDefault . . . 615
108.5 ErrorCode . . . 616
108.6 ErrorMessage . . . 616
108.7 ErrorLine . . . 617
108.8 ErrorFile . . . 617
108.9 ErrorAddress . . . 617
108.10 ErrorTargetAddress . . . 618
108.11 ErrorRegister . . . 618
108.12 RaiseError . . . 619
108.13 ExamineAssembly . . . 620
108.14 NextInstruction . . . 620
108.15 InstructionAddress . . . 621
108.16 InstructionString . . . 621

109 Packer **622**
109.1 AddPackFile . . . 622
109.2 AddPackMemory . . . 622
109.3 ClosePack . . . 623
109.4 CreatePack . . . 623
109.5 NextPackFile . . . 624
109.6 PackerCallback . . . 624
109.7 OpenPack . . . 624
109.8 PackFileSize . . . 625
109.9 PackMemory . . . 625
109.10 UnpackMemory . . . 625

110 Palette **627**
110.1 CreatePalette . . . 627
110.2 DisplayPalette . . . 627
110.3 GetPaletteColor . . . 628
110.4 FreePalette . . . 628
110.5 InitPalette . . . 628
110.6 IsPalette . . . 629
110.7 LoadPalette . . . 629
110.8 SetPaletteColor . . . 629

111 Particle **630**
111.1 CreateParticleEmitter . . . 630
111.2 IsParticleEmitter . . . 630
111.3 ParticleEmitterID . . . 631
111.4 ParticleEmitterLocate . . . 631
111.5 ParticleEmitterX . . . 631
111.6 ParticleEmitterY . . . 631
111.7 ParticleEmitterZ . . . 631
111.8 ParticleEmissionRate . . . 632
111.9 ParticleMaterial . . . 632
111.10 ParticleTimeToLive . . . 632
111.11 ParticleVelocity . . . 632
111.12 ParticleSize . . . 632
111.13 ParticleColorRange . . . 633
111.14 ParticleColorFader . . . 633
111.15 FreeParticleEmitter . . . 633
111.16 HideParticleEmitter . . . 633
111.17 MoveParticleEmitter . . . 634
111.18 ParticleEmitterDirection . . . 634
111.19 ResizeParticleEmitter . . . 634

112 Preference **635**
112.1 ClosePreferences . . . 635
112.2 CreatePreferences . . . 635
112.3 ExaminePreferenceGroups . . . 636
112.4 ExaminePreferenceKeys . . . 636
112.5 NextPreferenceGroup . . . 636
112.6 NextPreferenceKey . . . 636
112.7 PreferenceGroupName . . . 637
112.8 PreferenceKeyName . . . 637
112.9 PreferenceKeyValue . . . 637
112.10 OpenPreferences . . . 637
112.11 PreferenceGroup . . . 638
112.12 PreferenceComment . . . 638
112.13 ReadPreferenceDouble . . . 638
112.14 ReadPreferenceFloat . . . 638
112.15 ReadPreferenceInteger . . . 639
112.16 ReadPreferenceLong . . . 639
112.17 ReadPreferenceQuad . . . 639
112.18 ReadPreferenceString . . . 639
112.19 RemovePreferenceGroup . . . 640
112.20 RemovePreferenceKey . . . 640
112.21 WritePreferenceFloat . . . 640
112.22 WritePreferenceDouble . . . 640
112.23 WritePreferenceInteger . . . 640
112.24 WritePreferenceLong . . . 641
112.25 WritePreferenceQuad . . . 641
112.26 WritePreferenceString . . . 641

113 Printer **642**
113.1 DefaultPrinter . . . 642
113.2 NewPrinterPage . . . 642
113.3 PrinterOutput . . . 643
113.4 PrintRequester . . . 643
113.5 StartPrinting . . . 643
113.6 StopPrinting . . . 643
113.7 PrinterPageWidth . . . 644
113.8 PrinterPageHeight . . . 644

114 Process 645
114.1 AvailableProgramOutput . . . 645
114.2 CloseProgram . . . 645
114.3 CountProgramParameters . . . 646
114.4 EnvironmentVariableName . . . 646
114.5 EnvironmentVariableValue . . . 646
114.6 ExamineEnvironmentVariables . . . 646
114.7 GetEnvironmentVariable . . . 647
114.8 IsProgram . . . 647
114.9 KillProgram . . . 647
114.10 NextEnvironmentVariable . . . 648
114.11 ProgramExitCode . . . 648
114.12 ProgramFilename . . . 648
114.13 ProgramID . . . 649
114.14 ProgramParameter . . . 649
114.15 ProgramRunning . . . 649
114.16 ReadProgramData . . . 650
114.17 ReadProgramError . . . 650
114.18 ReadProgramString . . . 650
114.19 RemoveEnvironmentVariable . . . 651
114.20 RunProgram . . . 651
114.21 SetEnvironmentVariable . . . 652
114.22 WaitProgram . . . 653
114.23 WriteProgramData . . . 653
114.24 WriteProgramString . . . 653
114.25 WriteProgramStringN . . . 654

115 RegularExpression 655
115.1 CreateRegularExpression . . . 655
115.2 ExtractRegularExpression . . . 656
115.3 FreeRegularExpression . . . 656
115.4 IsRegularExpression . . . 657
115.5 MatchRegularExpression . . . 657
115.6 ReplaceRegularExpression . . . 657
115.7 RegularExpressionError . . . 658

116 Requester 659
116.1 ColorRequester . . . 659
116.2 FontRequester . . . 660
116.3 InputRequester . . . 661
116.4 MessageRequester . . . 662
116.5 NextSelectedFilename . . . 663
116.6 OpenFileRequester . . . 663
116.7 PathRequester . . . 665
116.8 SaveFileRequester . . . 666
116.9 SelectedFilePattern . . . 667
116.10 SelectedFontColor . . . 668
116.11 SelectedFontName . . . 668
116.12 SelectedFontSize . . . 668
116.13 SelectedFontStyle . . . 669

117 Scintilla 670
117.1 InitScintilla . . . 670
117.2 ScintillaGadget . . . 671
117.3 ScintillaSendMessage . . . 672

118 SerialPort 673
118.1 AvailableSerialPortInput . . . 673
118.2 AvailableSerialPortOutput . . . 673
118.3 CloseSerialPort . . . 673

118.4 GetSerialPortStatus . . . 674
118.5 IsSerialPort . . . 674
118.6 SerialPortError . . . 674
118.7 SerialPortID . . . 675
118.8 OpenSerialPort . . . 675
118.9 ReadSerialPortData . . . 676
118.10 SerialPortTimeouts . . . 676
118.11 SetSerialPortStatus . . . 677
118.12 WriteSerialPortData . . . 677
118.13 WriteSerialPortString . . . 678

119 Sort **679**
119.1 SortArray . . . 679
119.2 SortList . . . 679
119.3 SortStructuredArray . . . 680
119.4 SortStructuredList . . . 681

120 Sound **683**
120.1 CatchSound . . . 683
120.2 FreeSound . . . 684
120.3 InitSound . . . 684
120.4 IsSound . . . 684
120.5 LoadSound . . . 684
120.6 PlaySound . . . 685
120.7 SoundFrequency . . . 685
120.8 SoundPan . . . 686
120.9 SoundVolume . . . 686
120.10 StopSound . . . 686

121 Sound3D **687**
121.1 FreeSound3D . . . 687
121.2 IsSound3D . . . 687
121.3 LoadSound3D . . . 687
121.4 PlaySound3D . . . 688
121.5 SoundVolume3D . . . 688
121.6 StopSound3D . . . 688
121.7 SoundID3D . . . 689
121.8 SoundRange3D . . . 689
121.9 SoundCone3D . . . 689

122 SoundPlugin **690**
122.1 UseFLACSoundDecoder . . . 690
122.2 UseOGGSoundDecoder . . . 690

123 Sprite **691**
123.1 CatchSprite . . . 691
123.2 ChangeAlphaIntensity . . . 692
123.3 ClipSprite . . . 692
123.4 CopySprite . . . 693
123.5 CreateSprite . . . 693
123.6 DisplayAlphaSprite . . . 694
123.7 DisplayRGBFilter . . . 694
123.8 DisplayShadowSprite . . . 695
123.9 DisplaySolidSprite . . . 695
123.10 DisplaySprite . . . 695
123.11 DisplayTranslucentSprite . . . 695
123.12 DisplayTransparentSprite . . . 696
123.13 FreeSprite . . . 696
123.14 GrabSprite . . . 696
123.15 InitSprite . . . 697

123.16 IsSprite . . . 697
123.17 LoadSprite . . . 697
123.18 SaveSprite . . . 698
123.19 SpriteCollision . . . 699
123.20 SpriteDepth . . . 699
123.21 SpriteHeight . . . 699
123.22 SpriteID . . . 699
123.23 SpritePixelCollision . . . 700
123.24 SpriteWidth . . . 700
123.25 StartSpecialFX . . . 700
123.26 StopSpecialFX . . . 701
123.27 AvailableScreenMemory . . . 701
123.28 ChangeGamma . . . 701
123.29 ClearScreen . . . 702
123.30 CloseScreen . . . 702
123.31 FlipBuffers . . . 702
123.32 IsScreenActive . . . 703
123.33 ScreenID . . . 703
123.34 SetFrameRate . . . 703
123.35 OpenScreen . . . 703
123.36 OpenWindowedScreen . . . 704
123.37 ScreenOutput . . . 707
123.38 SpriteOutput . . . 707
123.39 TransparentSpriteColor . . . 707
123.40 UseBuffer . . . 708
123.41 ExamineScreenModes . . . 708
123.42 NextScreenMode . . . 708
123.43 ScreenModeDepth . . . 709
123.44 ScreenModeHeight . . . 709
123.45 ScreenModeRefreshRate . . . 709
123.46 ScreenModeWidth . . . 709

124 Sprite3D **710**
124.1 CreateSprite3D . . . 710
124.2 FreeSprite3D . . . 710
124.3 DisplaySprite3D . . . 711
124.4 InitSprite3D . . . 711
124.5 IsSprite3D . . . 711
124.6 RotateSprite3D . . . 712
124.7 Sprite3DBlendingMode . . . 712
124.8 Sprite3DQuality . . . 712
124.9 Start3D . . . 713
124.10 Stop3D . . . 713
124.11 TransformSprite3D . . . 713
124.12 ZoomSprite3D . . . 714

125 StatusBar **715**
125.1 AddStatusBarField . . . 715
125.2 CreateStatusBar . . . 715
125.3 FreeStatusBar . . . 716
125.4 IsStatusBar . . . 716
125.5 StatusBarImage . . . 716
125.6 StatusBarID . . . 717
125.7 StatusBarText . . . 717
125.8 StatusBarProgress . . . 718
125.9 StatusBarHeight . . . 718

126 String **720**
126.1 Asc . . . 720

126.2 Bin . . . 720
126.3 Chr . . . 721
126.4 CountString . . . 721
126.5 FindString . . . 722
126.6 Hex . . . 722
126.7 InsertString . . . 723
126.8 LCase . . . 723
126.9 Left . . . 723
126.10 Len . . . 724
126.11 LSet . . . 724
126.12 LTrim . . . 725
126.13 Mid . . . 725
126.14 RemoveString . . . 725
126.15 ReplaceString . . . 726
126.16 Right . . . 726
126.17 RSet . . . 727
126.18 RTrim . . . 727
126.19 StringByteLength . . . 728
126.20 StringField . . . 728
126.21 StrF . . . 728
126.22 StrD . . . 729
126.23 Str . . . 729
126.24 StrU . . . 730
126.25 ReverseString . . . 730
126.26 Space . . . 731
126.27 Trim . . . 731
126.28 UCase . . . 731
126.29 ValD . . . 732
126.30 ValF . . . 732
126.31 Val . . . 732

127 SysTray **733**
127.1 AddSysTrayIcon . . . 733
127.2 ChangeSysTrayIcon . . . 733
127.3 IsSysTrayIcon . . . 734
127.4 SysTrayIconToolTip . . . 734
127.5 RemoveSysTrayIcon . . . 734

128 Terrain **735**
128.1 CreateTerrain . . . 735
128.2 TerrainHeight . . . 736

129 Texture **737**
129.1 CreateTexture . . . 737
129.2 FreeTexture . . . 737
129.3 IsTexture . . . 738
129.4 LoadTexture . . . 738
129.5 TextureID . . . 738
129.6 TextureHeight . . . 738
129.7 TextureWidth . . . 739

130 Thread **740**
130.1 IsThread . . . 740
130.2 ThreadID . . . 741
130.3 CreateMutex . . . 741
130.4 CreateThread . . . 742
130.5 FreeMutex . . . 743
130.6 KillThread . . . 743
130.7 LockMutex . . . 744
130.8 PauseThread . . . 745

130.9 ResumeThread . . . 745
130.10 ThreadPriority . . . 746
130.11 TryLockMutex . . . 747
130.12 UnlockMutex . . . 748
130.13 WaitThread . . . 748
130.14 CreateSemaphore . . . 749
130.15 FreeSemaphore . . . 751
130.16 SignalSemaphore . . . 751
130.17 WaitSemaphore . . . 751
130.18 TrySemaphore . . . 751

131 ToolBar **752**
131.1 CreateToolBar . . . 752
131.2 FreeToolBar . . . 753
131.3 DisableToolBarButton . . . 753
131.4 GetToolBarButtonState . . . 754
131.5 IsToolBar . . . 754
131.6 SetToolBarButtonState . . . 754
131.7 ToolBarHeight . . . 754
131.8 ToolBarImageButton . . . 755
131.9 ToolBarSeparator . . . 755
131.10 ToolBarStandardButton . . . 756
131.11 ToolBarToolTip . . . 757
131.12 ToolBarID . . . 758

132 Window **759**
132.1 AddKeyboardShortcut . . . 759
132.2 AddWindowTimer . . . 761
132.3 RemoveWindowTimer . . . 762
132.4 EventTimer . . . 762
132.5 CloseWindow . . . 763
132.6 DisableWindow . . . 763
132.7 EventGadget . . . 763
132.8 EventMenu . . . 763
132.9 EventType . . . 764
132.10 EventWindow . . . 765
132.11 GetActiveWindow . . . 765
132.12 GetWindowColor . . . 765
132.13 GetWindowState . . . 766
132.14 GetWindowTitle . . . 766
132.15 HideWindow . . . 766
132.16 IsWindow . . . 766
132.17 OpenWindow . . . 767
132.18 RemoveKeyboardShortcut . . . 768
132.19 ResizeWindow . . . 768
132.20 SetActiveWindow . . . 768
132.21 SetWindowCallback . . . 768
132.22 SetWindowColor . . . 770
132.23 SetWindowState . . . 770
132.24 SetWindowTitle . . . 770
132.25 SmartWindowRefresh . . . 771
132.26 StickyWindow . . . 771
132.27 WindowEvent . . . 771
132.28 WaitWindowEvent . . . 772
132.29 WindowBounds . . . 773
132.30 WindowHeight . . . 774
132.31 WindowID . . . 774
132.32 WindowWidth . . . 774
132.33 WindowX . . . 775

132.34 WindowY . . . 775
132.35 WindowMouseX . . . 775
132.36 WindowMouseY . . . 775
132.37 WindowOutput . . . 776

133 Window3D **777**
133.1 CloseWindow3D . . . 777
133.2 DisableWindow3D . . . 777
133.3 EventGadget3D . . . 778
133.4 EventType3D . . . 778
133.5 EventWindow3D . . . 778
133.6 GetActiveWindow3D . . . 778
133.7 GetWindowTitle3D . . . 779
133.8 HideWindow3D . . . 779
133.9 IsWindow3D . . . 779
133.10 OpenWindow3D . . . 779
133.11 ResizeWindow3D . . . 780
133.12 SetActiveWindow3D . . . 780
133.13 SetWindowTitle3D . . . 780
133.14 WindowEvent3D . . . 780
133.15 WindowHeight3D . . . 781
133.16 WindowID3D . . . 781
133.17 WindowWidth3D . . . 781
133.18 WindowX3D . . . 781
133.19 WindowY3D . . . 782

134 XML **783**
134.1 IsXML . . . 784
134.2 FreeXML . . . 784
134.3 CreateXML . . . 784
134.4 LoadXML . . . 785
134.5 CatchXML . . . 785
134.6 XMLStatus . . . 786
134.7 XMLError . . . 787
134.8 XMLErrorLine . . . 787
134.9 XMLErrorPosition . . . 788
134.10 SaveXML . . . 788
134.11 ExportXMLSize . . . 788
134.12 ExportXML . . . 789
134.13 FormatXML . . . 789
134.14 GetXMLEncoding . . . 790
134.15 SetXMLEncoding . . . 790
134.16 GetXMLStandalone . . . 790
134.17 SetXMLStandalone . . . 791
134.18 RootXMLNode . . . 791
134.19 MainXMLNode . . . 791
134.20 ChildXMLNode . . . 792
134.21 ParentXMLNode . . . 792
134.22 XMLChildCount . . . 792
134.23 NextXMLNode . . . 793
134.24 PreviousXMLNode . . . 793
134.25 XMLNodeFromPath . . . 793
134.26 XMLNodeFromID . . . 794
134.27 XMLNodeType . . . 794
134.28 GetXMLNodeText . . . 795
134.29 SetXMLNodeText . . . 795
134.30 GetXMLNodeOffset . . . 796
134.31 SetXMLNodeOffset . . . 796
134.32 GetXMLNodeName . . . 796

134.33 SetXMLNodeName . . . 796
134.34 XMLNodePath . . . 797
134.35 GetXMLAttribute . . . 797
134.36 SetXMLAttribute . . . 797
134.37 RemoveXMLAttribute . . . 797
134.38 ExamineXMLAttributes . . . 798
134.39 NextXMLAttribute . . . 798
134.40 XMLAttributeName . . . 798
134.41 XMLAttributeValue . . . 798
134.42 CreateXMLNode . . . 799
134.43 CopyXMLNode . . . 799
134.44 MoveXMLNode . . . 800
134.45 DeleteXMLNode . . . 800
134.46 ResolveXMLNodeName . . . 800
134.47 ResolveXMLAttributeName . . . 801

Part I

General

Chapter 1

Introduction

PureBasic is an "high-level" programming language based on established "BASIC" rules. It is mostly compatible with any other "BASIC" compiler, whether it's for the Amiga or PC format. Learning PureBasic is very easy! PureBasic has been created for beginners and experts alike. Compilation time is extremely fast. This software has been developed for the Windows operating system. We have put a lot of effort into its realization to produce a fast, reliable and system-friendly language.

The syntax is easy and the possibilities are huge with the "advanced" functions that have been added to this language like pointers, structures, procedures, dynamic linked lists and much more. For the experienced coder, there are no problems gaining access to any of the legal OS structures or Windows API objects.

PureBasic is a portable programming language which currently works on AmigaOS, Linux, MacOS X and Windows computer systems. This means that the same code can be compiled natively for the OS and use the full power of each. There are no bottlenecks like a virtual machine or a code translator, the generated code produces an optimized executable.

The main features of PureBasic

- x86, x64, 680x0 and PowerPC support
- Built-in arrays, dynamic linked lists, complex structures, pointers and variable definitions
- Supported types: Byte (8 bits), Word (16 bits), Long (32 bits), Quad (64 bits), Float (32 bits), Double (64 bits) and Characters
- User defined types (structures)
- Built-in string types (characters), including ascii and unicode
- Powerful macro support
- Constants, binary and hexadecimal numbers supported
- Expression reducer by grouping constants and numeric numbers together
- Standard arithmetic support in respect of sign priority and parenthesis: +, -, /, *, and, or, «, »
- Extremely fast compilation
- Procedure support for structured programming with local and global variables
- All Standard BASIC keywords: If-Else-EndIf, Repeat-Until, etc
- Specialized libraries to manipulate BMP pictures, windows, gadgets, DirectX, etc
- Specialized libraries are very optimized for maximum speed and compactness
- The Win32 API is fully supported as if they were BASIC keywords
- Inline Assembler
- Precompiled structures with constants files for extra-fast compilation
- Configurable CLI compiler
- Very high productivity, comprehensive keywords, online help
- System friendly, easy to install and easy to use

Chapter 2

Terms And Conditions

This program is provided "**AS IS**". Fantaisie Software are NOT responsible for any damage (or damages) attributed to PureBasic. You are warned that you use PureBasic at your own risk. No warranties are implied or given by Fantaisie Software or any representative.

The demo version of this program may be freely distributed provided all contents, of the original archive, remain intact. You may not modify, or change, the contents of the original archive without express written consent from Fantaisie Software.

PureBasic has an user-based license. This means you can install it on every computer you need but you can't share it between two or more people.

All components, libraries, and binaries are copyrighted by Fantaisie Software. The PureBasic license explicitly forbids the creation of DLLs whose primary function is to serve as a 'wrapper' for PureBasic functions.

Fantaisie Software reserves all rights to this program and all original archives and contents.

PureBasic uses the OGRE OpenSource 3D-Engine and its licence can be consulted here .
The full sources of the PureBasic-customized OGRE engine can be downloaded at http://www.purebasic.com/OgreSources.zip.

PureBasic uses the OnError library, written and (c) 2003 by Siegfried Rings and Sebastian Lackner.

PureBasic uses the Scintilla editing component and its related license can be consulted here .

The PureBasic XML library uses the expat XML parser. Its license can be viewed here .

The PureBasic RegularExpression library uses the PCRE library. Its license can be viewed here .

Chapter 3

System requirements

PureBasic will run on any Windows-based system (which includes: Windows 95/98/Me, Windows NT/2000 and Windows XP/7), AmigaOS classic (3.0 or above), Linux (kernel 2.2 or above) and MacOS X (10.2 (Jaguar) or above).

If there are any problems, please contact us.

Linux

Required packages to use PureBasic:

- sdl 1.2 devel (For games)
- gtk 1.2+ devel or/and gtk 2+ (For GUI programs)
- libstdc++ devel
- gcc correctly installed
- iodbc and iodbc-devel to be able to use the Database commands (see www.iodbc.org)
- libgtkembedmoz.so shoud be installed to have the WebGadget() working (comes with the Mozilla/Firefox package).
- xine and xine-devel for the Movie commands

PureBasic has been succesfully tested on the following configurations (it doesn't mean it only works on these):

- Redhat 7.0, 7.2, 7.3, 8.0 and 9.0
- Mandrake 6.1, 8.0, 10.0 and 10.1
- Suse 7.3, 8.0, 9.0, 10.2
- Debian
- Ubuntu

MacOS X

You need to install the Apple developer tools to be able to use PureBasic. The tools can be found either on one of the install CDs or on the Apple web site: http://developer.apple.com/

Chapter 4

Installation

To install PureBasic, just click on the install wizard, follow the steps, and then click on the PureBasic icon (found on the desktop or in the start-menu) to launch PureBasic.

To use the command line compiler, open a standard command line window (CMD) and look in the "Compilers\" subdirectory for PBCompiler.exe. It's a good idea to consider adding the "PureBasic\Compilers\" directory to the PATH environment variable to make the compiler accessible from any directory.

Important note: to avoid conflicts with existing PureBasic installations (maybe even with user-libraries), please install a new PureBasic version always in its own new folder. See also the chapter Using several PureBasic versions .

Chapter 5

Order

PureBasic is a low-cost programming language. In buying PureBasic you will ensure that development will go further and faster. For personal use (i.e.: not used by a commercial organization) the updates are unlimited, unlike most other software out there. This means than when you buy PureBasic you will get all future updates for free, on the web site. Better still, you get the three versions of PureBasic (Amiga, Linux and Windows) for the same price! For ease of ordering, you can safely use our secure online method. Thanks a lot for your support!

The demo-version of PureBasic is limited as shown below:

- No DLL can be created
- you can't use the whole external Win32 API support
- no development kit for external libraries
- maximum number of source lines: 800

Full version of PureBasic:

Price for full version: 79 Euros

Payment by cheque (France only, to 'Frederic Laboureur') :

```
Fantaisie Software
10, rue de Lausanne
67640 Fegersheim
France
```

If you live in Germany or German-speaking country and prefer paying to bank account you can also send your registration to German team member. In this case please send your order to following address:

```
André Beer
Siedlung 6
09548 Deutschneudorf
Germany
e-mail: andre@purebasic.com

Bank Account:
Deutsche Kreditbank
```

```
 Account 15920010 - Bank code 12030000
 (For transactions from EU countries: IBAN: DE03120300000015920010 -
BIC/Swift-Code: BYLADEM1001)
```

Online Ordering

Payment online (www.purebasic.com) is available and very secure.

Delivering of the full version

A full version will be sent via e-mail or snail mail on immediate receipt. Please indicate your preference. If you want to use this order form click here (or here when ordering at André) then print it. Please mail to the address shown above.

Chapter 6

Contact

Please send bug reports, suggestions, improvements, examples of source coding, or if you just want to talk to us, to any of the following addresses:

Frédéric 'AlphaSND' Laboureur

Fred 'AlphaSND' is the founder of Fantaisie Software and the main coder for PureBasic. All suggestions, bug reports, etc. should be sent to him at either address shown below:

s-mail :

Frédéric Laboureur
10, rue de Lausanne
67640 Fegersheim
France

e-mail : fred@purebasic.com

Richard Andersson

Richard is a very talented coder who has helped Fred to develop the strong and powerful assembler sub-libraries needed for PureBasic (mainly for the Amiga version). Need a new library? Get in touch with him at:

e-mail : richard@purebasic.com

André Beer

André is responsible for the complete German translation of the PureBasic manual and website. PureBasic can be ordered in Germany also directly at him.

Just write an email with your full address (for the registration) to him. If needed you can also get an invoice from him. For more details just take a look here.

e-mail : andre@purebasic.com

Benny 'Berikco' Sels

Benny is the main coder for the Visual Designer. The Visual Designer updates and history can be found at his website: http://www.purebasic.be. All suggestions,bug reports, etc. about the Visual Designer should be sent to him at the address shown below:

e-mail : berikco@purebasic.com

Chapter 7

Acknowledgements

We would like to thank the many people who have helped in this ambitious project. It would not have been possible without them !

- All the registered users: To support this software... Many Thanks !

Coders

- **Timo 'Fr34k' Harter**: For the IDE, Debugger, many commands and the great ideas. PureBasic wouldn't be the same without him !

- **Benny 'Berikco' Sels**: To provide an amazing first class Visual Designer for PureBasic.. That's great to have such kind of help !

- **Danilo Krahn**: To have done an huge work on the editor and on the core command set, without forget the tons of nice suggestions about code optimization and size reduction... Thanks a lot.

- **Sebastion 'PureFan' Lackner**: For the very nice OnError library, which adds professional error handling for final executables and several other commands !

- **Siegfried Rings**: Also for the very nice OnError library, as its a two men work :)

- **Marc Vitry**: To have kindly shared its SerialPort library sources which helped a lot for the native PureBasic library.

- **Stefan Moebius**: To have sent its DX9 subsystem sources, which have been the basis of the current DX9 subsystem for PureBasic.

Miscellaneous

- **Roger Beausoleil**: The first to believe in this project, and his invaluable help with the main design and layout of PureBasic.

- **Andre Beer**: To spend time to translate all the guides into German. Big thanks.

- **Francis G. Loch**: To have corrected all the mistakes in the English guides ! Thanks again.

- **Steffen Haeuser**: For giving his valuable time, and assistance. For his invaluable explanations regarding aspects of PPC coding, and his very useful tips.

- **Martin Konrad**: A fantastic bug reporter for the Amiga version. To have done the forthcoming Visual Environment for PureBasic and some nice games..

- **James L. Boyd**: To have found tons of bugs in the x86 version and gave us tons of work :). Keep up the bug hunt !

- **Les**: For editing the English Fantaisie Software Web site & and this guide. This looks much better!

- **NAsm team**: To have done an incredible x86 assembler used to develop the PureBasic libraries

- **Tomasz Grysztar**: For FAsm, an excellent optimized x86 assembler currently used by PureBasic. http://flatassembler.net/

- **Pelle Orinius**: For the very good linker and resource compiler (from PellesC) currently used by PureBasic, and the invaluable and instant help when integrating the new linker in the PureBasic package. Thank you !

- **Jacob Navia**: For the linker (from the Lcc32 package) which was used in previous PureBasic versions

- **Thomas Richter**: For creating "PoolMem." A patch that decreases the compilation time by a factor of 2-3! Many thanks for allowing it to be added to the main archive.

- **Frank Wille**: For allowing the use of your excellent assemblers: "pasm," and "PhxAss". For all your tips, and invaluable assistance, in their usage. To help us a lot about debugging PowerPC executables..

- **Rings, Paul, Franco and MrVain/Secretly**: For extensive bug reports and help us to have a compiler as reliable and bug free as possible.

- **David 'Tinman' McMinn**: To enhance greatly the help and give the opportunity to the world to really discover PureBasic !

- **Leigh**: To have created and hosted the first PureBasic forums during 2 years. It has been a great amount of work, especially when the community started to grow. Thanks again !

Part II

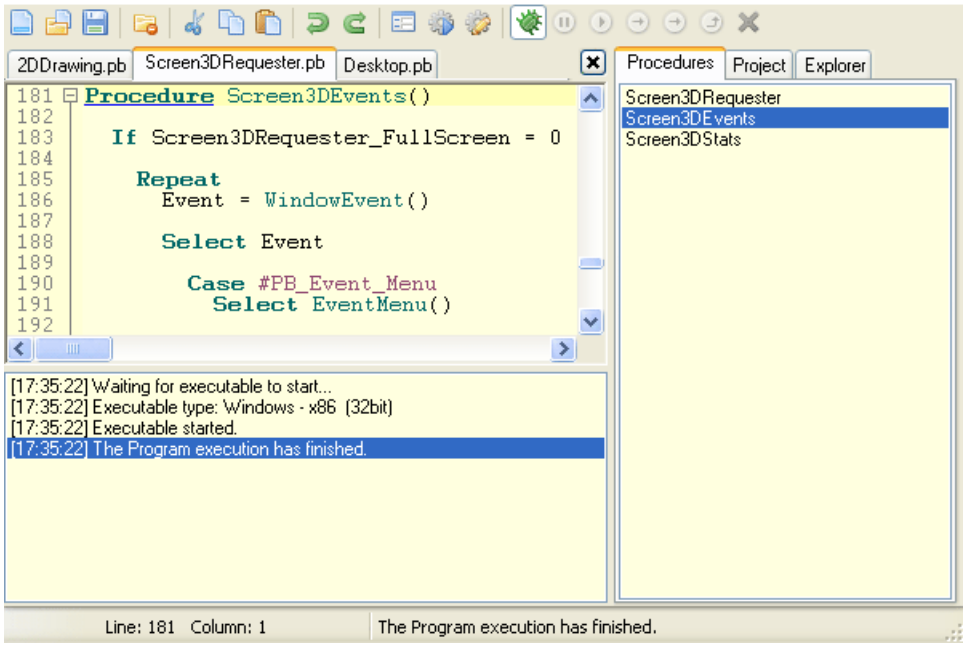
The PureBasic Editor

Chapter 8

Getting Started

The PureBasic IDE allows you to create and edit your PureBasic source codes, as well as run them, debug them and create the final executable. It has both an interface to the PureBasic Compiler , as well to the PureBasic Debugger .

The IDE main window contains of 3 major parts:



The code editing area (below the toolbar)
Here all the source codes are displayed. You can switch between them with the tabs located right above it.

The tools panel (on the right side by default)
Here you have several tools to make coding easier and increase productivity. The tools displayed here can be configured, and it can even be completely removed. See Customizing the IDE for more information.

The error log (located below the editing area)
In this area, the compiler errors and debugger messages are logged. It can be hidden/shown for each source code separately.

Other then that, there is the main menu and the toolbar. The toolbar simply provides shortcuts to menu features. It can be fully customized. To find out what each button does, move your mouse over it

and wait until a small Tooltip appears. It shows the corresponding menu command. The menu commands are explained in the other sections.

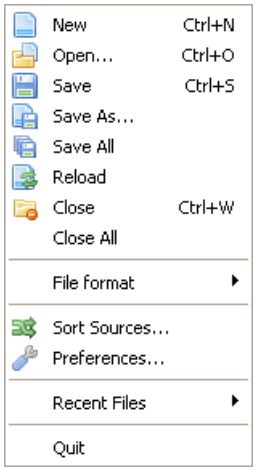
Chapter 9

Working with source files

The file menu allows you to do basic file operations like opening and saving source codes.

You can edit multiple source code files at the same time. You can switch between them using the panel located under the Toolbar. Also the shortcut keys Ctrl+Tab and Ctrl+Shift+Tab can be used to jump to the next or previous open source file, respectively.

Contents of the "File" menu:



New
Create a new empty source code file.

Open
Open an existing source code file for editing.

Any text file will be loaded into the source-editing field. You can also load binary files with the Open menu. These will be displayed in the internal File Viewer .

Save
Saves the currently active source to disk. If the file isn't saved yet, you will be prompted for a filename. Otherwise the code will be saved in the file it was saved in before.

Save As...
Save the currently active source to a different location than it was saved before. This prompts you for a new filename and leaves the old file (if any) untouched.

Save All

Saves all currently opened sources.

Reload
Reloads the currently active source code from disk. This discards any changes made that were not yet saved.

Close
Closes the currently active source code. If it was the only open code, the IDE will display a new empty file.

Close All
Closes all currently opened sources.

File format
In this submenu you can select the text encoding as well as the newline format which should be used when the currently active source code is saved to disk. The IDE can handle files in Ascii or UTF-8. The newline formats it can handle are Windows (CRLF), Linux/Unix (LF) and MacOSX (CR). The defaults for newly created source codes can be set in the preferences .

Sort Sources
When many files are open at the same time, you see the list of them on the panel under the Toolbar. This menu command can be used to change the order of the open source codes without the need to close/reopen any of them.

It will open a list of the open source code, where you can select one and move it up/down. The further up in the list a file is located, the further left it will be in the panel of open files.

Preferences
Here you can change all the settings that control the look & behaviour of the IDE. For a detailed description of that see Customizing the IDE .

Recent Files
Here you can see a list of the last accessed files. Selecting a file in this submenu will open it again.

Quit
This of course closes the IDE. You will be asked to save any non-saved source codes.

Chapter 10

Editing features

The PureBasic IDE acts like any other Text Editor when it comes to the basic editing features. The cursor keys as well as Page Up/Page Down, Home and End keys can be used to navigate through the code. Ctrl+Home navigates to the beginning of the file and Ctrl+End to the End.

The default shortcuts Ctrl+C (copy), Ctrl+X (cut) and Ctrl+V (paste) can be used for editing. The "Insert" key controls whether text is inserted or overwritten. The Delete key does a forward delete. Holding down the Shift key and using the arrow keys selects text.

Furthermore, the IDE has many extra editing features specific to programming or PureBasic.

Indentation:

When you press enter, the indentation (number of space/tab at the beginning of the line) of the current and next line will be automatically corrected depending on the keywords that exist on these lines. A "block mode" is also available where the new line simply gets the same indentation as the previous one. The details of this feature can be customized in the preferences .

Tab characters:

By default, the IDE does not insert a real tab when pressing the Tab key, as many programmers see it as a bad thing to use real tabs in source code.

It instead inserts two spaces. This behaviour can be changed in the Preferences. See Customizing the IDE for more information.

Special Tab behaviour:

When the Tab key is pressed while nothing or only a few characters are selected, the Tab key acts as mentioned above (inserting a number of spaces, or a real tab if configured that way).

However when one or more full lines are selected, the reaction is different. In that case at the beginning of each selected line, it will insert spaces or a tab (depending on the configuration). This increases the indentation of the whole selected block.

Marking several lines of text and pressing Shift+Tab reverses this behaviour. It removes spaces/tabs at the start of each line in order to reduce the indentation of the whole block.

Double-clicking on source text:

Double-clicking on a word selects the whole word as usual. However in some cases, double-clicking has a special meaning:

When double-clicking on the name of a procedure that is defined in the current source while holding down the Ctrl Key, the cursor automatically jumps to the declaration of this procedure.

When double-clicking on an IncludeFile or XincludeFile statement, the IDE will try to open that file. (This is only possible if the included file is written as a literal string, and not through for example a constant.)

In the same way, if you double-click on an IncludeBinary statement, the IDE will try to display that file in the internal file viewer .

Marking of matching Braces and Keywords:

```
46      Select Event
47        Case #PB_Event_Gadget
48
49          Select EventGadget()
50
51            Case 1
52              SetGadgetState(10,
53
54          EndSelect
55
56        Case #PB_Event_Menu ; We
57          SetGadgetText(10, GetGa
58
59      EndSelect
```

When the cursor is on an opening or closing brace the IDE will hilight the other brace that matches it. If a matching brace could not be found (which is a syntax error in PureBasic) the IDE will hilight the current brace in red. This same concept is applied to keywords. If the cursor is on a Keyword such as "If", the IDE will underline this keyword and all keywords that belong to it such as "Else" or "EndIf". If there is a mismatch in the keywords it will be underlined in red. The "Goto matching Keyword" menu entry described below can be used to quickly move between the matching keywords.

The brace and keyword matching can be configured in the Preferences .

Command help in the status bar:

```
ButtonGadget(#Gadget, x, y, Width, Height, Text$ [, Flags]) - Create a button gadget in the current GadgetList
```

While typing, the IDE will show the needed parameters for any PureBasic function whose parameters you are currently typing. This makes it easy to see any more parameters you still have to add to this function. This also works for procedures , prototypes , interfaces or imported functions in your code as long as they are declared in the same source code or project .

Folding options:

```
319 ⊞ Procedure.s ZipFileReques
377
378 ⊟ Procedure.s OpenZipFileRe
379 │   ProcedureReturn ZipFile
380 └ EndProcedure
```

When special folding keywords are encountered (Procedure / EndProcedure by default. More can be added), the IDE marks the region between these keywords on the left side next to the line numbers with a [-] at the starting point, followed by a vertical line to the end point.

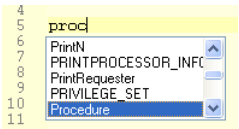
By clicking on the [-], you can hide ("fold") that section of source code to keep a better overview of larger

source files. The [-] will turn into a [+]. By clicking again, the code will again be shown (”unfolded”) again.

Note: Even though the state of these folded code lines is remembered when you save/reopen the file, the actual created code file always contains all lines. This only affects the display of the code in the IDE, not the code itself.

Another default fold keyword is ”;{” and ”;}”. Since ”;” marks a comment in PB, these will be totally ignored by the compiler. However, they provide the possibility to place custom fold points that do not correspond to a specific PB keyword.

Auto complete:



So that you do not have to remember the exact name of every command, there is the Auto complete feature to make things easier.

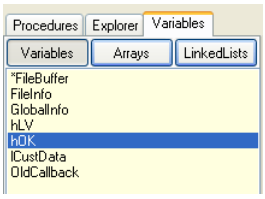
After you have typed the beginning of a command, a list of possible matches to the word start you have just typed will be displayed. A list of options is also displayed when you typed a structured variable or interface followed by a ”\”.

You can then select one of these words with the up/down keys and insert it at the point you are by pressing the Tab key. You can also continue typing while the list is open. It will select the first match that is still possible after what you typed, and close automatically when either you have just typed an exact match or if there are no more possible matches in the list.

Escape closes the auto complete list at any time. It also closes if you click with the mouse anywhere within the IDE.

Note: You can configure what is displayed in the Auto complete list, as well as turning off the automatic popup (requiring a keyboard shortcut such as Ctrl+Space to open list) in the Preferences. See the Auto complete section of Customizing the IDE for more information.

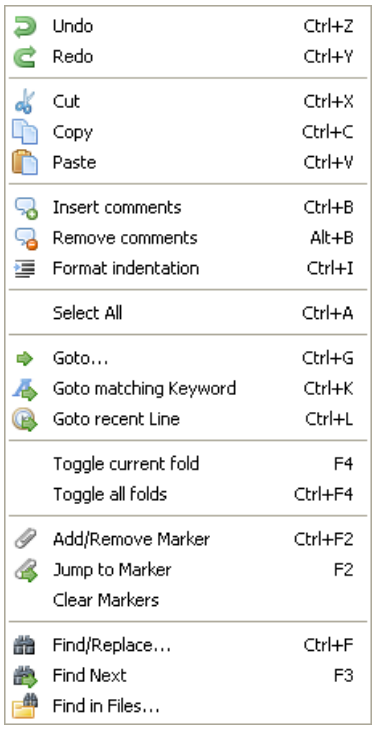
Tools Panel on the side:



Many tools to make navigating/editing the source code easier can be added to the Tools Panel on the side of the editor window. For an overview of them and how to configure them, see Buildin Tools .

The Edit Menu:

Following is an explanation of the Items in the Edit menu. Note that many of the Edit menu items are also accessible by right clicking on the source code, which opens a popup menu.



Undo
Undoes the last done action in the code editing area. There is an undo buffer, so several actions can be undone.

Redo
Redo the last action undone by the undo function.

Cut
Copy the selected part of the source code to the clipboard and remove it from the code.

Copy
Copy the selected text to the Clipboard without deleting it from the code.

Paste
Insert the content of the Clipboard at the current position in the code. If any text is selected before this, it will be removed and replaced with the content of the Clipboard.

Insert comments
Inserts a comment (";") before every line of the selected code block. This makes commenting large blocks of code easier than putting the ; before each line manually.

Remove comments
Removes the comment characters at the beginning of each selected line. This reverts the "Insert comments" command, but also works on comments that were set manually.

Format indentation
Reformats the indentation of the selected lines to align with the code above them and to reflect the keywords that they contain. The rules for the indentation can be specified in the preferences .

Select all Selects the whole source code.

Goto
This lets you jump to a specific line in your source code.

Goto matching Keyword
If the cursor is currently on a keyword such as "If" this menu option jumps directly to the keyword that matches it (in this case "EndIf").

Goto recent line
The IDE keeps track of the lines you view. For example if you switch to a different line with the above Goto function, or with the Procedure Browser tool. With this menu option you can jump back to the previous position. 20 such past cursor positions are remembered.

Note that this only records greater jumps in the code. Not if you just move up/down a few lines with the cursor keys.

Toggle current fold
This opens/closes the fold point in which the cursor is currently located.

Toggle all Folds
This opens/closes all fold points in the current source. Very useful to for example hide all procedures in the code. Or to quickly see the whole code again when some of the code is folded.

Add/Remove Marker
Markers act like Bookmarks in the source code. There presence is indicated by a little arrow next to the line numbers. You can later jump to these markers with the "Jump to marker" command.

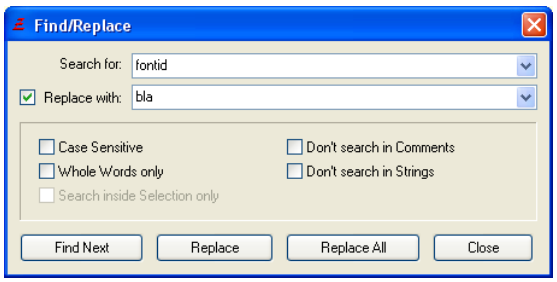
The "Add/Remove Marker" sets or removes a marker from the current line you are editing.

Note: You can also set/remove markers by holding down the Ctrl Key and clicking on the border that holds the markers. (not the Linenumber part of it)

Jump to Marker
This makes the cursor jump to the next marker position further down the code from the current cursor position. If there is no marker after the cursor position, it jumps to the first on in the source code. So by pressing the "Jump to Marker" shortcut (F2 by default) several times, you can jump to all the markers in the code.

Clear Markers This removes all markers from the current source code.

Find/Replace



The find/replace dialog enables you to search for specific words in your code, and also to replace them with something else.

The "Find Next" button starts the search. The search can be continued after a match is found with the Find Next menu command. (F3 by default).

You can make the search more specific by enabling one of the checkboxes:
Case Sensitive : Only text that matches the exact case of the search word will be found.
Whole Words only : Search for the given word as a whole word. Do not display results where the search word is part of another word.
Don't search in Comments : Any match that is found inside a comment is ignored.
Don't search in Strings : Any match that is found inside a literal string (in " ") is ignored.
Search inside Selection only : Searches only the selected region of code. This is really useful only together with the "Replace All" button, in which case it will replace any found match, but only inside the selected region.

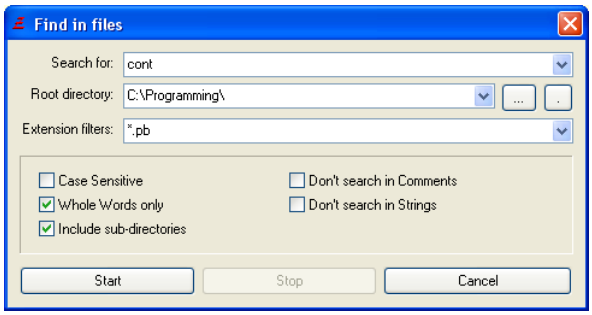
By enabling the "Replace with" checkbox, you go into replace mode. "Find Next" will still only search, but with each click on the "Replace" button, the next match of the search word will be replaced by

whatever is inside the "Replace with" box.

By clicking on "Replace All", all matches from the current position downwards will be replaced (unless "Search inside Selection only" is set).

Find Next
This continues the search for the next match of the last search started by the Find/Replace dialog.

Find in Files



The Find in Files Dialog lets you carry out a search inside many files in a specific directory.

You have to specify a search keyword, as well as a base directory ("root directory") in which to search. You can customize the searched files by specifying extension filters. Any number of filters can be given separated by ",". (*.* or an empty extension field searches all files). As with "Find/Replace", there are checkboxes to make the search more specific.

The "Include sub-directories" checkbox makes it search (recursively) inside any subdirectory of the given root directory too.

When starting the search, a separate window will be opened displaying the search results, giving the file, line number as well as the matched line of each result.

Double-clicking on an entry in the result window opens that file in the IDE and jumps to the selected result line.

Chapter 11

Managing projects

The IDE comes with features to easily handle larger projects. These features are completly optional. Programs can be created and compiled without making use of the project management. However, once a program consists of a number of source code and maybe other related files, it can be simpler to handle them all in one project.

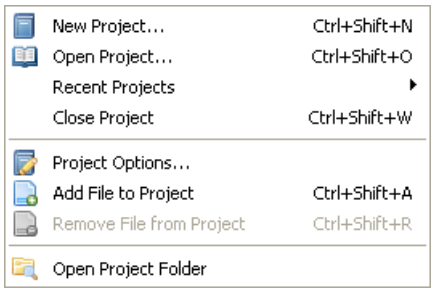
Project management overview

A project allows the management of multiple source codes and other related files in one place with quick access to the files through the project tool . Source files included in a project can be scanned for AutoComplete even if they are not currently open in the IDE. This way functions, constants, variables etc. from the entire project can be used with AutoComplete. The project can also remember the source files that are open when the project is closed and reopen them the next time to continue working exactly where you left off.

Furthermore, a project keeps all the compiler settings in one place (the project file) and even allows to manage multiple ”compile targets” per project. A compile target is just a set of compiler options. This way multiple versions of the same program, or multiple smaller programs in one project can be easily compiled at once.

All filenames and paths in a project are stored relative to the project file which allows a project to be easily moved to another location as long as the relative directory structure remains intact.

The Project menu



New Project
Creates a new project. If there is a project open at the time it will be closed. The project options window will be opened where the project filename has to be specified and the project can be configured.

Open Project
Opens an existing project. If there is a project open at the time it will be closed. Previously open source

codes of the project will be opened as well, depending on the project configuration.

Recent Projects
This submenu shows a list of recently opened project files. Selecting one of the entries openes this project.

Close Project
Closes the currently open project. The settings will be saved and the currently open source files of the project will be closed, depending on the project configuration.

Project Options
Opens the project options window. See below for more information.

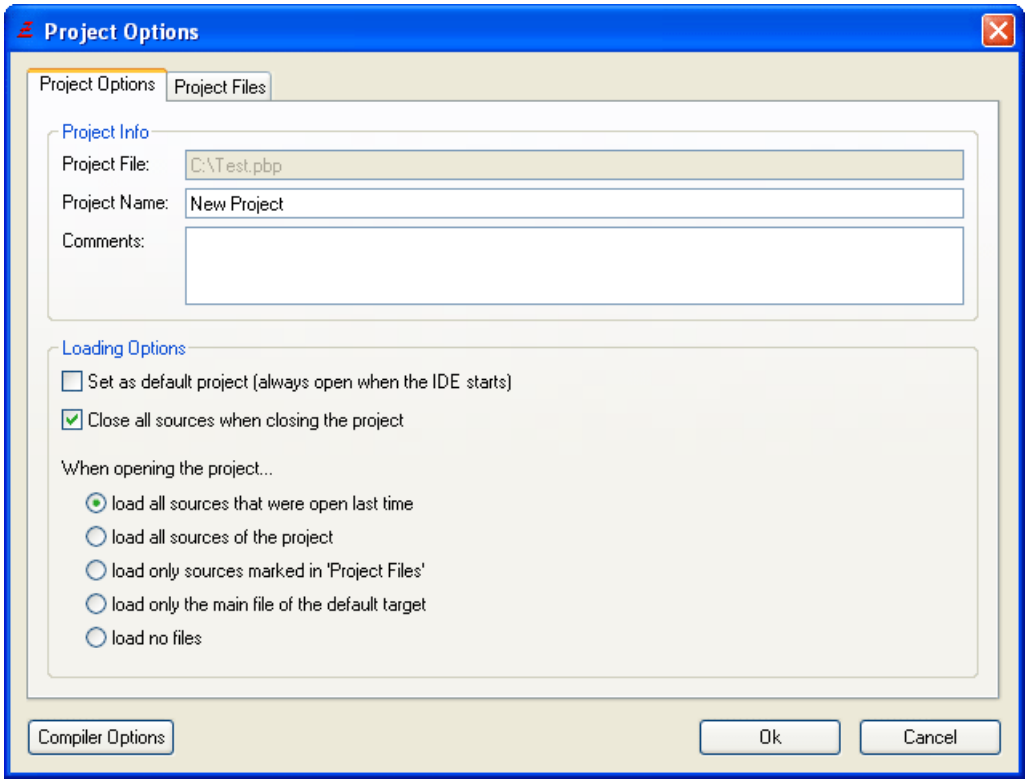
Add File to Project
Adds the currently active source code to the current project. Files belonging to the project are marked with a ">" in the file panel.

Remove File from Project
Removes the currently active source from the current project.

Open Project folder
Opens the folder that contains the project file in whatever file manager is available on the system.

The project options window

The project options window is the central configuration for the project. The general project settings as well as the settings for the individual files in the project can me made here.



The following settings can be made on the "Project Options" tab:

Project File
Shows the filename of the project file. This can only be changed during project creation.

Project Name

The name of the project. This name is displayed in the IDE title bar and in the "Recent Projects" menu.

Comments
This field allows to add some comments to the project. They will be displayed in the project info tab.

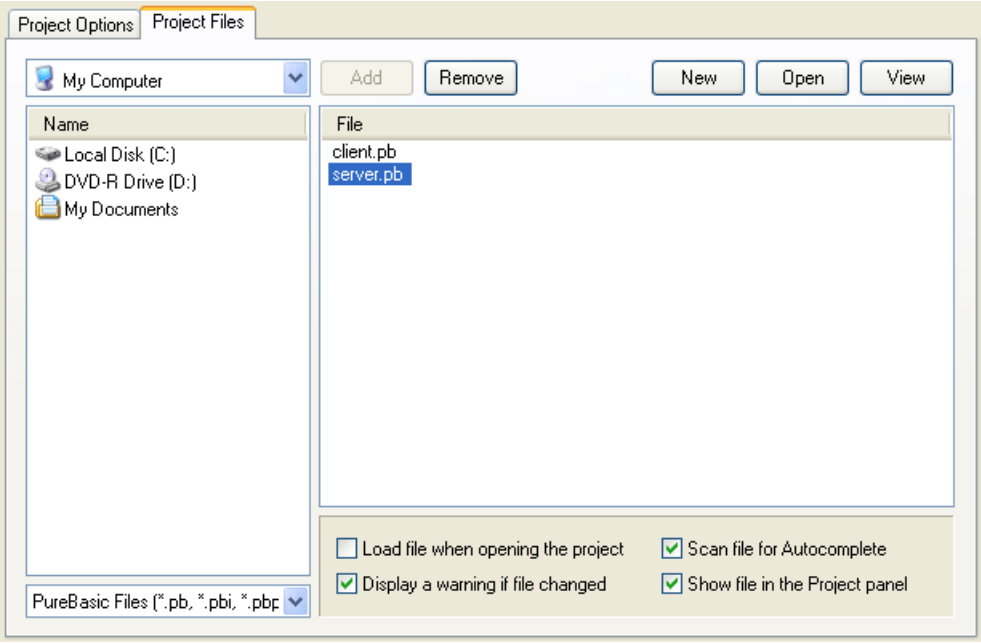
Set as default project
The default project will be loaded on every start of the IDE. Only one project can be the default project at a time. If there is no default project, the IDE will load the project that was open when the IDE was closed last time if there was one.

Close all sources when closing the project
If enabled, all sources that belong to the project will be closed automatically when the project is closed.

When opening the project...

> **load all sources that where open last time**
> When the project is opened, all the sources that were open when the project was closed will be opened again.
>
> **load all sources of the project**
> When the project is opened, all (source-)files of the project will be opened.
>
> **load only sources marked in 'Project Files'**
> When the project is opened, only the files that are marked in the 'Project Files' tab will be opened. This way you can start a session always with this set of files open.
>
> **load only the main file of the default target**
> When the project is opened, the main file of the default target will be opened too.
>
> **load no files**
> No source files are opened when the project is opened.

The "Project Files" tabs shows the list of files in the project on the right and allows to change their settings. The explorer on the left is for the selection of new files to be added.



The buttons on the top have the following function:

> **Add**
> Add the selected file(s) in the explorer to the project.
>
> **Remove**
> Remove the selected files in the file list from the project.

New
Shows a file requester to select a filename for a new source file to create. The new file will be created, opened in the IDE and also added to the project.

Open
Shows a file requester to select an existing file to open. The file will be opened in the IDE and added to the project.

View
Opens the selected file(s) in the file list in the IDE or if they are binary files in the FileViewer.

The checkboxes on the bottom specify the options for the files in the project. They can be applied to a single file or to multiple files at once by selecting the files and changing the state of the checkboxes. The settings have the following meaning:

Load file when opening the project
Files with this option will be loaded when the project is open and the "load only sources marked in 'Project Files"' option is specified on the "Project Options" tab.

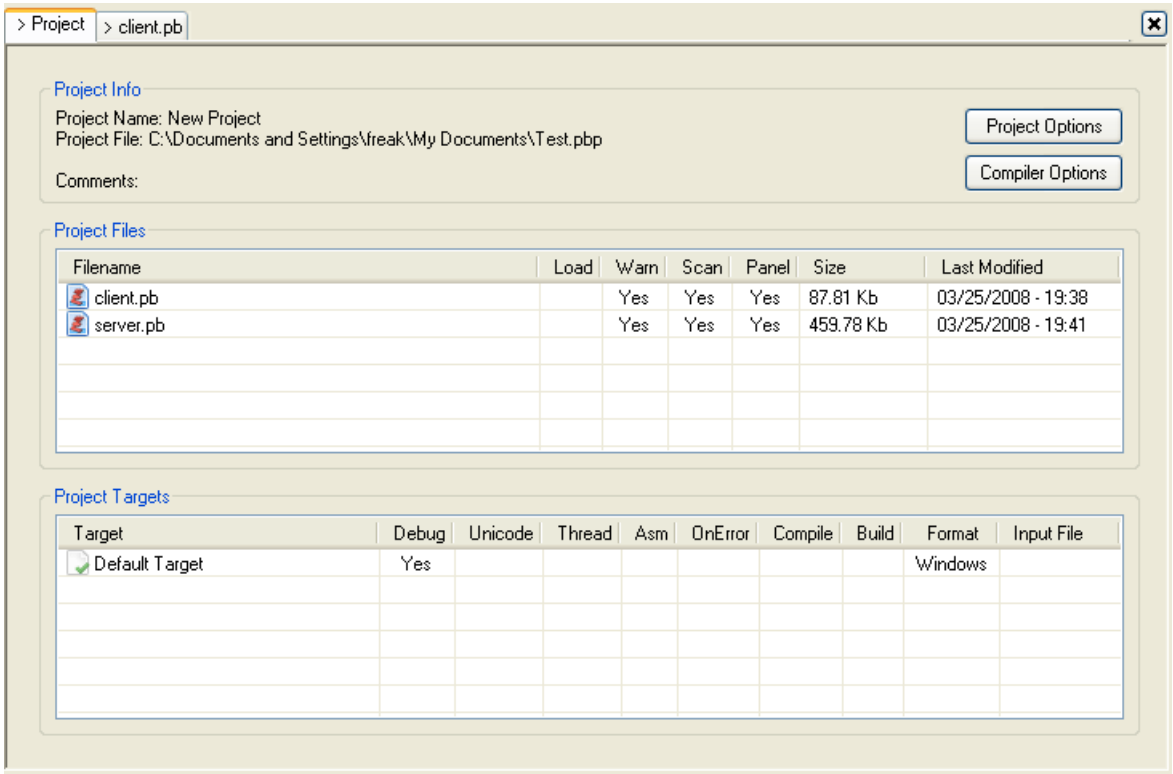
Display a warning if file changed
When the project is closed, the IDE will calculate a checksum of all files that have this option set and display a warning if the file has been modified when the project is opened the next time. This allows to be notified when a file that is shared between multiple projects has been edited while working on another project. This option should be disabled for large data files to speed up project loading and saving, or for files which are changed frequently to avoid getting a warning every time the project is opened.

Scan file for AutoComplete
Files with this option will be scanned for AutoComplete data even when they are not currently loaded in the IDE. This option is on by default for all non-binary files. It should be turned off for all files that do not contain source code as well as for any files where you do not want the items to turn up in the AutoComplete list.

Show file in Project panel
Files with this option will be displayed in the project sidepanel. If the project has many files it may make sense to hide some of them from the panel to have a better overview and faster access to the important files in the project.

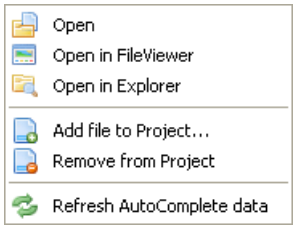
The project overview

When a project is open, the first tab of the file panel shows an overview of the project and its files.



Project Info
This section shows some general info about the project, such as the project filename, its comments or when and where the project was last opened.

Project Files
This section shows all files in the project and their settings from the Project Options window. Double-clicking on one of the files opens the file in the IDE. Right-clicking displays a context menu with further options:



Open - Open the file in the IDE.
Open in FileViewer - Open the file in the FileViewer of the IDE.
Open in Explorer - Open the file in the operating systems file manager.
Add File to Project - Add a new file to the project.
Remove File from Project - Remove the selected file(s) from the project.
Refresh AutoComplete data - Rescan the file for AutoComplete items.

Project Targets
This section shows all compile targets in the project and some of their settings. Double-clicking on one of the targets opens this target in the compiler options . Right-clicking one one of the targets displays a context menu with further options:

Edit target - Open the target in the compiler options.
Set as default target - Set this target as the default target.
Enable in 'Build all Targets' - Include this target in the 'Build all Targets' compiler menu option.

The project panel

There is a sidepanel tool which allows quick access to the files belonging to the project. For more information see the builtin tools section.

Chapter 12

Compiling your programs

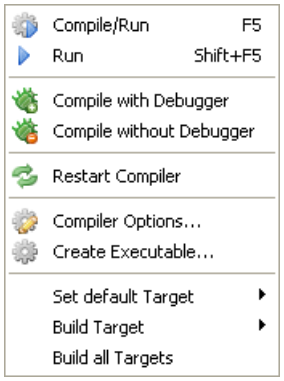
Compiling is simple. Just select ”Compile/Run” (F5 by default) and your program will be compiled and executed for a testing run.

To customise the compiling process, you can open the ”Compiler options” dialog. The settings made there are associated with the current source file or the current project, and also remembered when they are closed. The place where this information is saved can be configured. By default, it is saved at the end of the source code as a comment. (invisible in the IDE)

In case of an error that prevents the compiler from completing the compilation, it aborts and displays an error message. This message is also logged in the error log, and the line that caused the error is marked.

A number of functions from older versions of PureBasic that have been removed from the package still exist for a while as a compatibility wrapper to allow older codes to be tested/ported more easily. If such a function is used in the code, the compiler will issue a warning. A window will be opened displaying all warnings that were issued during compilation. Doubleclicking on a warning will display the file/line that caused the warning. Note that such compatibility wrappers will not remain indefinitely but will be removed in a future update, so it is recommended to fix issues that cause a compiler warning instead of relying on such deprecated functions.

The compiler menu



Compile/Run
This compiles the current source code with the compiler options that were set for it and executes it. The executable file is stored in a temporary location, but it will be executed with the current path set to the directory of the source code. (so loading a file from the same directory as the source code will work)

The source code need not be saved for this. (but any included files must be saved.)

The ”Compile/Run” option respects the debugger setting (on or off) from the compiler options or

debugger menu. (they are the same)

Run
This executes the last compiled source code once again. Whether or not the debugger is enabled depends on the setting of the last compilation.

Compile with Debugger
This is the same as "Compile/Run" except that it ignores the debugger setting and enabled the debugger for this compilation. This is useful when you usually have the debugger off, but want to have it on for just this one compilation.

Compile without Debugger
Same as "Compile with Debugger" except that it forces the debugger to be off for this compilation.

Restart Compiler (not present on all OS)
This causes the compiler to restart. It also causes the compiler to reload all the libraries and resident files, and with that, the list of known PureBasic functions, Structures, Interfaces and Constants is updated too. This function is useful when you have added a new User Library to the PB directory, but do not want to restart the whole IDE. It is especially useful for library developers to test their library.

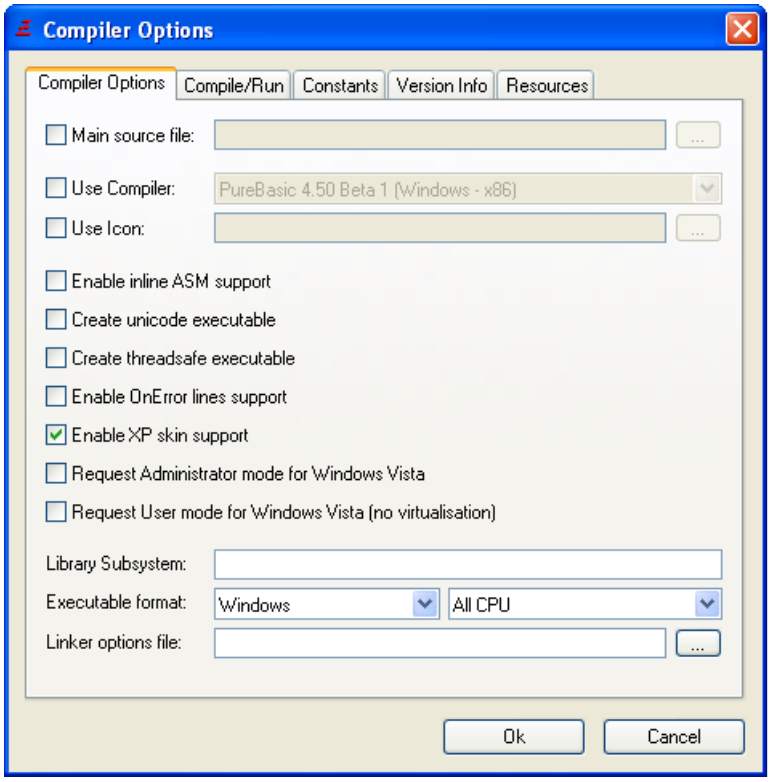
Compiler Options
This opens the compiler options dialog, that lets you set the options for the compilation of this source file.

Set default Target
When a project is open, this submenu shows all compile targets and allows to quickly switch the current default target. The default target is the one which is compiled/executed with the "Compile/Run" menu entry.

Build Target
When a project is open, this submenu shows all compile targets and allows to directly compile one of them.

Build all Targets
When a project is open, this menu entry compiles all targets that have this option enabled in the compiler options. A window is opened to show the build progress.

Compiler options for non-project files



Main source file
By enabling this option, you can define another file that will be the one sent to the compiler instead of this one. The use of this is that when you are editing a file that does not run by itself, but is included into another file, you can tell the compiler to use that other file to start the compilation.

Note: When using this option, you MUST save your source before compiling, as only files that are written to disk will be used in this case. Most of the compiler settings will be taken from the main source file, so when setting this, they are disabled. Only some settings like the debugger setting will be used from the current source.

Use Compiler
This option allows the selection of a different compiler to use instead of the compiler of the current PureBasic version. This makes it easy to compile different versions of the same program (x86 and x64 or PowerPC) without having to start up the IDE for the other compiler just for the compilation. Additional compilers for this option have to be configured in the preferences .

If the compiler version matches that of the default compiler but the target processor is different then the builtin debugger of the IDE can still be used to debug the compiled executable. This means that an executable compiled with the x86 compiler can be debugged using the x64 IDE and vice versa on Windows and Linux. The same applies to the x86 and PowerPC compilers for Mac OSX. This is especially useful as this way the fast x86 IDE and debugger can be used on a Mac with an Intel processor while still compiling programs for the PowerPC processor through the slower emulation. If the version does not match then the standalone debugger that comes with the selected compiler will be used for debugging to avoid version conflicts.

Use Icon (Windows and MacOS X only)
Here you can set an icon that will be displayed when viewing the created executable in the explorer. It is also displayed in the title bar of your programs windows and the Taskbar.

Enable inline ASM support
This enables the inline asm parser. See the Inline x86 ASM section of the help file for more information on this option.

Create unicode executable
This tells the compiler to create a unicode executable. The string management, as well as all PB functions will work with unicode strings . To write unicode text in the sourcefiles, it is recommended to set the text encoding of the sourcefile to UTF8 (see below), as then unicode text can be directly entered in the IDE without problem.

To debug unicode programs, it is best to select the Standalone Debugger from the Preferences , as the debugger that is integrated into the IDE can currently not display unicode strings. (it converts them to ascii before displaying them). The standalone Debugger however can handle this.

Create threadsafe executable
This tells the compiler to use a special version of certain commands to make them safe to be used in threads. See the Thread library for more information.

This also enables the Debugger to display correct information if threads are used. Without this option, the debugger might output wrong line numbers when threads are involved for example.

Enable XP Skin support (Windows only)
Includes support for skinned windows on Windows XP or Windows Vista

Request Administrator mode for Windows Vista (Windows only)
The created executable will always be started with administrator rights on Windows Vista. (it will not launch if the administrator password is not entered). This option should be set for programs that need to access restricted folders or restricted areas of the registry to get full access.
If this option is turned on, the standalone debugger will automatically selected when debugging, so the program can be tested in administrator mode.

Note: This option has no effect when the program is run on other versions of Windows.

Request User mode for Windows Vista (Windows only)
This option disables the "Virtualisation" feature for this executable on Windows Vista. Virtualisation caused file and registry access to be redirected to a special user folder if the user does not have the needed rights to do the operation. (this is done for compatibility with older programs)
Note that this redirection is done without notifying the user which can lead to some confusion if he tries to find saved files on the filesystem. Because of this, it is recommended to disable this feature if the program complies with the Windows Vista file/registry access rules.

Note: This option has no effect when the program is run on other versions of Windows. It cannot be combined with the "Administrator mode" option above.

Enable OnError lines support (Windows only)
Includes line numbers information with the executable for the OnError-Library .

Library Subsystem
Here you can select a different PB subsystem for compilation. Currently the following subsystems are supported:

<u>Windows</u>
nt4 : uses DirectX3 only for any library that uses DirectX in order to be compatible to Windows NT 4.
opengl : uses OpenGL for the 3D libraries.

<u>Linux</u>
gtk1 : uses Gtk1.2 instead of Gtk2 for the compilation. (requires the gtk1.2-dev package)
Note that this subsystem is no longer maintained and some features may be missing.

<u>MacOSX</u>
There are currently no official subsystems for MacOSX.

Executable format
This allows you to specify the created executable format:
<u>Windows</u> : a normal windows executable.

Console : an executable with a default console. This one still can create windows and such, but it always has a console open. When executed from a command prompt, this executable type uses the command terminal as its console and writes there, whereas the "Windows" executable would create a separate Console window when using OpenConsole() . This setting must be used to create a Console application that can have its input/output redirected with pipes.
Shared DLL : create a windows dll. See Building a dll for more info.
Note: When you do "Compile/Run" with a dll source code, it is executed as a normal executable. A dll is only created when you use "create executable".

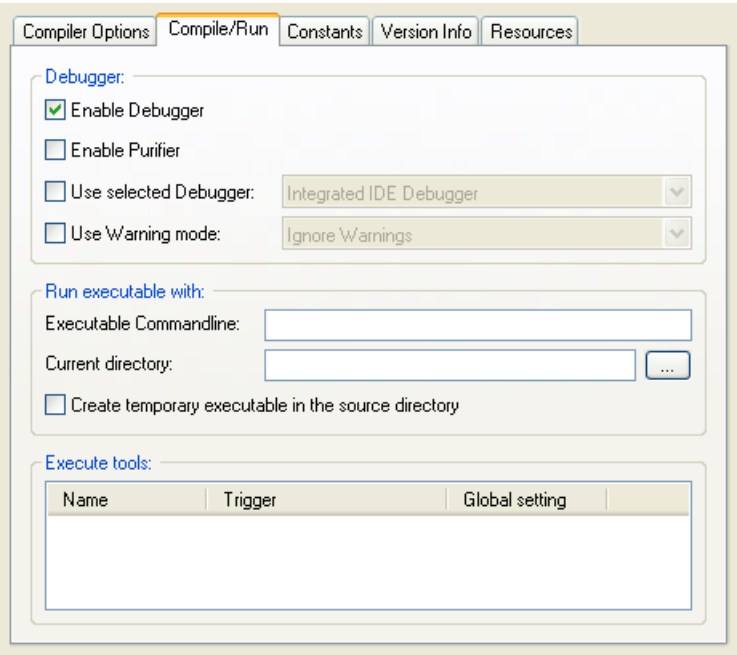
Cpu Optimisation (next to Executable format)
This setting allows to include Cpu optimised PB functions in your executable:
All CPU : The generic functions are included that run on all CPUs.
Dynamic CPU : The generic functions as well as any available CPU specific function are included. The function to execute is decided at runtime. This creates a bigger executable, but it will run as fast as possible on all CPUs.
All other options : Include only the functions for a specific CPU. The executable will not run on any Cpu that does not support this feature.

Note: No PB functions actually support this feature for now (it is ignored for them). However, some User Libraries include such optimisations.

Linker options file
A textfile can be specified here with further commandline options that should be passed to the linker when creating the executable. The file should contain one option per line.

Compile/Run

This section contains options that affect how the executable is run from the IDE for testing. Except for the tools option, they have no effect when the "Create executable" menu is used.



Enable Debugger
This sets the debugger state (on/off) for this source code, or if the main file option is used, for that file too. This can also be set from the debugger menu.

Enable Purifier
This enables purifier support for the debugger. The purifier can detect a certain type of programming errors such as writing past the end of an allocated memory buffer. See Included debugging tools for

more details.

Use selected Debugger
This allows to choose a different debugger type for this file only. If this option is disabled, the default debugger is used which can be specified in the preferences .

Use Warning mode
This allows to choose a different warning mode for this file only. If this option is disabled, the default setting is used which can be specified in the preferences . The available options are:
Ignore Warnings: Warnings will be ignored without displaying anything.
Display Warnings: Warnings will be displayed in the error log and the source code line will be marked, but the program continues to run.
Treat Warnings as Errors: A warning will be treated like an error.

Executable Commandline
The string given here will be passed as the commandline to the program when running it from the IDE.

Current directory
The directory specified here will be set as the current directory for the program when running it from the IDE.

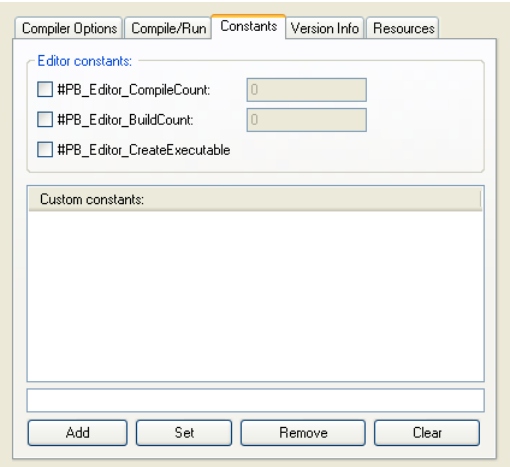
Create temporary executable in the source directory
With this option turned on, the temporary executable file for running the program from the IDE will be placed inside the source directory. This can be useful if the program depends on files inside the source directory for testing. With this option turned off, the executable is created in the systems temporary directory.

Execute tools
Here external tools can be enabled on a per-source basis. The "Global settings" column shows if the tool is enabled or disabled in the tools configuration . A tool will only be executed for the source if it is both enabled globally and for this source.

Note: For a tool to be listed here, it must have the "Enable Tool on a per-source basis" option checked in the tools configuration and be executed by a trigger that is associated with a source file. (ie not by menu or editor startup for example)

Constants

In this section, a set of special editor constants as well as custom constants can be defined which will be predefined when compiling this source.



#PB_Editor_CompileCount
If enabled, this constant holds the number of times that the code was compiled (both with "Compile/Run" and "Create Executable") from the IDE. The counter can be manually edited in the

input field.

#PB_Editor_BuildCount
If enabled, this constant holds the number of times that the code was compiled with "Create Executable" only. The counter can be manually edited in the input field.

#PB_Editor_CreateExecutable
If enabled, this constants holds a value of 1 if the code is compiled with the "Create Executable" menu or 0 if "Compile/Run" was used.

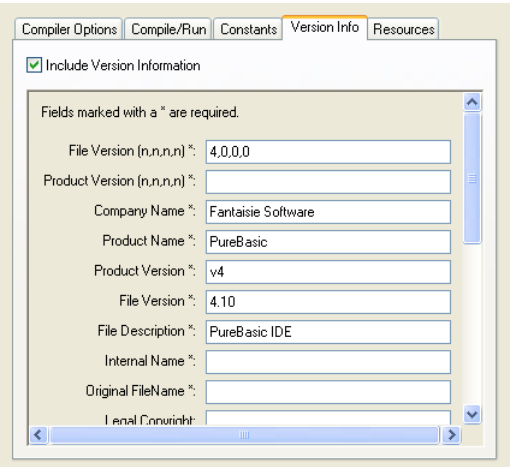
Custom constants
Here, custom constants can be defined and then easily switched on/off through checkboxes. Constant definitions should be added as they would be written within the sourcecode. This provides a way to enable/disable certain features in a program by defining a constant here and then checking in the source for it to enable/disable the features with CompilerIf/CompilerEndIf .

Inside the definition of these constants, environment variables can be used by specifying them in a "bash" like style with a "$" in front. The environment variable will be replaced in the constant definition before compiling the source. This allows to pass certain options of the system that the code is compiled on to the program in the form of constants.

Example: #Creator="$USERNAME"
Here, the $USERNAME will be replaced by the username of the logged in user on Windows systems. If an environment variable does not exist, it will be replaced by an empty string.

Note: To test within the sourcecode if a constant is defined or not, the Defined() compiler function can be used.

Version Information



By enabling this, a resource is included in the executable with information about your program. It can be viewed by right-clicking on the executable in the windows explorer and selecting "Properties". Also it can be read by other programs such as setup tools.

Fields marked with a * are required if you want to include the version info. (if not all required fields are set, the information may not display correctly on some versions of Windows.

The first two fields MUST be composed of 4 numbers separated by commas. All other fields may be any string. In the 3 empty boxes, you can define your own fields to include in the Version info block.

In all the string fields, you may include special tokens that are replaced when compiling:
%OS : replaced with the version of Windows used to compile the program
%SOURCE : replaced with the filename (no path) of the source file.
%EXECUTABLE : replaced with the name of the created executable (this only works when "create

executable" is used, not with "Compile/Run".
%COMPILECOUNT : replaced with the value for the #PB_Editor_CompileCount constant.
%BUILDCOUNT : replaced with the value for the #PB_Editor_BuildCount constant.

Furthermore, you can use any token listed with the FormatDate() command. These tokens will be replaced with their respective meaning in FormatDate() used with the date of the compilation (i.e. %yy gives the year of the compilation)

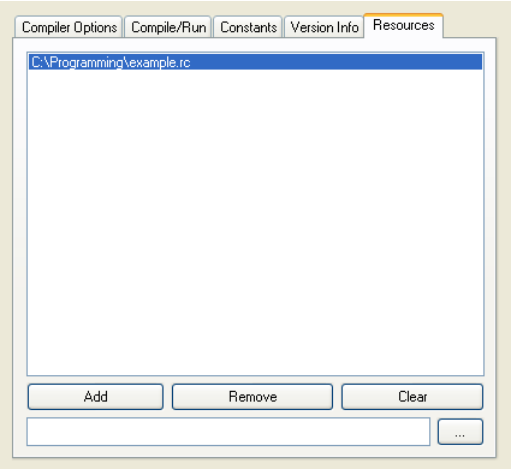
Meaning of the lower 3 fields:

File OS
Specifies the OS that this Program is compiled for. (using VOS_DOS or VOS_WINDOWS16 makes not much sense. They are only included to have the full list)

File Type
Type of the executable. (here VFT_UNKNOWN, VFT_APP or VFT_DLL only really make sense for PB programs)

Language
Specifies the language in which this version info is written.

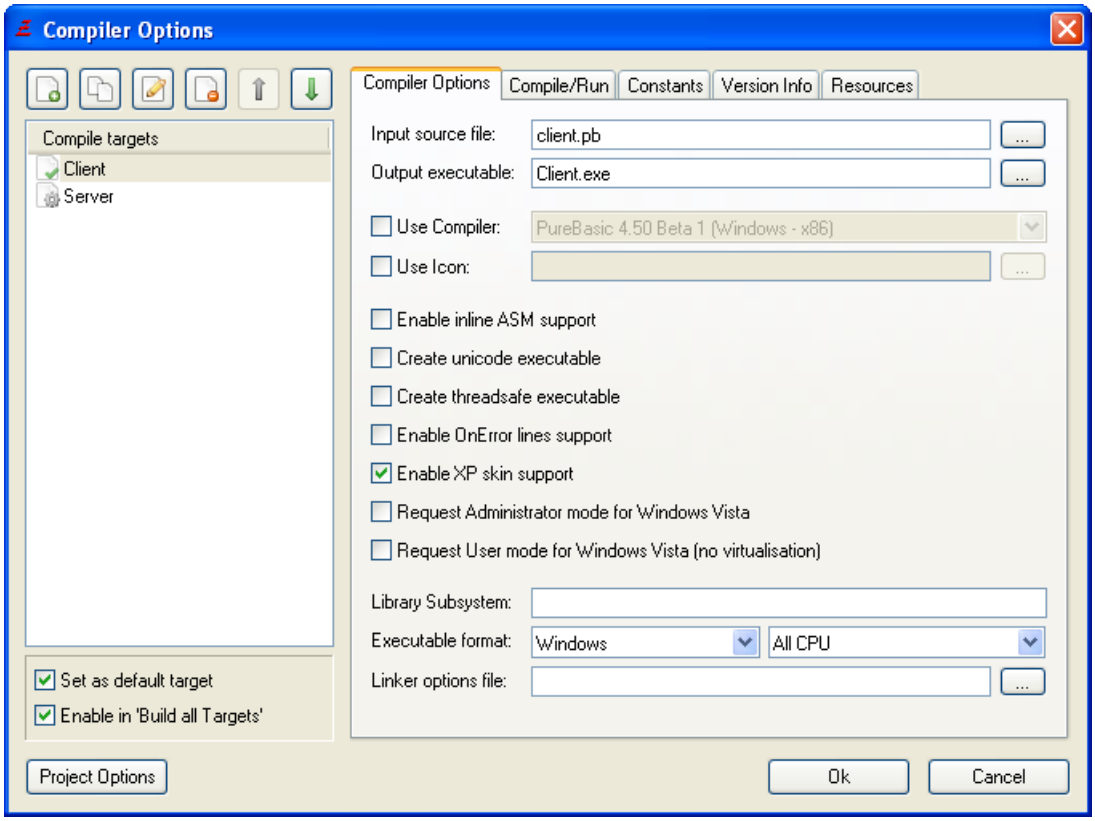
Resources



Here you can include as many Resource scripts (*.rc files) as you want. They will be compiled and included with the executable. You can use any resource editor (for example the PellesC IDE) to create such scripts.

Note: Since Resources are a specific to the Windows platform only, PB does not include a Library to manage them and they are not further documented here. See documentation on the Windows API and resources for more information.

Compiler options for projects



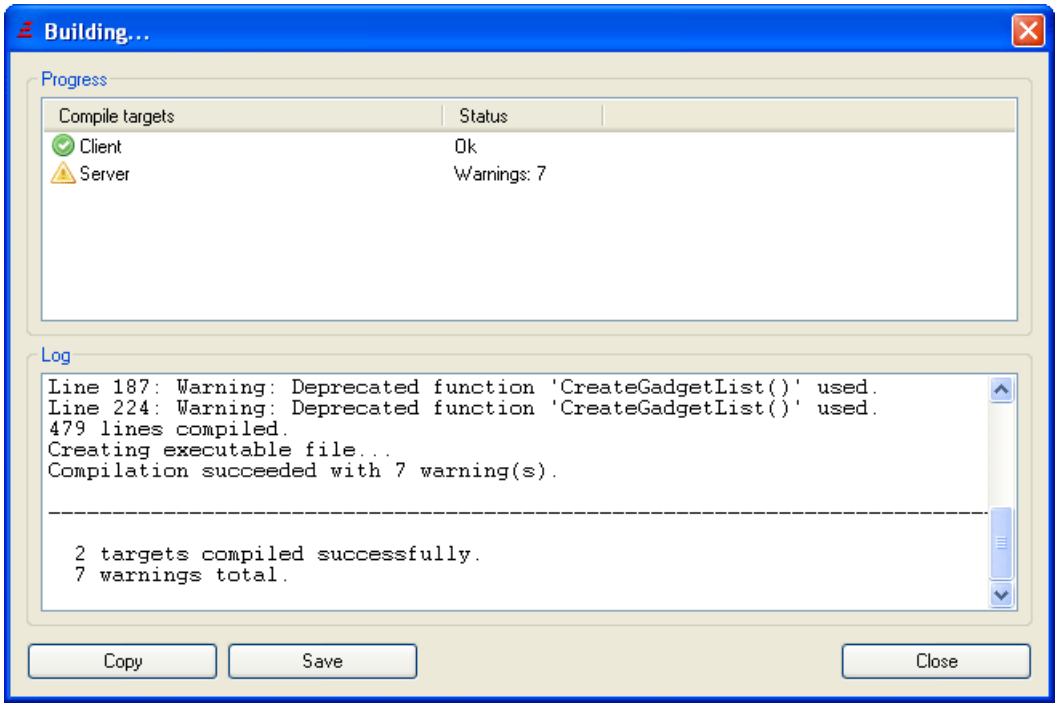
The compiler options for projects allow the definition of multiple compile targets. Each target is basically a set of compiler options with a designated source file and output executable. The left side of the compiler options window is extended with the list of the defined compile targets. The toolbar on top of it allows to create, delete, copy, edit or move targets in the list.

The default target is the one which will be compiled when the ”Compile/Run” menu entry is selected. It can be quickly switched with the ”Set as default target” checkbox or from the compiler menu. The ”Enable in ’Build all Targets”’ option specifies wether or not the selected target will be built when the ’Build all Targets’ menu entry is used.

The right side of the compiler options is almost the same as in the non-project mode and reflects the settings for the compile target that is currently selected on the left. The only difference are the ”Input source file” and ”Output executable” fields on the first tab. These fields have to be specified for all compile targets. Other than that, the compiler options are identical to the options described above.

In project mode, the information about the compile target is stored in the project file and not in the individual source files. Information that belongs to the file (such as the folding state) are still saved for the individual source files in the location specified by the Preferences .

The Build progress window



When the 'Build all Targets' menu entry is selected on an open project, all targets that have the corresponding option set in the compiler options will be compiled in the order they are defined in the compiler options. The progress window shows the current compile progress as well as the status of each target. When the process is finished, the build log can be copied to the clipboard or saved to disk.

Chapter 13

Using the debugger

PureBasic provides a powerful debugger that helps you find mistakes and bugs in your source code. It lets you control the program execution, watch your variables , arrays or linkedlists or display debug output of your programs. It also provides advanced features for assembly programmer to examine and modify the CPU registers or view the program stack, or the Memory of your program. It also provides the possibility to debug a program remotely over the network.

To enable the debugger for your program, you can select "Use Debugger" from the debugger menu, or set it in your programs Compiler options. By using the "Compile with Debugger" command from the Compiler menu, you can enable the debugger for just one compilation.

You can directly use debugger commands in your source, such as CallDebugger, Debug, DebugLevel, DisableDebugger and EnableDebugger.

The PureBasic debugger comes in 3 forms:

A Debugger integrated directly with the IDE, for an easy to use, quick way to debug your programs directly from the programming environment. This debugger also provides the most features.

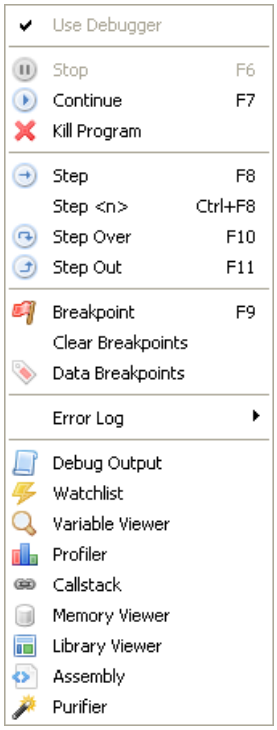
A separate, standalone debugger, that is useful for some special purposes (for example, when the same program must be executed and debugged several times at once) or to be used with third party code Editors. It provides most of the features of the integrated IDE debugger, but because it is separate from the IDE, some of the efficiency of the direct access from the IDE is lost. The standalone debugger can be used to debug programs remotely through a network connection.

A console only debugger. This debuggers primary use is for testing non-graphical environment like on linux systems without an X server, or to remotely develop through ssh.

The type of debugger that is used can be selected in the preferences .

All this debugging functionality however comes at a price. Running a program in debug mode is significantly slower in its execution that running it without the debugger. This should be no problem however, since this is for testing only anyway.
If you need to use the debugger, but have some parts in you program that require the full execution speed, you can disable the debugger in just that section with the DisableDebugger / EnableDebugger keywords.

The Debugger integrated into the IDE



You can access all the debugger features while the program is running from the debugger menu, or the corresponding toolbar buttons or shortcuts.

While you are debugging your program, all the source files that belong to that program (also included files) will be locked to read-only until the program has finished. This helps to ensure that the code that is marked as the currently executed one is has not actually been modified already without a recompilation.

Note that a program can be run only one time at once in IDE debugger mode. If you try to executed it again, you are given the option to execute it with the standalone Debugger.

Tip:
The debugger menu is also added to the system-menu of the Main IDE window (the menu you get when clicking the PB icon in the left/top of the window). This allows you to access the debugger menu also from the Taskbar, by right-clicking on the Taskbar-Icon of the IDE.

Program Control

There are functions for basic control of the running program. You can halt the execution to examine variables and the code position or let the code execute line by line to follow the program flow. While the program is halted, the line that is currently being executed is marked in your source code. (with very light-blue background color in the default colors.)

The state of the program can be viewed in the IDE status bar, and in the Error log area.

Menu commands for program control:

Stop
Halts the Program and displays the current line.

Continue
Continues the program execution until another stop condition is met.

Kill Program

This forces the program to end, and closes all associated debugger windows.

Step
This executes one line of source code and then stops the execution again.

Step <n>
This will execute a number of steps that you can specify and then stop the execution again.

Step Over
This will execute the current line in the source and then stop again, just like the normal 'Step'. The difference is that if the current line contains calls to procedures , the execution will not stop inside these procedures like it does with the normal 'Step', but it will execute the whole procedure and stop after it returned. This allows to quickly skip procedures in step mode.

Step Out
This will execute the remaining code inside the current procedure and stop again after the procedure has returned. If the current line is not in any procedure, a normal 'Step' will be done.

Line Breakpoints

Breakpoints are another way to control the execution of your program. With the Breakpoint menu command, you mark the currently selected line as a breakpoint (or remove any breakpoint that exists in that line).

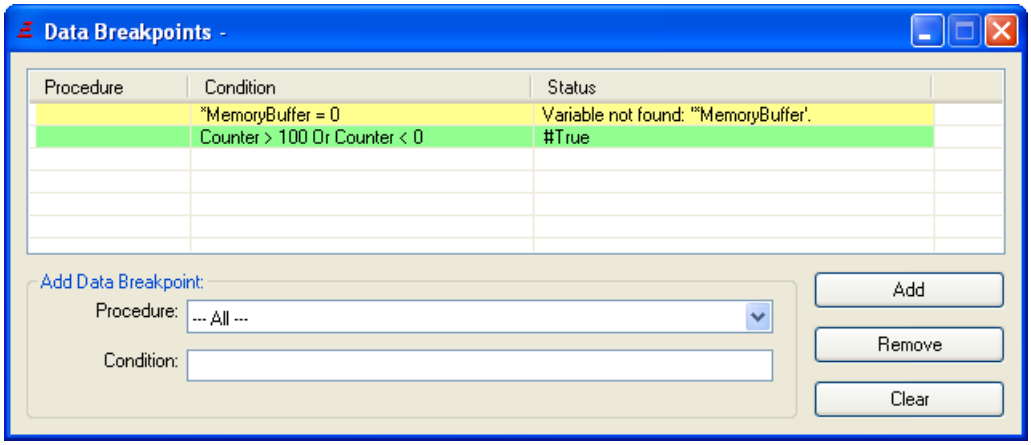
When the execution of the code reaches that line, it will stop at this point. Note that if you select a non-executable line (such as an empty line or a Structure definition), it will halt the execution on the next executable line after that.

After the execution of your program has stopped at a breakpoint, you can use any of the Program control commands to continue/end the execution.

Breakpoints can be set and removed dynamically, while your program is running, or while you are editing your source code. With the "Clear Breakpoints" command, you can clear all breakpoints in a source file.

Note: You can also set/remove Breakpoints by holding down the Alt Key and clicking on the border that contains the Breakpoint marks.

Data Breakpoints



In addition to the line specific breakpoints, the debugger also provides data breakpoints. Data breakpoints halt the program if a given condition is met. This way it is easy to find out when a variable or other value in the program changes and halt the program if that happens. The condition can be any PureBasic expression that can be evaluated to true or false. This can be anything that could be put

after an If keyword, including logical operators such as And, Or or Not. Most functions of the Math , Memory and String libraries are also available.

Example conditions:

```
MyVariable$ <> "Hello" Or Counter < 0  ; halt if MyVariable$ changes from "Hello" or if the Counter falls below zero
PeekL(*SomeAddress+500) <> 0           ; halt if the long value at the given memory location is not equal to zero
```

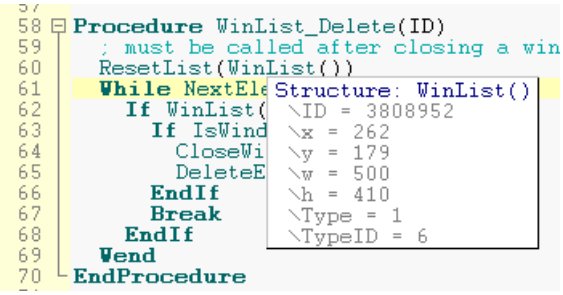
Data breakpoints can be added using the Data Breakpoint option from the Debugger menu. They can be limited to a specific procedure or they can be added for all code. The special "Main" entry of the procedure selection specifies that the data breakpoint should only be checked when the execution is not in any procedure.

The status column shows the status of all breakpoint conditions on their last evaluation. This can be true, false or an error if the condition is not a valid expression. Once a condition is evaluated to true, the program execution will be halted. This condition is automatically removed from the list as soon as the program continues, so that it does not halt the program again immediately.

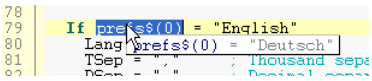
Note: Checking for data breakpoints slows down the program execution because the breakpoint conditions have to be re-evaluated for every executed line of code to check if the condition is met. So data breakpoints should only be added when needed to keep the program execution fast otherwise. Limiting a data breakpoint to a certain procedure also increases the speed because the check then only affects the given procedure and not the entire program.

Examining variables during runtime

The value of a variable can be very quickly viewed while the program is running by placing the mouse cursor over a variable in the sourcecode and waiting for a brief moment. If the variable is currently in scope and can be displayed, its value will be shown as a tooltip on the mouse location.



More complex expressions (for example array fields) can be viewed by selecting them with the mouse and placing the mouse cursor over the selection.



The debugger tools also offer a number of ways to examine the content of variables , arrays or linkedlists .

Errors in the Program

If the debugger encounters an error in your program, it will halt the execution, mark the line that contains the error (red background in the default colors) and display the error message in the error log and the status bar.

At this point, you can still examine the variables of your program, the callstack or the memory, however other features like the Register display or stack trace are not available after an error.

If the error is determined to be fatal (like an invalid memory access, or division by 0), you are not allowed to continue the execution from this point. If the error was reported by a PureBasic library, you are allowed to try to continue, but in many cases, this may lead to further errors, as simply continuing just ignores the displayed error.

After an error (even fatal ones), you have to use the "Kill Program" command to end the program and continue editing the source code. The reason why the program is not automatically ended is that this would not allow to use the other debugger features (like variable display) to find the cause of the error.

Note: you can configure the debugger to automatically kill the program on any error. See Customizing the IDE for that.

Debugger warnings

In some cases the debugger cannot be sure whether a given parameter is actually an error in the program or was specified like that on purpose. In such a case, the debugger issues a warning. By default, a warning will be displayed with file and line number in the error log and the line will be marked (orange in the default colors). This way the warnings do not go unnoticed, but they do not interrupt the program flow. There is also the option of either ignoring all warnings or treating all warnings like errors (stopping the program). The handling of debugger warnings can be customized globally in the Preferences or for the current compiled program in the Compiler options .

The Error log

The error log is used to keep track of the compiler errors, as well as the messages from the debugging. Messages are always logged for the file they concern, so when an error happens in an included file , this file will be displayed, and a message logged for it.

The "Error log" submenu of the Debugger menu provides functions for that:

Show error log
Shows / hides the log for the current source.

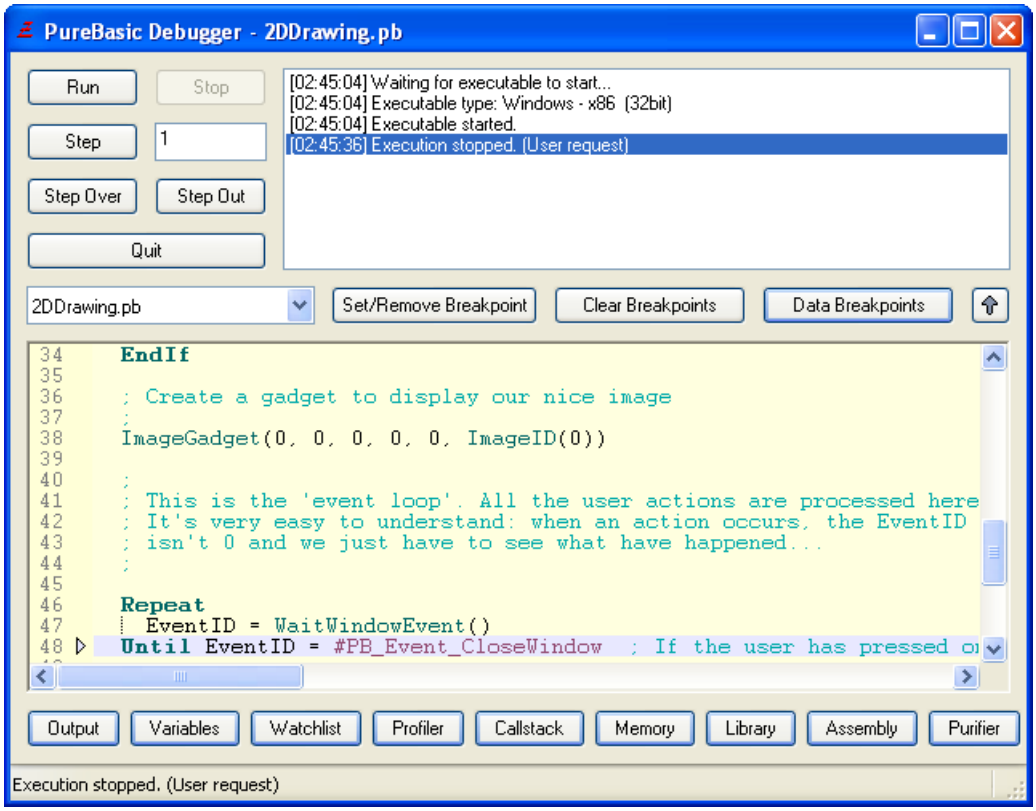
Clear log
Clears the log for this file.

Copy log
Copies the contents of the error log to the clipboard.

Clear Error marks
After you have killed the program, any error mark in the source file will remain. This is to help you identify the line that caused the problem and solve it. The "Clear error Marks" command can be used to remove these marks.

You can also configure the IDE to automatically clean the error marks when the program ends. See Configuring the IDE for that.

The Standalone Debugger



The standalone debugger is very similar to the one in the IDE, and will be explained here only briefly:

On the Debugger window, you have control buttons to carry out the basic program control, as described above. The "Step" button carries out as many steps as are set in the edit field next to it. Closing the Debugger with "Quit" or the close button will also kill the debugged program.

The Error log area can be hidden by the up arrow button on the right side in order to make the debugger window smaller.

The code view is used to display the currently executed code line as well as any errors or breakpoints. Use the combo box above it to select the included file to view. The "Set Breakpoint", "Remove Breakpoint" and "Clear Breakpoints" can be used to manage breakpoints in the currently displayed source file. The code view also provides the mouse-over feature from the integrated debugger to quickly view the content of a variable.

The debugger tools can be accessed from the buttons below the code area. Their usage is the same as with the integrated IDE debugger.

Note: The Standalone Debugger has no configuration of its own. It will use the debugger settings and coloring options from the IDE. So if you use a third-party Editor and the standalone debugger, you should run the IDE at least once to customize the Debugger settings.

Executing the standalone debugger from the commandline:
To execute a program compiled on the commandline with enabled debugger (-d or /DEBUGGER switch), call the debugger like this:

pbdebugger <executable file> <executable commandline>

If you execute a debugger-enabled executable from the commandline directly, it will only use the commandline debugger.

Remote debugging with the standalone debugger:

The network debugging feature allows the easy debugging of programs on remote servers or inside of virtual machines while still using a comfortable graphical interface instead of the commandline debugger. The compilation of the program has to be handled separately either on the remote machine or on the local machine before transfering the file to the target machine.

The debuggers between all operating systems and processor types supported by PureBasic are compatible as long as the PureBasic version between the debugger and the compiler that compiled the program matches. This means that a program running on Linux x64 can be debugged using an x86 Windows machine without problems for example. The debugger and the compiled executable can both act as either the client or the server for the network connection depending on the commandline parameters. One instance of the debugger can be used to debug one program only. Multiple connections are not possible.

Running the debugger in network mode:
The following commandline parameters control the network capabilities of the standalone debugger.

```
pbdebugger.exe /CONNECT=host[:port]  [/PASSWORD=password]
pbdebugger.exe /LISTEN[=interface][:port]  [/PASSWORD=password]
```

The "connect" mode connects the debugger as a client to an executable which must have been started as a server before. The "listen" mode creates a server and waits for an incomming connection from an executable. If an the IP address of a local network interface is specified, the server will only listen on that specific interface. Otherwise it will listen on all network interfaces for connections on the specified port. If no port is specified, the default port (port 10101) is used.

The password option enables encryption on the debugger data stream using AES. If the client is started without the password option but the server requires a password then you will be prompted to enter the password to establish a connection.

Running the executable in network mode:
The executable has to be compiled in debugger mode as usual (using the /DEBUGGER or –debugger compiler switch). It can then be started from the commandline with the following parameters to enable network mode. The rules for client and server mode are the same as above.

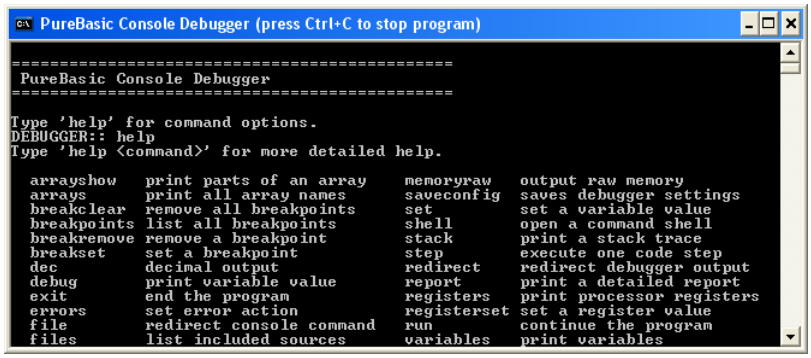
```
yourprogram.exe /DEBUGCONNECT=host[:port]  [/PASSWORD=password]
yourprogram.exe /DEBUGLISTEN[=interface][:port]  [/PASSWORD=password]
```

If commandline parameters cannot be used to start the program in network mode (for example because the program also reads its commandline parameters and would be confused by the debugger related options) there is an alternative way by setting the following environment variable before executing the program (case sensitive):

```
PB_DEBUGGER_Communication=NetworkClient;host[:port][;password]
PB_DEBUGGER_Communication=NetworkServer[;interface][:port][;password]
```

Once the network connection is established between debugger and executable, the debugging session works in the same way as local debugging. If the debugger is closed, the debugged executable is terminated as well. It is not possible to disconnect or reconnect the debugger after the connection was established.

The commandline debugger:



The commandline debugger is not a part of the IDE and therefore not explained in detail here.

While the program is running, hit Ctrl+C in the console to open a debugger console prompt. In this prompt type "help" to get an overview of all available commands. Type "help <commandname>" for a more detailed description of the command.

Debugging threaded programs:

To use the debugger with a program that creates threads , the 'Create threadsafe executable' Option must be set in the Compiler options , as otherwise the information displayed by the debugger concerning line numbers, errors, local variables and such could be wrong due to the multiple threads.

The following features and limitations should be considered when debugging a threaded program:

While the program is running, the variable viewer, callstack display or assembly debugger will display information on the main thread only. When the program is stopped, they display information on the thread they were stopped in. So to examine local variables or the callstack of a thread, the execution must be halted within that thread (by putting a breakpoint or a CallDebugger statement there). The various 'Step' options always apply to the thread where the execution was last stopped in.
If an error occours, the execution is halted within that thread, so any information displayed by the variable viewer or callstack display is of the thread that caused the error.
The watchlist only watches local variables of the main thread, not those of any additional running threads.

While the execution is stopped within one thread, the execution of all other threads is suspended as well.

Chapter 14

Included debugging tools

These tools provide many features to inspect your program while it is running. They can not be used while you are editing the source code. These tools are available in both the integrated Debugger and the standalone debugger. The console debugger provides many of these features too, but through a debugger console.

Some of the tools include the viewing of variables. Here is an explanation of the common fields of all these variable displays:

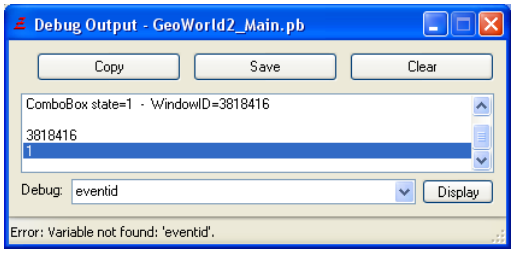
Scope
The scope of a variable is the area in which it is valid. It can be global , local , shared , static or threaded , depending on how it is used in your source code. 'byref' ("by reference", i.e. using the address) is used to indicate an Array or LinkedList that was passed as parameter to a procedure.

Variable type
The variable type is indicated through a colored icon:
B : Byte
A : Ascii
C : Character
W : Word
U : Unicode
L : Long
I : Integer
Q : Quad
F : Float
D : Double
S : String
Sn : Fixed length string

A Structure is either marked as a dot, or with an arrow. If marked with an arrow, it can be expanded by double-clicking on it to view the members of this structure. A down arrow marks an expanded structure. A Structure marked with a dot cannot be expanded (usually because it is just a structure pointer).

Dynamic arrays inside structures are shown with the dimensions they are currently allocated with. Lists and maps inside structures are shown with their size and their current element (if any).

The Debug output window



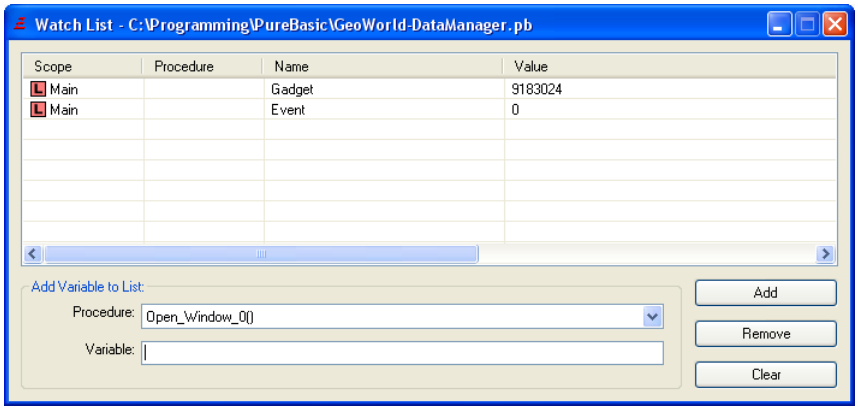
In this window, the output of the Debug statement will be displayed. The Debug statement is a quick and simply way to print messages for debugging purposes.

The debug window will automatically open at the first output is produced by your program. If you then close it, it will not automatically open on subsequent messages, however they will still be logged. You can copy this output to the clipboard or save it to a file. There is also a button to clear the messages from the window.

The entry field at the bottom of the window allows an expression to be entered, which will be evaluated and the result printed in the output box. This allows to quickly check the state of variables or array fields without the need to look them up in one of the debugger tools. The evaluation is started by pressing Enter or clicking on the "Display" button. If the expression cannot be evaluated for some reason, an errormessage is displayed in the statusbar.

The expression can be any valid PB expression (not including logical ones or containing PB keywords). It can contain variables , arrays , linked lists , constants and also some commands from the Math , Memory and String libraries.

The Watchlist



The watchlist can be used to track changes in variables of your program in real time, while the program is running. It can only display single variables (no full structures), however, these variables can be a part of structures. Elements of dynamic array, list or map inside structures can not be displayed in the watchlist.

To add a variable, select its procedure (if it is a local variable) or select "— Main —" if it is a global variable or part of an array or linked list . Then type the variable name, as you would access it in your source code into the Variable field and press Add.

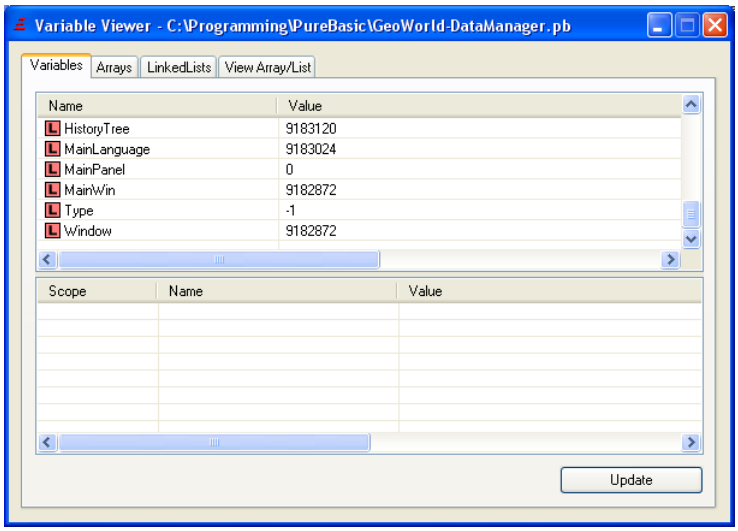
```
Examples:
MyVariable$                        - add a normal string variable
Array(1, 5)                        - add an array field
Structure\subfield[5]\value          add a variable inside a structure
MyLinkedList()\structuresubfield     add a variable inside a structured
 linkedlist
```

You can also add new watched variables from the VariableViewer, by right-clicking on them and selecting "add to watchlist"

In the list you will see the values of the watched variables. If the value is displayed as "—", it means that this variable is not valid at the current point in the source code. (for example if you watch a local variable, or a linked list element and the list has no current element)

The watched variables are remembered between the debugging sessions, and even saved with the compiler options, so you do not need to re-populate this list all the time.

The Variable Viewer



The Variable viewer allows to examine the program's variables, arrays , linked lists and maps . The individual tabs show global and threaded items in the top part and local , shared and static items in the bottom part.

The "Update" Button can be used to get the most recent data from the program. If the program is halted or in step mode, the content is updated on each step automatically. By right-clicking on any variable or Array/Linked list field, you can copy that variable, or add it to the watchlist for realtime-tracking of its value.

On Windows, the content of the Variable viewer can be sorted by name, scope or variable value by clicking on the header of the appropriate column.

The 'Variables' tab
This tab, shows the variables of the program. By right-clicking on a variable, it is possible to add it to the watchlist.

The 'Arrays' tab
This tab shows a list of all arrays in the program and the dimensions in which they are currently defined. (-1 means that Dim was not called yet) By right-clicking on an array, the content of the array can be viewed in the "View Array/List/Map" tab.

The 'Linked Lists' tab
This tab shows a list of all Linked Lists, the number of elements they currently hold ("-" indicates that NewList was not called yet), as well as the index of the current element of the list. ("-" indicates that there is no current element) By right-clicking on a list, the content of the list can be viewed in the "View Array/List/Map" tab.

The 'Maps' tab
This tab shows a list of all maps, the number of elements they currently hold ("-" indicates that NewMap was not called yet), as well as the key of the current element of the map. ("-" indicates that

there is no current element) By right-clicking on a map, the content of the map can be viewed in the "View Array/List/Map" tab.

The 'View Array/List/Map' tab
This tab can be used to view individual entries of an array, a linked list or a map. This includes arrays, lists or maps inside structures as well. To do so enter the name of the array, map or list including a "()" at the end, select what kind of items to display and press "Display". Note that the content of the display is not automatically updated when the program is in step mode.

"Display all items" simply displays everything. "Display Non-zero items only" will only display those items that do not hold the value 0 or an empty string. This makes viewing large arrays/lists with only few valid items in them simpler. A structure is considered "zero" if all of its items either hold the value 0 or an empty string.

"Display Range" allows to display only a specific range of an array, a list or a map. The range can be given for each Array dimension individually, separated by commas. If one dimension is not specified at all, all of its items will be displayed. Here are a few examples for valid range input:

```
"1-2, 2-5, 2" : first index between 1 and 2, a second index between 2
 and 5 and a third index of 2.
"1, 2-5"      : first index of 1 and a second index between 2 and 5.
"1, , 5"      : first index of 1, any second index and a third index
 of 5.
```

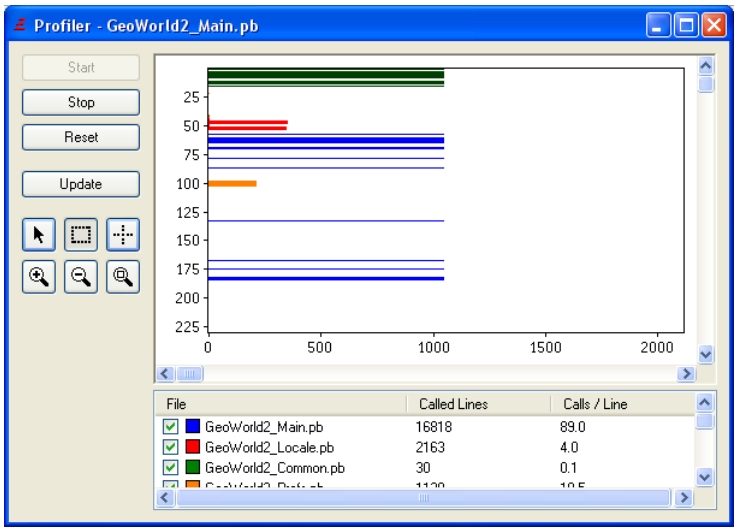
For list display, the "Display Range" option can be used to display a range of list elements by their index (zero based).

```
"0"     : first element
"1-3"   : second to the fourth element
```

For map display, the "Display Range" option can be used to filter the keys to be displayed. It has to contain a mask for the key string of the map elements (no quotation marks). A "?" matches one character, a "*" matches any number of characters. Here are a few examples for valid mask input:

```
"hat" : matches only the item with "hat" as key.
"?at" : matches items with "hat", "bat" etc as key.
"h*t" : matches items with keys that start with "h" and end with "t"
 and anything inbetween.
```

The Profiler



The profiler tool can count how often each line in the sourcecode is executed. This collected data can be used to identify which portions of the code are used most often and where improvements make most

sense. It also helps to identify problems where a piece of the code is executed too often as a result of an error.

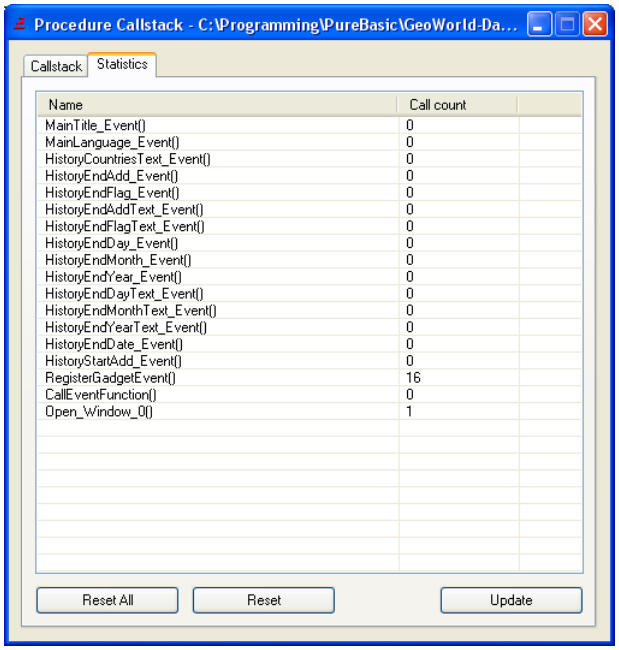
Recording the data
The recording of the data can be controlled by the Start, Stop and Reset (to set all counts to 0) buttons in the profiler window. The Update button can be used to update the graph while the program is running. Each time the program is halted or in step mode, the data is updated automatically. By default, the profiler is recording data from the start of the program. This can be changed in the Preferences .

Examining the data
The recorded data is displayed as a graph, with the vertical axis showing the source line number and the horizontal axis showing how often the lines were executed. If the running program consists of more than one source file, a list of source files is presented below the graph. To display the graph for a file either select it, or check its checkbox. Multiple files can be shown in the graph at once to better compare them. Right-clicking on one of the filenames allows to change the color used to display that file in the graph.

Mouse modes in the graph
Right-clicking inside the graph brings up a popupmenu which allows to zoom in or out or to show the source line that was clicked on in the IDE or code display of the debugger. The action taken by a left-click can be controlled by the buttons on the left side:

- Arrow button: Left-clicking and dragging allows to scroll the graph display.
- Box button: Left-clicking and dragging allows to select an area which will be zoomed in.
- Cross button: While this button is activated, moving the mouse on the graph displays a crosshair to better identify the line and call count under the mouse.
- Zoom buttons: These buttons allow to zoom in/out and zoom out so all lines can be viewed.

The Callstack viewer

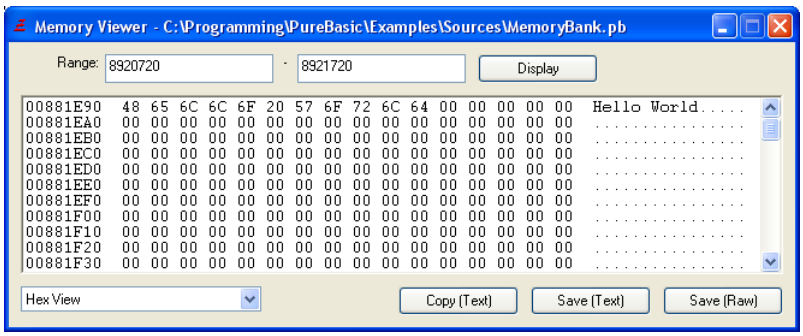


The callstack viewer shows which nested procedure calls led to the current position in the code. Each entry in the list means one procedure that is currently open. It shows the line and file from which it was called, and the arguments that were used to call the procedure. By clicking on the Variables button for each procedure, you can look at the variables of that instance of the procedure.

This allows to easily trace, from which part of the code, a procedure was called. The callstack view does only automatically update when you stop the program, or use Step to execute single lines. While the program is running, you have to use the Update button to update the view for the current code position.

The "Statistics" tab shows the number of times each procedure in the code was called. You can reset the count for all procedures with "Reset all", or for the currently marked entry with the "Reset" button. Like with the callstack, the updates are not automatic while the program is not stopped. Use the Update button for that.

The Memory Viewer



The memory viewer lets you view a memory area in your program. The range to view can be entered into the range fields as any valid PB expression. (This can be a normal decimal value, a hex number preceeded by the '$' character or any other valid expression, including variables or pointers from the code). If the content of the second range field starts with a "+" sign, the content is interpreted as relative to the first field.

Example: "*Buffer + 10" to "+30" will display the 30 bytes of memory starting at the location 10 bytes after what *Buffer points to.

If the memory area is valid for viewing, it will be displayed in the area below. If parts of the area are not valid for reading, you will get an error message. The type of display can be changed with the combo box in the lower left corner. The following view modes are available:

Hex View
The memory will be displayed like in any hex viewer, giving the memory location in hex display on the left, followed by the hexadecimal byte values, and then the string representation in the right column.

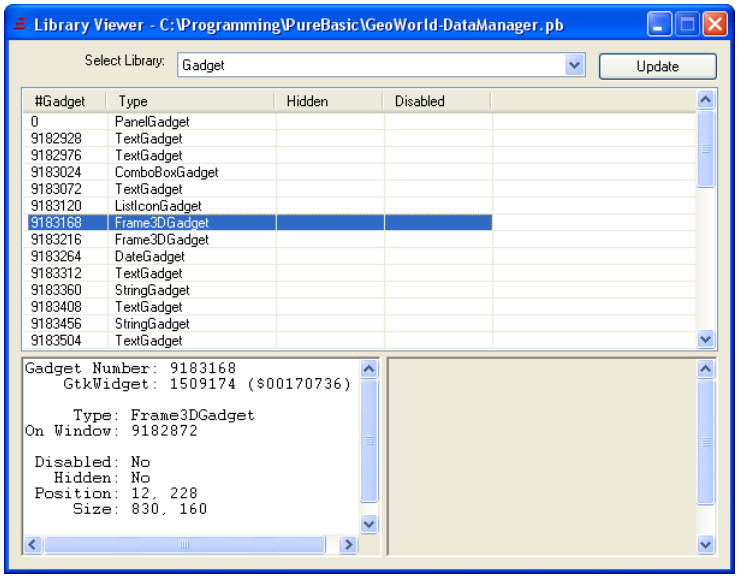
Byte/Character/Word/Long/Quad/Float/Double table
The memory area will be shown as a table of the specified variable type. Whether or not this table is single-column or multi-column can be set in the Preferences. (see Configuring the IDE)

String view
This displays the memory area as a string, with any non-string characters displayed in [] (for example "[NULL]" for the 0 byte.) A linebreak is added after newline characters and [NULL] to improve the readability of the output. The memory area can be interpreted as an Ascii, Unicode or Utf8 string.

You can also export the viewed memory area from the memory viewer:

Copy (Text): Copies the displayed area as text to the clipboard.
Save (Text): Saves the displayed area as text to a file.
Save (Raw): Saves the memory area as raw binary to a file.

The Library Viewer



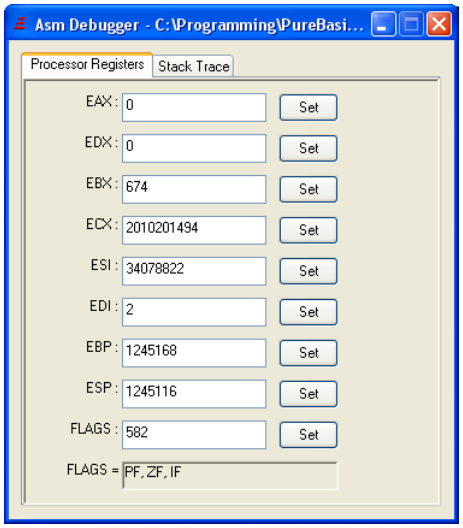
The Library Viewer provides information about the objects that were created by certain libraries. It allows to quickly check which images are currently loaded in the program, or which gadgets are created.

Once the program is started, the combobox on top of the window can be used to select the library to view. The list below will then show all objects of the library that currently exist in the executable along with some additional information on each object. The "Update" button will update this list of objects. Selecting an object in the list will display more detailed information about it in the text area on the left, and if supported by the library also a visual display of the object on the right. (for Images, Sprites, ...)

If the combobox displays "No Information", this means that your executable has used no library that supports this feature.

Currently, the Library Viewer is supported by the following libraries:
Thread
Gadget
Window
File
Image
Sprite
Sprite3D
XML

The Assembly Debugger



The asm debugger is provided for advanced programmers to examine and change the CPU register content and to examine the programs stack for debugging of inline asm code.

The Processor Register view is only available while the program execution is halted. By changing any of the register values and clicking "Set", you can modify the value in that register.

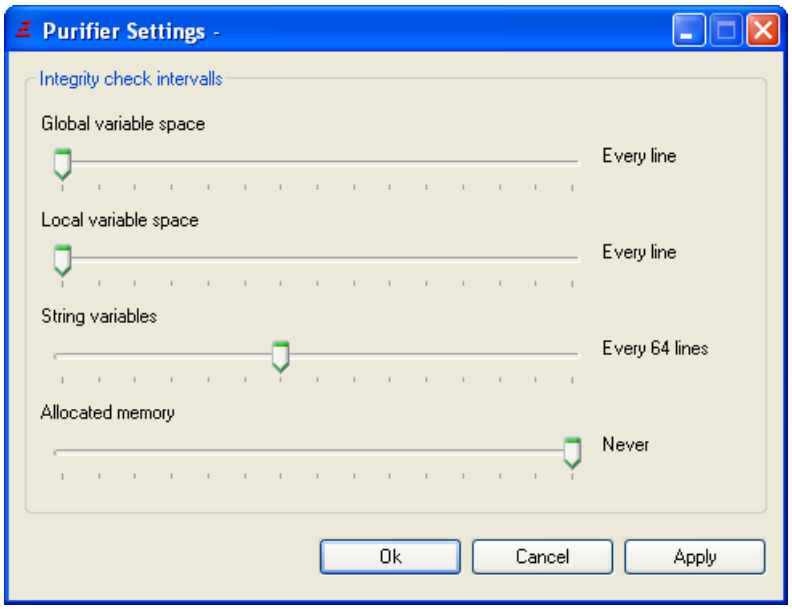
The Stack trace shows the content of the programs stack given in relation to the ESP register. If the current stack position is not aligned at a 4 byte boundary, there can be no information given about the content of the stack. In this case the stack is shown as a hex display.

If the stack pointer is properly aligned, the stack contents are displayed with comments about the meaning of the contained values. (detailing the pushed registers and other values on a PB procedure call.)

The stack trace is updated automatically when you stop the execution or step though the program, unless you specify otherwise in the Preferences . If you disable the automatic update, there will be an "Update" button displayed to do so manually.

Note: The Assembly debugger is currently not available on MacOSX.

The Purifier



The purifier can detect memory errors such as writing past the end of an allocated memory buffer or string. Without the purifier some of these mistakes would lead to crashes and others would go unnoticed because the write operation overwrites some other valid memory.

The purifier requires special output from the compiler to work, which is why it is only available if the "Enable Purifier" option is set in the compiler options when the program is compiled.

The purifier detects these errors by placing a special 'salt'-value around global and local variables, strings and allocated memory buffers. These salt values are then checked at regular intervalls and an error is displayed if they were changed. These checks slow down the program execution considerably especially for large programs, which is why the rate at which the checks are performed can be specified in the purifier window:

Global variable space
Defines the intervall in source lines after which the global variables are checked.

Local variable space
Defines the intervall in source lines after which the local variables are checked.

String variables
Defines the intervall in source lines after which the memory used by string variables is checked.

Allocated memory
Defines the intervall in source lines after which the memory allocated by AllocateMemory() is checked.

Chapter 15

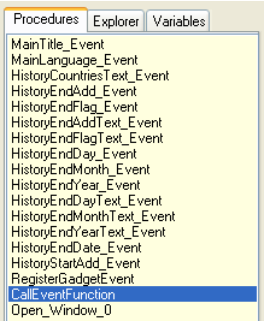
Using the buildin Tools

The PureBasic IDE comes with many building tools, to make programming tasks easier and increase your productivity. Many of them can be configured to be either accessible from the Menu as separate windows, or to be permanently displayed in the Panel on the side of the editing area.

For information on how to configure these tools and where they are displayed, see Configuring the IDE .

Tools for the Side Panel Area

Procedure Browser



This tool displays a list of all procedures and macros declared in the current source code. By double-clicking on an entry in that list, the cursor automatically jumps to that procedure.

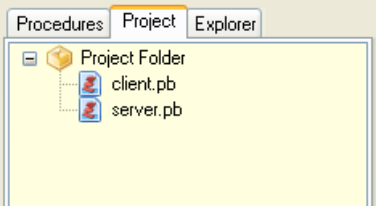
Macros will be marked in the list by a "+" sign before the name.

You can also place special comment marks in your code, that will be displayed in the list too. They look like this: ";- <description>". The ; starts a comment, the - that follows it immediately defines such a mark.

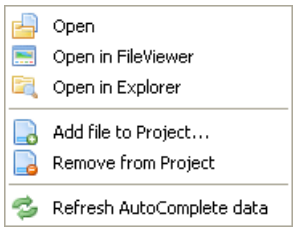
The description will be shown in the Procedure list, and clicking on it will jump to the line of this mark. Such a comment mark can be distinguished from a Procedure by the "> " that is displayed before it in the procedure list.

The list of procedures can be sorted, and it can display the procedure/macro arguments in the list. For these options, see Configuring the IDE .

Project Panel

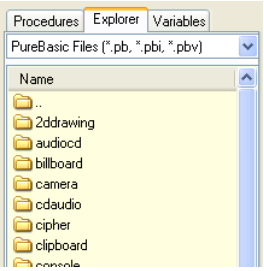


This tool displays a tree of all files in the current project . A double-click on a file opens it in the IDE. This allows fast access to all files in the project. A right-click on a file opens a context menu which provides more options:



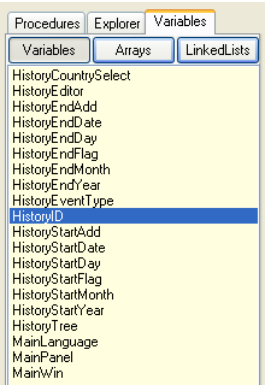
Open - Open the file in the IDE.
Open in FileViewer - Open the file in the FileViewer of the IDE.
Open in Explorer - Open the file in the operating systems file manager.
Add File to Project - Add a new file to the project.
Remove File from Project - Remove the selected file(s) from the project.
Refresh AutoComplete data - Rescan the file for AutoComplete items.

Explorer



The Explorer tool displays an explorer, from which you can select files and open them quickly with a double-click. Text files will be loaded into the edit area and binary files will be displayed in the internal File Viewer.

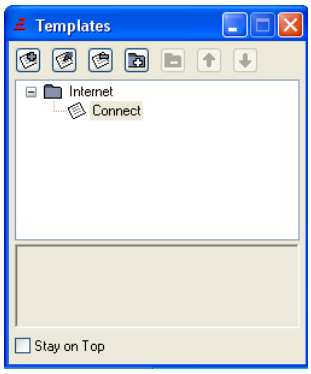
Variable Viewer



The variable viewer can display variables , Arrays , Linked Lists , Constants , Structures and Interfaces defined in your source code, or any currently opened file. You can configure what exactly it should display in the preferences .
Note: The displaying of variables is somewhat limited for now. It can only detect variables that were

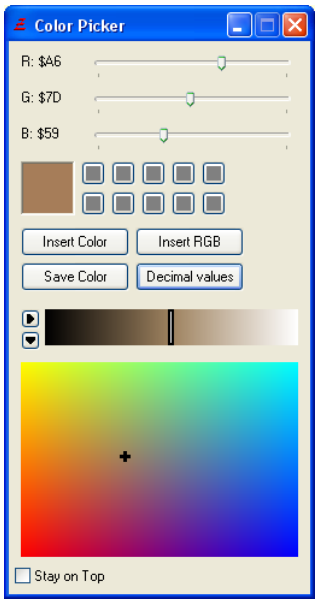
explicitly declared by Define , Global , Shared , Protected or Static .

Code Templates



The templates tool allows you to manage a list of small code parts, that you can quickly insert into your source code with a double-click. It allows you to manage the codes in different directories, and put a comment to each code. This tool is perfect to manage small, often used code parts.

Color Picker



The color picker helps you to find the perfect color value for whatever task you need. You can select a color from the area at the bottom and then modify its brightness with the slider above. The two little arrow buttons can be use to reset either selection to the center.

Additionally you can select the color with the sliders on the top. Note that modifying the sliders does not change the selection in the color area, but the other way around it works.

The "Insert Color" button inserts the color value as hex into the source code (i.e. "$FF00FF"). The "Insert RGB" button inserts it in the form of the Rgb() function. (i.e. "RGB(255, 0, 255)"). The "Decimal values" button determines whether the values next to the sliders on top are displayed as hex or decimal.

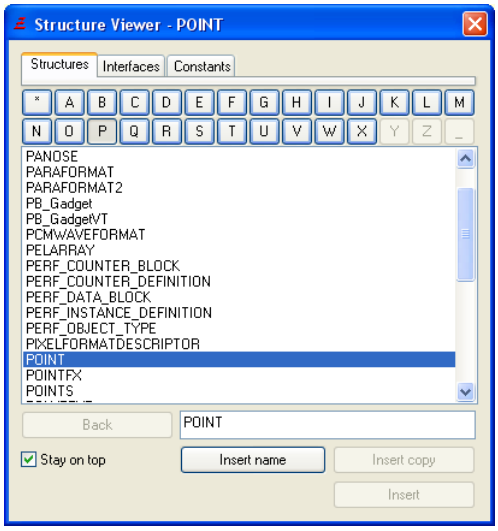
The "Save color" button saves the color in the history next to the field showing the current color. The color will be added at the top-left field, and all others are moved by one spot. Clicking on any of these saved color values sets them as the current color again.

Ascii Table

The Ascii table tool displays a table showing all the Ascii characters, together with their index in decimal and hex, as well as the corresponding html notation. By double-clicking on any line, this character will be inserted into the source code. With the buttons on the bottom, you can select which column of the table to insert on a double-click.

Other buildin tools

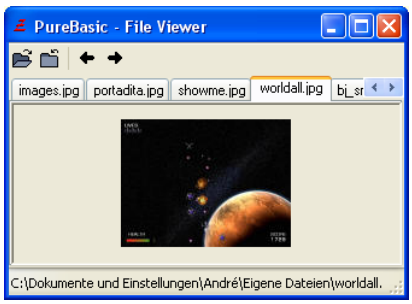
Structure Viewer



The structure viewer allows you to view all the Structures, Interfaces and constants, that are predefined in PureBasic. Double-clicking on a Structure or Interface shows the declaration. On top of the list you can select a filter to display only entries that start with a given character.

The "Back" button navigates back through the viewed entries.
"Insert name" inserts just the name of the selected entry.
"Insert copy" inserts a copy of the declaration of that entry.
"Insert" lets you enter a variable name and then inserts a definition of that variable and the selected entry and all elements of it.

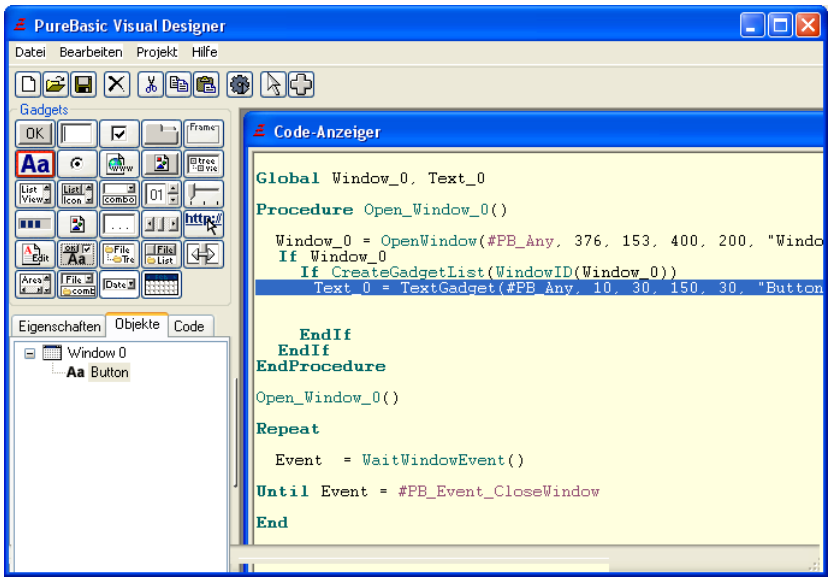
File Viewer



The internal file viewer allows you do display certain types of files. Text files, images and web pages (windows only). Any unknown file type will be displayed in a hex-viewer. The "Open" button opens a new file, the "X button" closes it and the arrows can be used to navigate through the open files.
Also any binary file that you attempt to open from the Explorer tool, or by double-clicking on an IncludeBinary keyword will be displayed in this file viewer.

Other entries in the Tools menu

Visual Designer



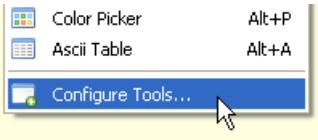
The PureBasic Visual Designer can be used to design the user interface for you application. It is not a tool of the IDE, but a separate program. It also has its own help file. See the "VisualDesigner.chm" file in your PureBasic directory for more information.

Chapter 16

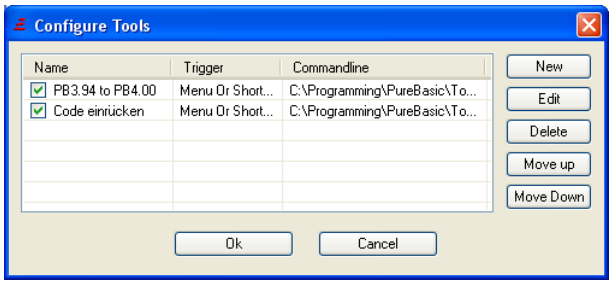
Using external tools

The PureBasic IDE allows you to configure external programs to be called directly from the IDE, through the Menu, Shortcuts, the Toolbar, or on special "triggers". The use of this is to make any other program you use while programming easily accessible.

You can also write your own little tools in PureBasic that will perform special actions on the source code you are currently viewing to automate common tasks. Furthermore, you can configure external file viewers to replace the internal File Viewer of the IDE for either specific file types or all files.



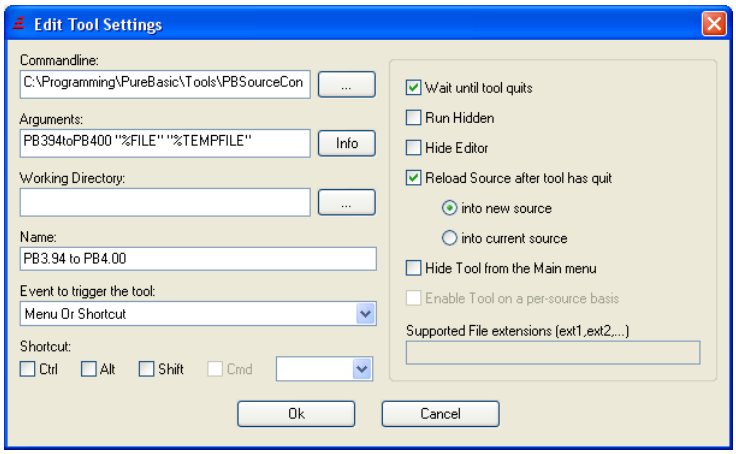
With the "Config tools" command in the Tools menu, you can configure such external tools. The list you will see displays all the configured tools in the order they appear in the Tools menu (if not hidden). You can add and remove tools here, or change the order by clicking "Move Up"/"Move Down " after selecting an item.



Any tool can be quickly enabled or disabled from the "Config tools" window with the checkbox before each tool entry. A checked checkbox means the tool is enabled, an unchecked one means it is currently disabled.

Configuring a tool

The basic things you need to set is the commandline of the program to run, and a name for it in the Tools list/Menu. Everything else is optional.



Commandline
Select the program name to execute here.

Arguments
Place commandline arguments that will be passed to the program here. You can place fixed options, as well as special tokens that will be replaced when running the program:

%PATH : will be replaced with the path of the current source code. Remains empty if the source was not saved.
%FILE : filename of the current source code. Remains empty if it has not yet been saved. If you configure the tool to replace the file viewer, this token represents the file that is to be opened.
%TEMPFILE : When this option is given, the current source code is saved in a temporary file, and the filename is inserted here. You may modify or delete the file at will.
%COMPILEFILE : This token is only valid for the compilation triggers (see below). This is replaced with the temporary file that is sent to the compiler for compilation. By modifying this file, you can actually change what will be compiled.
%EXECUTABLE : This will be replaced by the name of the executable that was created in with the last "Create Executable". For the "After Compile/Run" trigger, this will be replaces with the name of the temporary executable file created by the compiler.
%CURSOR : this will be replaced by the current cursor position. In the form of LINExCOLUMN.
%SELECTION : this will be replaced by the current selection in the form of LINESTARTxCOLUMNSTARTxLINEENDxCOLUMNEND. This can be used together with %TEMPFILE, if you want your tool to do some action based on the selected area of text.
%WORD : contains the word currently under the cursor.

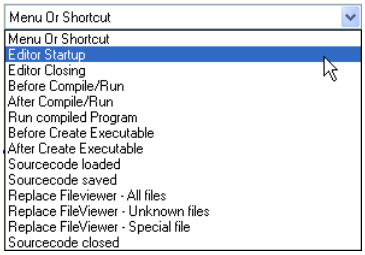
Note: for any filename or path tokens, it is generally a good idea to place them in "" (i.e. "%TEMPFILE") to ensure that also paths with spaces in them are passed correctly to the tool. These tokens and a description can also be viewed by clicking the "Info" button next to the Arguments field.

Working Directory
Select a directory in which to execute this tool. By specifying no directory here, the tool will be executed in the directory of the currently open source code.

Name
Select a name for the tool. This name will be displayed in the tools list, and if the tool is not hidden from the menu, also in the Tools menu.

Event to trigger the tool

Here you can select when the tool should be executed. Any number of tools can have the same trigger, they will all be executed when the trigger event happens. The order of their execution depends on the order they appear in the tools list.



Menu Or Shortcut
The tool will not be executed automatically. It will be run by a shortcut or from the Menu. Note: to execute a tool from the Toolbar, you have to add a button for it in the Toolbar configuration in the Preferences (see Configuring the IDE for more).

With this trigger set, the "Shortcut" option below becomes valid and lets you specify a shortcut that will execute this tool.

Editor Startup
The tool will be executed right after the IDE has been fully started.

Editor End
The tool will be executed right before the IDE ends. Note that all open sources have already been closed at this time.

Before Compile/Run
The tool will be executed right before the compiler is called to compile a source code. Using the %COMPILEFILE token, you can get the code to be compiled and modify it. This makes it possible to write a small pre-processor for the source code. Note that you should enable the "Wait until tool quits" option if you want your modifications to be given to the compiler.

After Compile/Run
The tool will be executed right after the compilation is finished, but before the executable is executed for testing. Using the %EXECUTABLE token, you can get access to the file that has just been created. Note that you can modify the file, but not delete it, as that results in an error message when the IDE tries to execute the file.

Run compiled Program
The tool will be executed when the user selects the "Run" command from the compiler menu. The tool is executed before the executable is started. The %EXECUTABLE token is valid here too.

Before create Executable
The same as for the "Before Compile/Run" trigger applies here too, only that the triggering event is when the user creates the final executable.

After create Executable
The tool is executed after the compilation to create the final executable is complete. You can use the %EXECUTABLE token to get the name of the created file and perform any further action on it.

Sourcecode loaded
The tool is executed after a source code has been loaded into the IDE. The %FILE and %PATH tokens are always valid here, as the file was just loaded from the disk.

Sourcecode saved
The tool will be executed after a source code in the IDE has been saved successfully. The %FILE and %PATH tokens are always valid here, as the file has just been saved to disk.

Sourcecode closed
The tool will be executed whenever a source file is about to be closed. At this point the file is still there, so you can still get its content with the %TEMPFILE token. %FILE will be empty if the file was never saved.

File Viewer All Files
The tool will completely replace the internal file viewer. If an attempt is made in the IDE to open a file that cannot be loaded into the edit area, the IDE will first try the tools that have a trigger set for the specific file type, and if none is found, the file will be directed to this tool. Use the %FILE token to get the filename of the file to be opened.

Note: Only one tool can have this trigger. Any other tools with this trigger will be ignored.

File Viewer Unknown file
This tool basically replaces the hex viewer, which is usually used to display unknown file types. It will be executed, when the file extension is unknown to the IDE, and if no other external tool is configured to handle the file. (If a tool is set with the "File Viewer All Files" trigger, than this tool will never be called.)

Note: Only one tool can have this trigger set.

File Viewer Special file
This configures the tool to handle specific file extensions. It has a higher priority than the "File Viewer All files" or "File Viewer Unknown file" triggers and also higher than the internal file viewer itself. Specify the extensions that the tool should handle in the edit box on the right. Multiple extensions can be given.

A common use for this trigger is for example to configure a program like Acrobat Reader to handle the "pdf" extension, which enables you to easily open pdf files from the Explorer, the File Viewer, or by double-clicking on an Includebinary statement in the source.

Other options on the right side

Wait until tool quits
The IDE will be locked for no input and cease all its actions until you tool has finished running. This option is required if you want to modify a source code and reload it afterwards, or have it passed on to the compiler for the compilation triggers.

Run hidden
Runs the program in invisible mode. Do not use this option for any program that might expect user input, as there will be no way to close it in that case.

Hide editor
This is only possible with the "wait until tool quits" option set. Hides the editor while the tool is running.

Reload Source after the tool has quit
This is only possible with the "wait until tool quits" option set, and when either the %FILE or %TEMPFILE tokens are used in the Arguments list.

After your program has quit, the IDE will reload the source code back into the editor. You can select whether it should replace the old code or be opened in a new code view.

Hide Tool from the Main menu
Hides the tool from the Tools menu. This is useful for tools that should only be executed by a special trigger, but not from the menu.

Enable Tool on a per-source basis
Tools with this option set will be listed in the "Execute tools" liste in the compiler options , and only executed for sources where it is enabled there. Note that when disabling the tool with the checkbox here in the "Config tools" window, it will be globally disabled and not run for any sourcecode, even if enabled there.

This option is only available for the following triggers:
- Before Compile/Run
- After Compile/Run

- Run compiled Program
- Before create Executable
- After create Executable
- Sourcecode loaded
- Sourcecode saved
- Sourcecode closed

Supported File extensions
Only for the "File Viewer Special file" trigger. Enter the list of handled extensions here.

Tips for writing your own code processing tools

The IDE provides additional information for the tools in the form of environment variables. They can be easily read inside the tool with the commands of the Process library .

This is a list of provided variables. Note that those that provide information about the active source are not present for tools executed on IDE startup or end.

```
PB_TOOL_IDE          - Full path and filename of the IDE
PB_TOOL_Compiler     - Full path and filename of the Compiler
PB_TOOL_Preferences  - Full path and filename of the IDE's Preference
 file
PB_TOOL_Language     - Language currently used in the IDE
PB_TOOL_FileList     - A list of all open files in the IDE, separated
 by Chr(10)

PB_TOOL_Debugger     - These variables provide the settings from the
 Compiler Options

PB_TOOL_InlineASM      window for the current source. They are set to
 "1" if the option
PB_TOOL_Unicode        is enabled, and "0" if not.
PB_TOOL_Thread
PB_TOOL_XPSkin
PB_TOOL_OnError

PB_TOOL_SubSystem    - content of the "Subsystem" field in the
 compiler options
PB_TOOL_Executable   - same as the %COMPILEFILE token for the
 commandline
PB_TOOL_Cursor       - same as the %CURSOR token for the commandline
PB_TOOL_Selection    - same as the %SELECTION token for the commandline
PB_TOOL_Word         - same as the %WORD token for the commandline

PB_TOOL_MainWindow   - OS handle to the main IDE window
PB_TOOL_Scintilla    - OS handle to the Scintilla editing component of
 the current source
```

When the %TEMPFILE or %COMPILEFILE tokens are used, the IDE appends the compiler options as a comment to the end of the created temporary file, even if the user did choose to not save the options there when saving a source code.

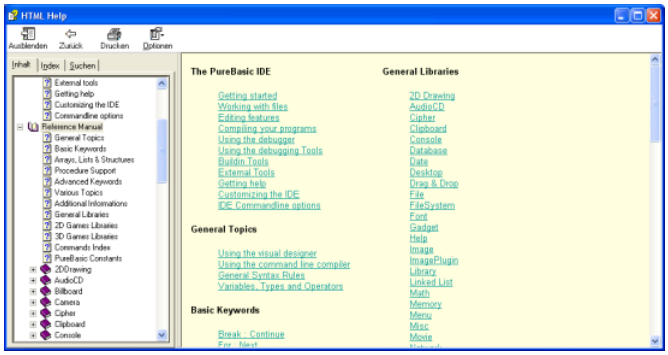
This enables your tool to read the compiler settings for this file, and take them into account for the actions your carries out.

Chapter 17

Getting Help

The PureBasic IDE provides ways to access the PureBasic helpfile, as well as other files and documentation you want to view while programming.

Quick access to the reference guide



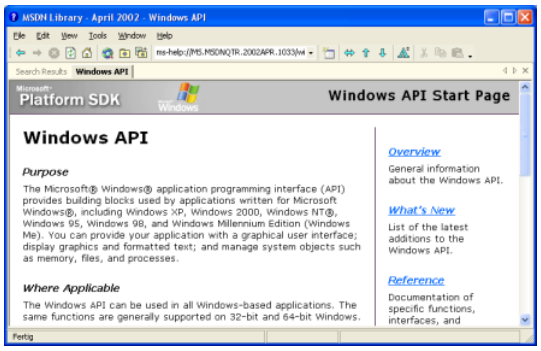
By pressing the help shortcut (F1 by default) or selecting the "Help..." command from the Help menu while the mouse cursor is over a PureBasic keyword or function, the help will be opened directly at the description of that keyword or function.

If the word at the cursor position has no help entry, the main reference page will be displayed.

Quick access to Windows API help

There are two ways to get the same quick access as for the PureBasic functions (by pressing F1 with the cursor on the function) also for functions of the Windows API. To enable this feature you have to download additional help files for these functions:

Microsoft Platform SDK



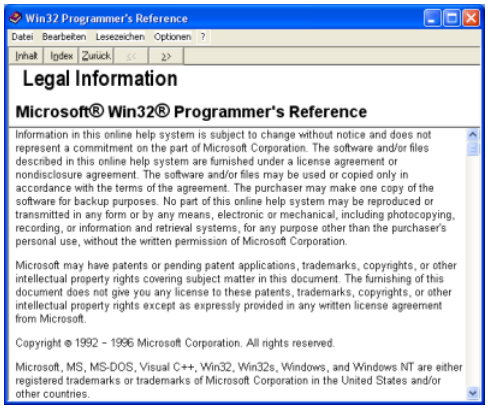
The Microsoft Platform SDK provides the most complete and up to date programming reference available for the windows platform. It provides information on all api functions, as well as overviews and introductions to the different technologies used when programming for the windows platform. It is however quite big (up to 400MB depending on the selected components).

For the IDE help, you can either install the "February 2003" or the "Windows Server 2003 SP1" edition of the SDK.

It can be downloaded from here:
http://www.microsoft.com/msdownload/platformsdk/sdkupdate/

Note that the SDK can also be ordered on CD from this page. Also if you are the owner of any development products from Microsoft (such as Visual Studio), there might be a copy of the SDK included on one of the CDs that came with it.

The win32.hlp helpfile



There is a much smaller alternative to the complete SDK by microsoft (7.5 MB download). This help is quite old (written for Windows95 in fact), so it does not provide any information on new APIs and technologies introduced since then.

However, it provides good information about commonly used APIs, that is still valid today, as these mostly did not change. This download is recommended if you only need occasional help for API functions, but do not want to download the full SDK.

It can be downloaded from here:
http://www.purebasic.com/download/WindowsHelp.zip

To use it from the PureBasic IDE, just create a "Help" subdirectory in your purebasic folder and copy the "win32.hlp" file into it.

Accessing external helpfiles from the IDE

If you have other helpfiles you wish to be able to access from the IDE, then create a "Help" subdirectory in your purebasic folder and copy them to it. These files will appear in the "External Help" submenu of the Help menu, and in the popupmenu you get when right-clicking in the editing area. Chm and Hlp files will be displayed in the MS help viewer. The IDE will open the helpfiles in the internal fileviewer. So files like text files can be viewed directly like this. For other types, you can use the Config Tools menu to configure an external tool to handle the type of helpfile you use. The help will then be displayed in that tool.

For example, if you have pdf helpfiles, configure an external tool to handle pdf files and put the files in the Help subdirectory of PureBasic. Now if you click the file in the "external help" menu, it will be opened in that external tool.

Chapter 18

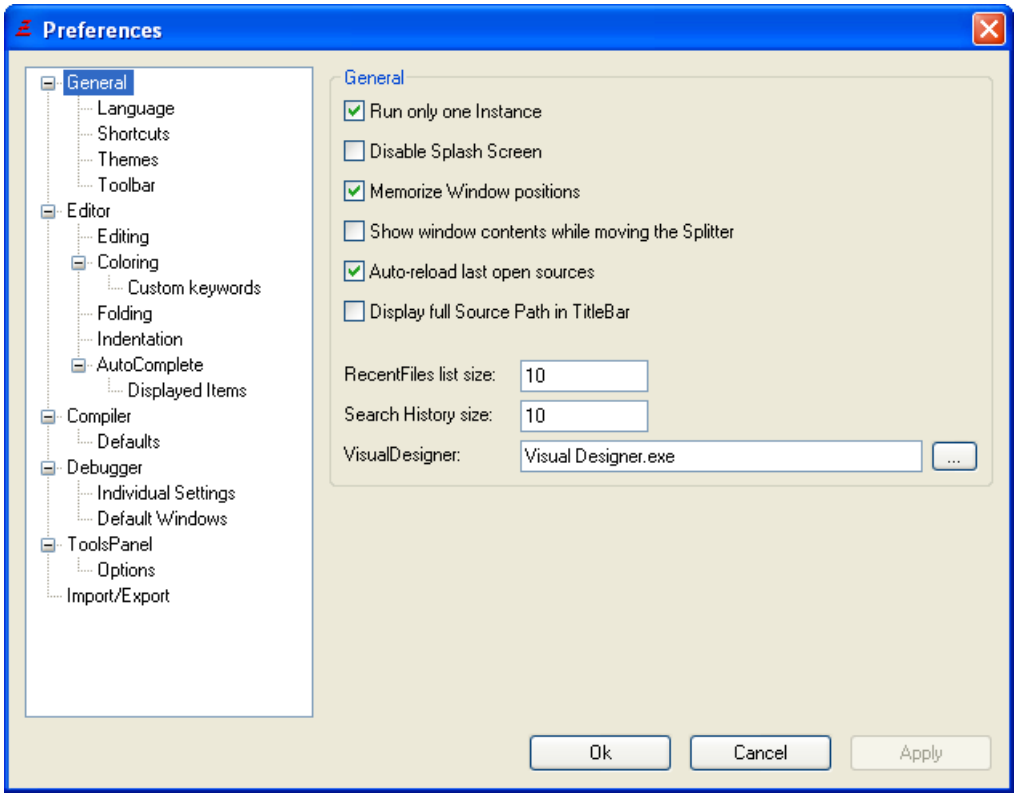
Customizing the IDE

The PureBasic IDE provides many options to customize or disable some of its features in order to become the perfect tool for you.

These options are accessible from the Preferences command in the File menu, and the meaning of each setting is described here.

Any changes made will only take effect once you click the ”OK” button or ”Apply”.

General



Options that affect the general behaviour of the IDE.

Run only one Instance
If set, prevents the IDE from being opened more than once. Clicking on a PB file in the explorer will open it in the already existing IDE instance instead of opening a new one.

Disable Splash screen
Disables the splash screen that is displayed on start-up.

Memorize Window positions
Remembers the position of all IDE windows when you close them. If you prefer to have all windows open at a specific location and size, enable this option, move all windows to the perfect position, then restart the IDE (to save all options) and then disable this option to always open all windows in the last saved position.

Show window contents while moving the Splitter
Enable this only if you have a fast computer. Otherwise moving the Splitter bar to the Error Log or Tools Panel may flicker a lot.

Auto-Reload last open sources
On IDE start-up, opens all the sources that were open when the IDE was closed the last time.

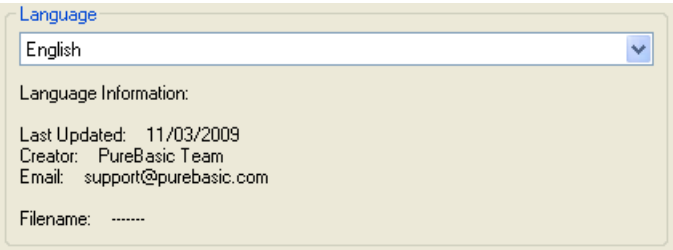
Display full Source Path in Title bar
If set, the IDE title bar will show the full path to the currently edited file. If not, only the filename is shown.

Recent Files list
This setting specifies how many entries are shown in the "Recent Files" submenu of the File menu.

Search History size
This setting specifies how many recent search words are remembered for "Find/Replace" and "Find in Files"

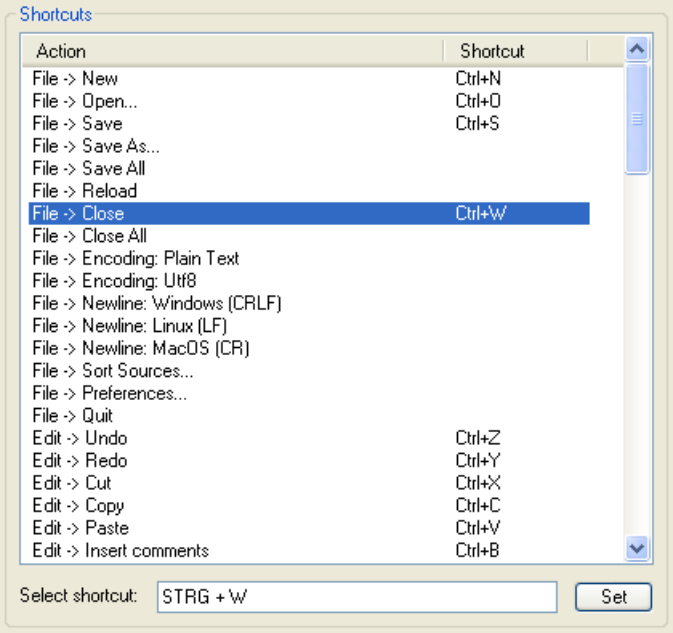
Visual Designer
This setting allows to set a different visual designer program when selecting the Menu command or toolbar icon.

General - Language



This allows you to change the language of the IDE. The combo box show the available languages, and you can view some informations about the language file. (for example who translated it and its last time of update.)

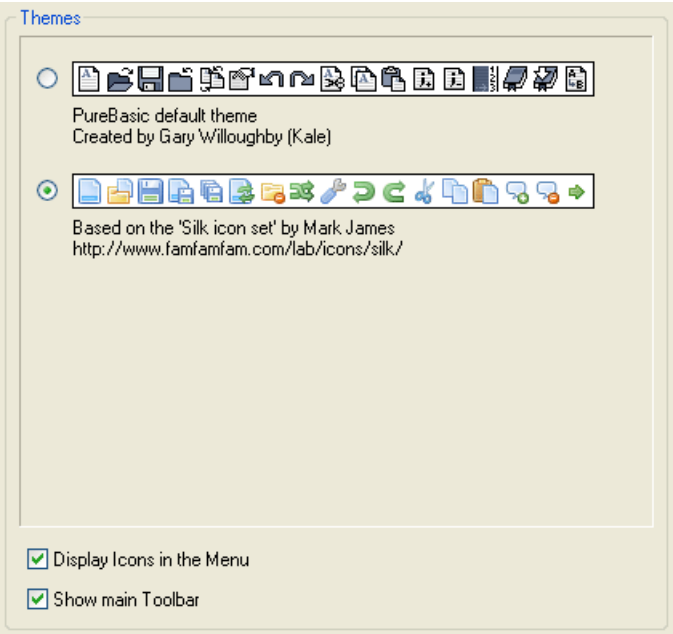
General - Shortcuts



Here you can fully customize all the shortcut commands of the IDE. Select an entry from the list, select the shortcut field, enter the new key combination and click "Set" to change the entry.

Note that Tab & Shift+Tab are reserved for block-indentation and unindentation and cannot be changed. Furthermore some key combination might have a special meaning for the OS and should therefore not be used.

General - Themes



This section shows the available icon themes for the IDE and allows to select the theme to use. The IDE comes with two themes by default.

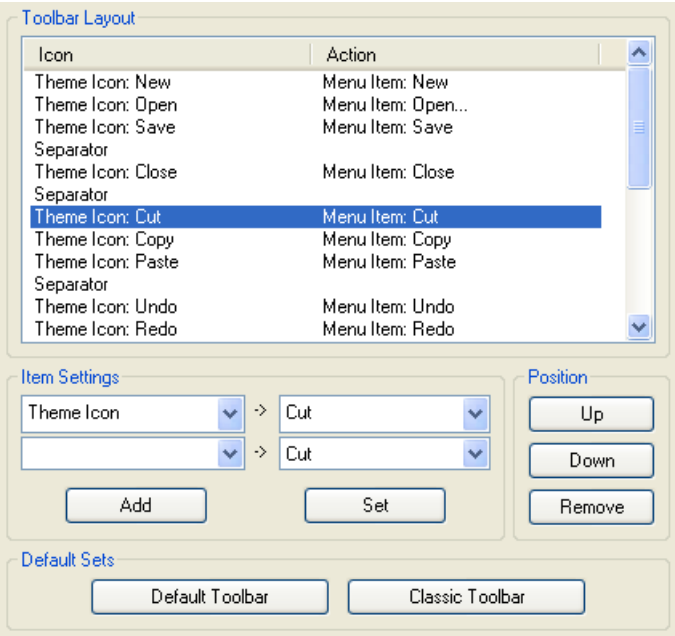
More themes can be easily added by creating a zip-file containing the images (in png format) and a "Theme.prefs" file to describe the theme. The zip-file has to be copied to the "Themes" folder in the PureBasic installation directory to be recognized by the IDE. The "SilkTheme.zip" file can be used as an

example to create a new theme.

Display Icons in the Menu
Allows to hide/show the images in the IDE menus.

Show main Toolbar
Allows to hide/show the main toolbar in order to gain space for the editing area.

General - Toolbar



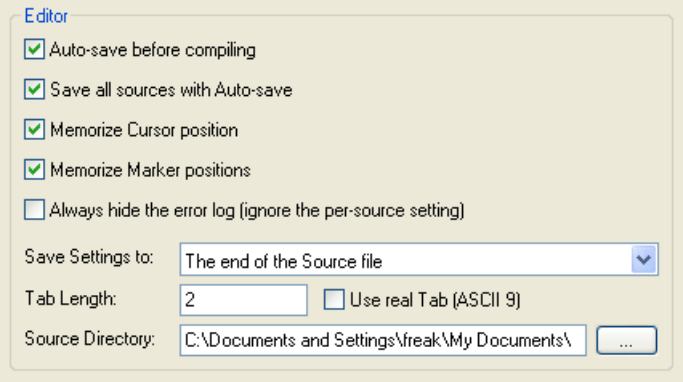
This allows to fully customize the main Toolbar. By selecting an entry and using the Buttons in the "Position" section, you can change the order. The "Item Settings" section can be used to modify the entry or add a new one. (new ones are always added at the end of the list)

Types of items:
Separator : a vertical separator line.
Space : an empty space, the size of one toolbar icon.
Standard Icon : allows you to select a OS standard icon from the combo box on the right.
IDE Icon : allows you to select one of the IDE's own icons in the combo box on the right.
Icon File : allows you to specify your own icon file to use in the edit box on the right. (PNG files are supported on all platforms, Windows additionaly supports icon files.)

If you do not select a separator or space, you can specify an action to carry out when the button is pressed:
Menu Item : carries out the menu command specified in the combo box on the right.
Run tool : executes the external tool specified in the combo box on the right.

The "Default Sets" section contains two standard toolbar sets which you can select, and later modify.

Editor



Settings that affect the management of the source codes.

Auto-save before compiling
Saves the current source code before each compile/run or executable creation. Note that any open include files are not automatically saved.

Save all sources with Auto-save
Saves all sources instead of just the current one with one of the Auto-save options.

Memorize cursor position
Saves the current cursor position, as well as the state of all folding marks with the compiler options for the source file.

Memorize Marker positions
Saves all the Markers with the options for the source file.

Always hide the error log
The error log can be shown/hidden on a per-source basis. This option provides a global setting to ignore the per-source setting and never display the error log. It also removes the corresponding menu entries from the IDE menu.

Save settings to
This option allows to specify where the compiler options of a source file are saved:

> The end of the Source file
> Saves the settings as a special comment block at the end of each source file.
>
> The file <filename>.pb.cfg
> Creates a .pb.cfg file for each saved source code that contains this information.
>
> A common file project.cfg for every directory
> Creates a file called project.cfg in each directory where PB files are saved. This one file will contain the options for all files in that directory.
>
> Don't save anything
> No options are saved. When reopening a source file, the defaults will always be used.

Tab Length
Allows to specify how many spaces are inserted each time you press the Tab key.

Use real Tab (Ascii 9)
If set, the tab key inserts a real tab character instead of spaces. If not set, there are spaces inserted when Tab is pressed.

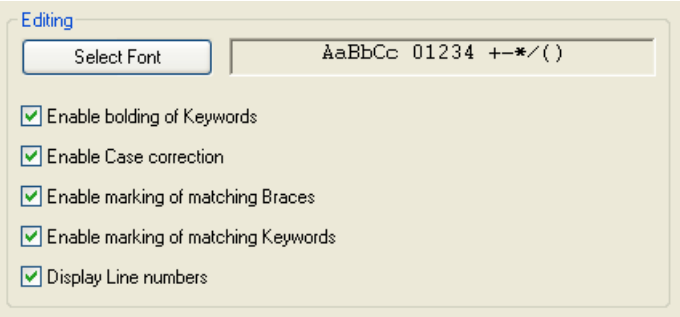
Note that if real tab is used, the "Tab Length" option specifies the size of one displayed tab character.

Source Directory

Specifies the default directory used in the Open and Save dialogs if no other files are currently open (if another file is open, its path will be used as default).

Set this to the path were you usually save the source codes.

Editor - Editing



Use "Select Font" to change the font used to display the source code. To ensure a good view of the source code, it should be a fixed-size font, and possibly even one where bold characters have the same size as non-bold ones.

Enable bolding of keywords
If your font does not display bold characters in the same size as non-bold ones, you should disable this option. If disabled, the keywords will not be shown as bold.

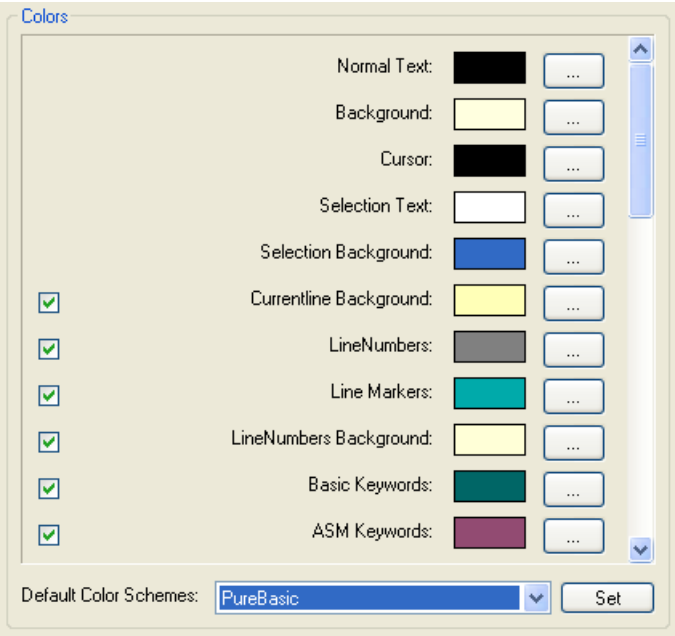
Enable case correction
If enabled, the case of PureBasic keywords, PureBasic Functions as well as predefined constants will automatically be corrected while you type.

Enable marking of matching Braces
If enabled, the brace matching the one under the cursor will be hilighted.

Enable marking of matching Keywords
If enabled, the keyword(s) matching the one under the cursor will be underlined.

Display line numbers
Shows or hides the line number column on the left.

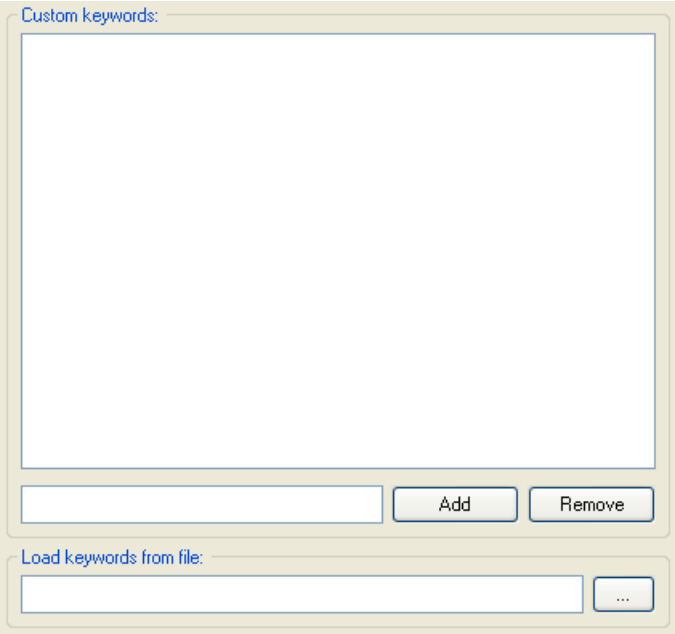
Editor - Coloring



Here you can change the color settings for the syntax coloring, as well as he debugger marks. Default color schemes can be selected from the box on the bottom, and also modified after they have been set. Individual color settings can be disabled by use of the checkboxes.

Note: The 'Accessibility' color scheme has (apart from high-contrast colors) a special setting to always use the system color for the selection in the code editor. This helps screen-reader applications to better detect the selected text.

Editor - Coloring - Custom Keywords

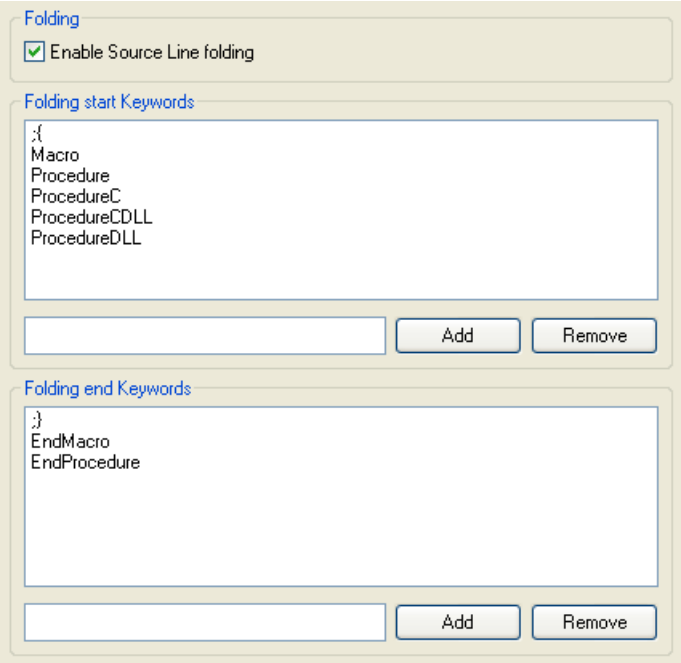


In this section, a list of custom keywords can be defined. These keywords can have a special color assigned to them in the coloring options and the IDE will apply case-correction to them if this feature is enabled. This allows for applying a special color to special keywords by preprocessor tools or macro sets, or to simply have some PB keywords colored differently.

Note that these keywords take precedence above all other coloring in the IDE, so this allows to change the color or case correction even for PureBasic keywords.

The keywords can be either entered directly in the preferences or specified in a textfile with one keyword per line. (or both)

Editor - Folding

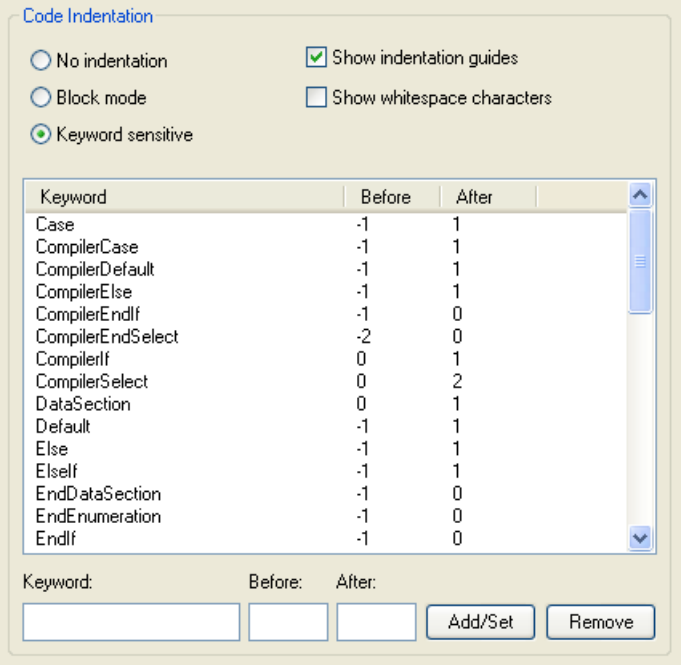


Here you can set the keywords in the source code that start/end a foldable section of code. You can add any number of words that will mark such a sections. You can also choose to completly disable the folding feature.

Words that are found inside comments are ignored, unless the defined keyword includes the comment symbol at the start. (like the default ";{" keyword)

A keyword may not include spaces.

Editor - Indentation



Here you can specify how the editor handles code indentation when the return key is pressed.

No indentation
Pressing return always places the cursor at the beginning of the next line.

Block mode
The newly created line gets the same indentation as the one before it.

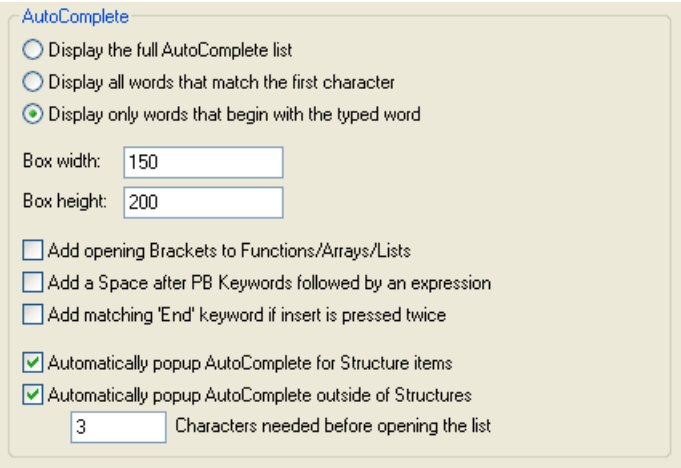
Keyword sensitive
Pressing the return key corrects the indentation of both the old line and the new line depending on the keywords on these lines. The rules for this are specified in the keyword list below. These rules also apply when the "Reformat indentation" item in the edit menu is used.

Show indentation guides
Causes vertical lines to be shown to visualize the indentation on each line. This makes it easier to see which source lines are on the same level of indentation.

Show whitespace characters
Causes whitespace characters to be visible as little dots (spaces) or arrows (tab characters).

The keyword list contains the keywords that have an effect on the indentation. The "Before" setting specifies the change in indentation on the line that contains the keyword itself while the "After" setting specifies the change that applies to the line after it.

Editor - Auto complete



Display the full Auto complete list
Always displays all keywords in the list, but selects the closest match.

Display all words that start with the first character
Displays only those words that start with the same character as you typed. The closest mach is selected.

Display only words that start with the typed word
Does not display any words that do not start with what you typed. If no words match, the list is not displayed at all.

Box width / Box height
Here you can define the size of the auto complete list (in pixel). Note that these are maximum values. The displayed box may become smaller if there are only a few items to display.

Add opening Brackets to Functions/Arrays/Lists
Will automatically add a "(" after any function/Array/Linked list inserted by auto complete. Functions with no parameters or Linked lists get a "()" added.

Add a Space after PB Keywords followed by an expression
When inserting PB keywords that cannot appear alone, a space is automatically added after them.

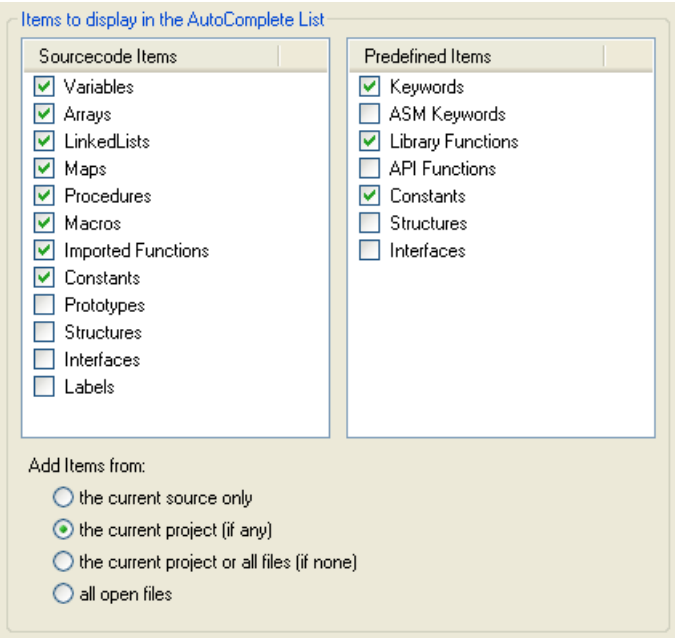
Add matching End' keyword if Tab/Enter is pressed twice
If you press Tab or Enter twice, it will insert the corresponding end keyword (for example "EndSelect" to "Select" or "EndIf " to "If") to the keyword you have just inserted. The end keyword will be inserted after the cursor, so you can continue typing after the first keyword that was inserted.

Automatically popup AutoComplete for Structure items
Displays the list automatically whenever a structured variable or interface is entered and the "\" character is typed after it to show the list of possible structure fields. If disabled, the list can still be displayed by pressing the keyboard shortcut to open the AutoComplete window (usually Ctrl+Space, this can be modified in the Shortcuts section of the Preferences).

Automatically popup AutoComplete outside of Structures
Displays the list automatically when the current word is not a structure after a certain amound of characters has been typed, and a possible match in the list is found. If disabled, the list can still be displayed by pressing the assigned keyboard shortcut.

Characters needed before opening the list
Here you can specify how many characters the word must have minimum before the list is automatically displayed.

Editor - Auto complete - Displayed items



This shows a list of possible items that can be included with the possible matches in the AutoComplete list.

Sourcecode Items
Items defined in the active source code, or other open sources (see below).

Predefined Items
Items that are predefined by PureBasic, such as the PureBasic keywords, functions or predefined constants.

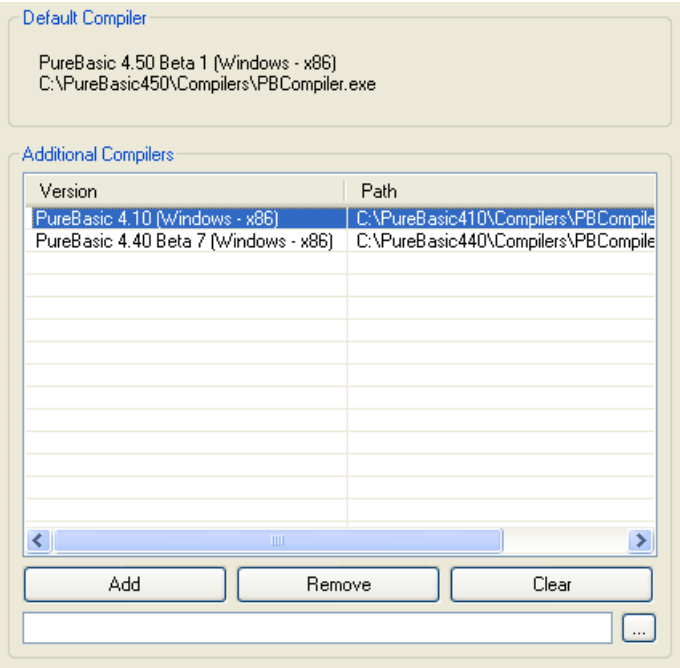
Add Items from: the current source only
Sourcecode items are only added from the active source code.

Add Items from: the current project (if any)
Sourcecode items are added from the current project if there is one. The other source codes in the project do not have to be currently open in the IDE for this.

Add Items from: the current project or all files (if none)
Sourcecode items are added from the current project. If the current source code does not belong to the open project then the items from all open source codes will be added.

Add Items from: all open files
Sourcecode items are added from all currently open source codes.

Compiler

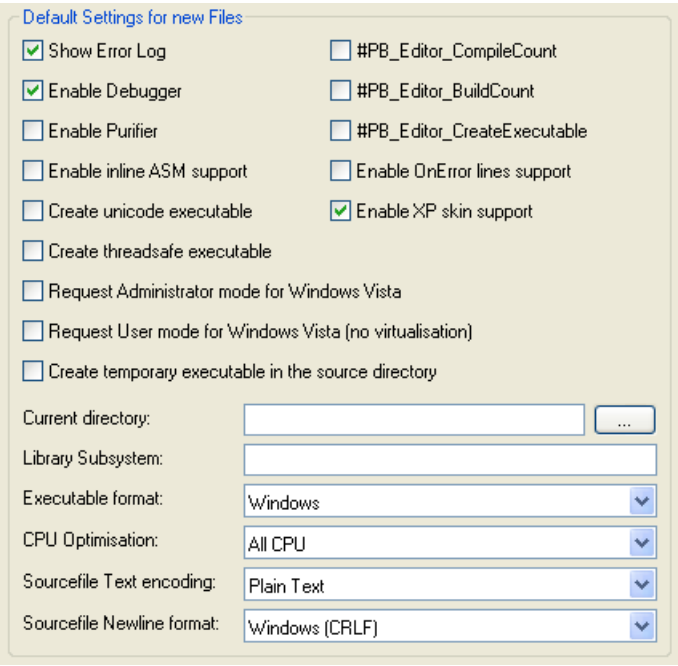


This page allows to select additional compilers which should be available for compilation in the Compiler Options . This allows switching between different compilers of the same version (like the x86 and x64 compilers) or even switching between different versions easily.

Any PureBasic compiler starting from Version 4.10 can be added here. The target processor of the selected compilers does not have to match that of the default compiler, as long as the target operating system is the same. The list displays the compiler version and path of the selected compilers.

The information used by the IDE (for code highlighting, auto complete, structure viewer) always comes from the default compiler. The additional compilers are only used for compilation.

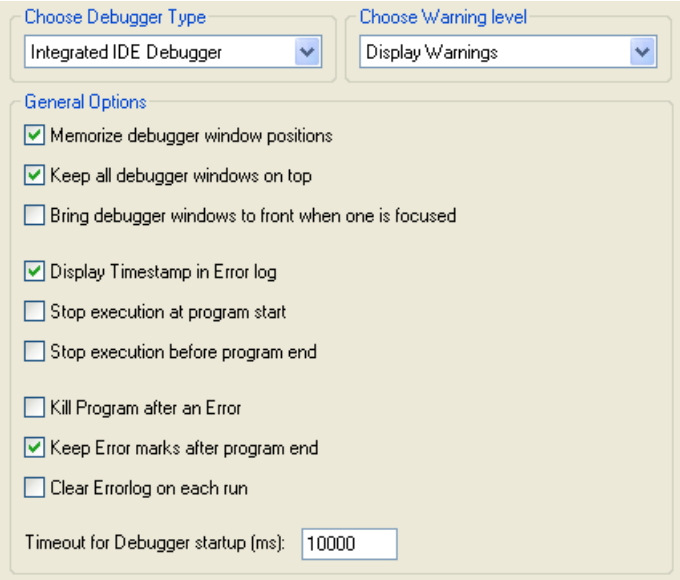
Compiler - Defaults



This page allows to set the default compiler options that will be used when you create a new source code with the IDE.

For an explanation of the meaning of each field, see the Compiler Options .

Debugger



Settings for the internal Debugger, or the Standalone Debugger. The commandline debugger is configured from the commandline only.

Debugger Type
Select the type of debugger you want to use when compiling from the IDE here.

Choose Warning level
Select the action that should be taken if the debugger issues a warning. The available options are:
Ignore Warnings: Warnings will be ignored without displaying anything.
Display Warnings: Warnings will be displayed in the error log and the source code line will be marked, but the program continues to run.
Treat Warnings as Errors: A warning will be treated like an error.
See Using the debugger for more information on debugger warnings and errors.

Memorize debugger window positions
The same as the "Memorize Window positions" for in the General section, but for all Debugger windows.

Keep all debugger windows on top
All debugger windows will be kept on top of all other windows, even from other applications.

Bring Debugger windows to front when one is focused
With this option set, focusing one window that belongs to the debugger of a file, all windows that belong to the same debugging session will be brought to the top.

Display Timestamp in error log
Includes the time of the event in the error log.

Stop execution at program start
Each program will be started in an already halted mode, giving you the opportunity to start moving step by step, right from the start of the program.

Stop execution before program end
Stops the program execution right before the executable would unload. This gives you a last chance to

use the debugging tools to examine Variables or Memory before the program ends.

Kill Program after an Error
If a program encounters an error, it will be directly ended and all debugger windows closed. This gives the opportunity to directly modify the code again without an explicit "Kill Program", but there is no chance to examine the program after an error.

Keep Error marks after program end
Does not remove the lines marked with errors when the program ends. This gives the opportunity to still see where an error occurred while editing the code again.

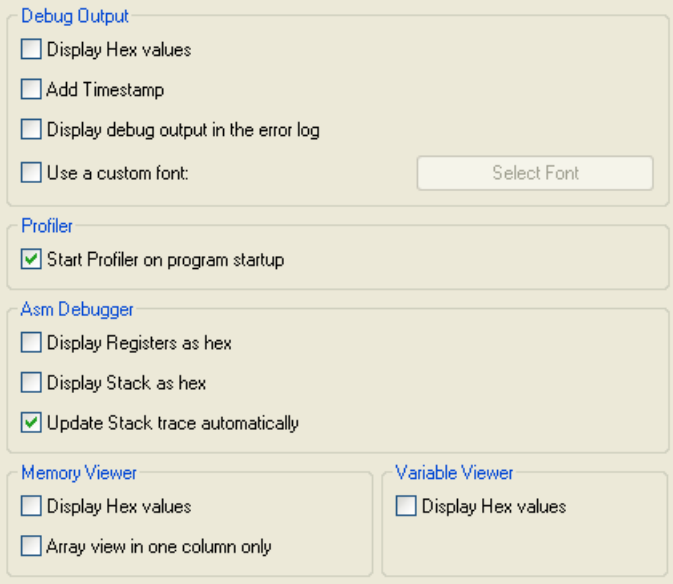
The marks can be manually removed with the "Clear error marks" command in the "Error log" submenu of the debugger menu.

Clear error log on each run
Clears the log window when you execute a program. This ensures that the log does not grow too big. (This option is also available with the commandline debugger selected.)

Timeout for Debugger startup
Specifies the time in milliseconds how long the debugger will wait for a program to start up before giving up. This timeout prevents the debugger from locking up indefinately if the executable cannot start for some reason.

Debugger - Individual Settings

This allows to set options for the individual debugger tools. The "Display Hex values" options switch between displaying Byte, Long and Word as decimal or hexadecimal values.



Debug Output Add Timestamp
Adds a timestamp to the output displayed from the Debug command.

Debug Output - Display debug output in the error log
With this option enabled, a Debug command in the code will not open the Debug Output window , but instead show the output in the error log .

Debug Output Use custom font
A custom font can be selected here for the debug output window . This allows to specify a smaller font for much output or a proportional one if this is desired.

Profiler - Start Profiler on program startup

Determines wether the profiler tool should start recording data when the program starts.

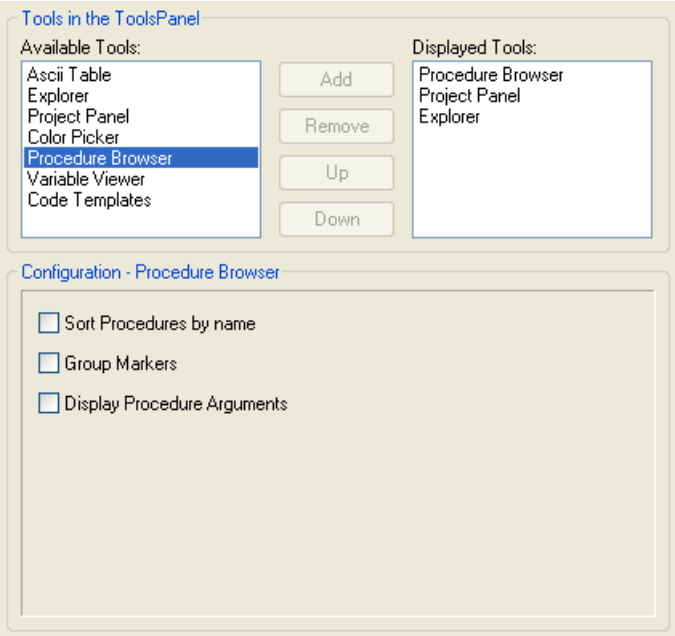
Asm Debugger Update Stack trace automatically
Updates the stack trace automatically on each step/stop you do. If disabled, a manual update button will be displayed in the Asm window.

Memory Viewer Array view in one column only
If the memory area is displayed in array view, this option selects whether it should be multi-column (with 16 bytes displayed in each column) or with only one column.

Debugger - Default Windows

The debugger tools you select on this page will automatically be opened each time you run a program with enabled debugger.

Tools Panel



This allows to configure the internal tools that can be displayed in the side panel. Each tool that is in the "Displayed Tools" list is displayed in the Panel on the side of the edit area. Each tool that is not listed there is accessible from the Tools menu as a separate window.
Put only those tools in the side panel that you use very frequently, and put the most frequently used first, as it will be the active one once you open the IDE.

By selecting a tool in either of the lists, you get more configuration options for that tool (if there are any) in the ”Configuration” section below.

Here is an explanation of those tools that have special options:

Explorer
You can select between a Tree or List display of the file system. You can also set whether the last displayed directory should be remembered, or if the Source Path should be the default directory when the IDE is started.

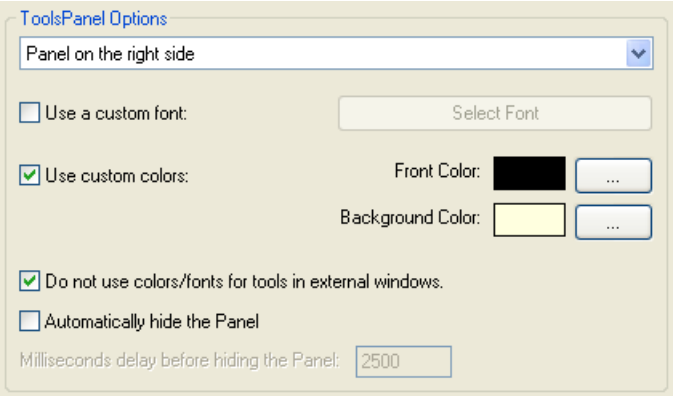
Color Picker
You can change the number of saved color buttons that will be displayed in the Colorpicker.

Procedure Browser
”Sort Procedures by Name” : sorts the list alphabetically (by default they are listed as they appear in the code).
”Group Markers” : groups the ”;-” markers together.
”Display Procedure Arguments” : Displays the full declaration of each procedure in the list.

Variable Viewer
The ”Display Elements from all open sources” option determines whether the list should only include items from this code, or from all open codes.

Furthermore you can select the type of items displayed in the Variable viewer.

Tools panel - Options



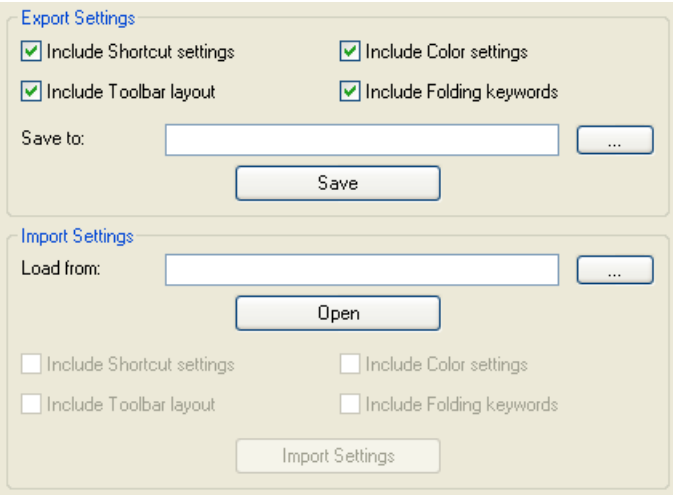
Here you can customize the appearance of the Tools Panel a bit more. You can select the side on which it will be displayed, a Font for its text, as well as a foreground and background color for the displayed tools. The font and color options can be disabled to use the OS defaults instead.

Do not use colors/fonts for tools in external windows
If set, the color/font options only apply to the Tools displayed in the Panel, those that you open from the Tools menu will have the default colors.

Automatically hide the Panel
To save space, the Panel will be hidden if the mouse is not over it. Moving the mouse to the side of the IDE will show it again.

Milliseconds delay before hiding the Panel
Sets a timeout in ms, after which the Panel is hidden if you leave it with the mouse.

Import/Export



This section allows you to export the layout settings of the IDE in a platform independent format, which allows you to import them again into the PureBasic IDE on another Operating System, or to share your options with other PB users.

To export your settings, select what types of settings you want to include, select a filename and press the "Save" button.

To import settings, select the filename and press "Open". You will then see the options that are included in this file as enabled checkboxes. After selecting what you want to import, click the "Import Settings" button.

For the new settings to take effect, you have to first click the apply button.

Note: You can import the style files from the jaPBe Editor, but only the color settings will be imported.

Chapter 19

Commandline options for the IDE

The PureBasic IDE allows you to modify the paths and files being used from the commandline. This allows you to create several shortcuts that start the IDE with different configurations for different users, or for different projects.

Possible commandline options are:

```
/P <Preferences file>    loads/saves all the configuration to/from
 the given file.
/T <Templates file>      loads/saves the code templates from/to the
 given file.
/A <tools file>          loads/saves the configuration of the
 external tool from/to this file.
/S <Source path>         overwrites the "Source path" setting from
 the preferences
/E <Explorer path>       starts the Explorer tool with the given path
/L <Line number>         moves the cursor to the given line number in
 the last opened file
/NOEXT                   disables the registering of the .pb
 extension in the registry.
/LOCAL                   puts all preferences in the PureBasic
 directory instead of the user profile location.
/PORTABLE                the same as /LOCAL and /NOEXT combined
```

The default files for /P /T and /A are saved in the %APPDATA%\PureBasic\ directory on the system.

The /NOEXT command is useful when you have several different PB versions at once (for testing of beta versions for example), but want the .pb extension to be associated with only one of them. The /PORTABLE command can be used to keep all configuration inside the local directory to easily copy PureBasic to different computers (or run it from USB sticks for example).

Example:

```
PureBasic.exe Example.pb /PORTABLE
```

You can also put filenames of source files to load on the commandline. You can even specify wildcards for them. (so with "*.pb" you can load a whole directory.)

Part III

Language Reference

Chapter 20

Working with different number bases

(Note: These examples use the ^ symbol to mean 'raised to the power of' - this is only for convenience in this document, PureBasic does not currently have a power-of operator! Use the PureBasic command Pow() from the "Math" Library instead.)

Introduction

A number base is a way of representing numbers, using a certain amount of possible symbols per digit. The most common you will know is base 10 (a.k.a. decimal), because there are 10 digits used (0 to 9), and is the one most humans can deal with most easily.

The purpose of this page is to explain different number bases and how you can work with them. The reason for this is that computers work in binary (base 2) and there are some cases where it is advantageous to understand this (for example, when using logical operators, bit masks, etc).

Overview of number bases

Decimal System

Think of a decimal number, and then think of it split into columns. We'll take the example number 1234. Splitting this into columns gives us:

```
1    2    3    4
```

Now think of what the headings for each column are. We know they are units, tens, hundreds and thousands, laid out like this:

```
1000   100    10     1
   1     2     3     4
```

We can see that the number 1234 is made up out of

```
  1*1000=1000
+ 2* 100= 200
+ 3*  10=  30
+ 4*   1=   4
Total   =1234
```

If we also look at the column headings, you'll see that each time you move one column to the left we multiply by 10, which just so happens to be the number base. Each time you move one column to the

right, you divide by 10. The headings of the columns can be called the weights, since to get the total number we need to multiply the digits in each column by the weight. We can express the weights using indices. For example 10 ^2 means '10 raised to the power of two' or 1*10*10 (=100). Similarly, 10 ^4 means 1*10*10*10*10 (=10000). Notice the pattern that whatever the index value is, is how many times we multiply the number being raised. 10 ^0 means 1 (since we multiply by 10 no times). Using negative numbers shows that we need to divide, so for example 10 ^-2 means 1/10/10 (=0.01). The index values make more sense when we give each column a number - you'll often see things like 'bit 0' which really means 'binary digit in column 0'.

```
In this example, ^ means raised to the power of, so 10^2 means 10
 raised to the power of 2.
Column number              3     2     1     0
Weight (index type)     10^3  10^2  10^1  10^0
Weight (actual value)   1000   100    10     1
Example number (1234)      1     2     3     4
```

A few sections ago we showed how to convert the number 1234 into its decimal equivalent. A bit pointless, since it was already in decimal but the general method can be shown from it - so this is how to convert from any number to its decimal value:

```
B = Value of number base

1) Separate the number in whatever base you have into it's columns.
 For
   example, if we had the value 'abcde' in our fictional number base
 'B',
   the columns would be:   a   b   c   d   e

2) Multiply each symbol by the weight for that column (the weight
 being
   calculated by 'B' raised to the power of the column number):
     a * B^4 = a * B * B * B * B
     b * B^3 = b * B * B * B
     c * B^2 = c * B * B
     d * B^1 = d * B
     e * B^0 = e

3) Calculate the total of all those values. By writing all those
 values in their
   decimal equivalent during the calculations, it becomes far easier
 to see the
   result and do the calculation (if we are converting into decimal).
```

Converting in the opposite direction (from decimal to the number base 'B') is done using division instead of multiplication:

```
1) Start with the decimal number you want to convert (e.g. 1234).

2) Divide by the target number base ('B') and keep note of the result
 and the
   remainder.

3) Divide the result of (2) by the target number base ('B') and keep
 note of
   the result and the remainder.

4) Keep dividing like this until you end up with a result of 0.

5) Your number in the target number base is the remainders written in
 the order
```

```
    most recently calculated to least recent. For example, your number
  would be
    the remainders of the steps in this order 432.
```

More specific examples will be given in the sections about the specific number bases.

Binary System

Everything in a computer is stored in binary (base 2, so giving symbols of '0' or '1') but working with binary numbers follows the same rules as decimal. Each symbol in a binary number is called a bit, short for binary digit. Generally, you will be working with bytes (8 bits), words (16 bits) or longwords (32 bits) as these are the default sizes of PureBasic's built in types. The weights for a byte are shown:

```
(^ means 'raised to the power of', number base is 2 for binary)
Bit/column number          7      6      5      4      3      2      1      0
Weight (index)           2^7    2^6    2^5    2^4    2^3    2^2    2^1    2^0
Weight (actual value)    128     64     32     16      8      4      2      1
```

So, for example, if we had the number 00110011 (Base 2), we could work out the value of it like this:

```
  0 * 128
+ 0 *  64
+ 1 *  32
+ 1 *  16
+ 0 *   8
+ 0 *   4
+ 1 *   2
+ 1 *   1
=      51
```

An example of converting back would be if we wanted to write the value 69 in binary. We would do it like this:

```
69 / 2 = 34 r 1      ^
34 / 2 = 17 r 0     /|\
17 / 2 =  8 r 1      |
 8 / 2 =  4 r 0      |    Read remainders in this direction
 4 / 2 =  2 r 0      |
 2 / 2 =  1 r 0      |
 1 / 2 =  0 r 1      |
(Stop here since the result of the last divide was 0)

Read the remainders backwards to get the value in binary = 1000101
```

Another thing to consider when working with binary numbers is the representation of negative numbers. In everyday use, we would simply put a negative symbol in front of the decimal number. We cannot do this in binary, but there is a way (after all, PureBasic works mainly with signed numbers, and so must be able to handle negative numbers). This method is called 'twos complement' and apart from all the good features this method has (and won't be explained here, to save some confusion) the simplest way to think of it is the weight of the most significant bit (the MSb is the bit number with the highest weight, in the case of a byte, it would be bit 7) is actually a negative value. So for a two's complement system, the bit weights are changed to:

```
(^ means 'raised to the power of', number base is 2 for binary)
Bit/column number          7      6      5      4      3      2      1      0
Weight (index)          -2^7    2^6    2^5    2^4    2^3    2^2    2^1    2^0
Weight (actual value)   -128     64     32     16      8      4      2      1
```

and you would do the conversion from binary to decimal in exactly the same way as above, but using the new set of weights. So, for example, the number 10000000 (Base 2) is -128, and 10101010 (Base 2) is -86.

To convert from a positive binary number to a negative binary number and vice versa, you invert all the bits and then add 1. For example, 00100010 would be made negative by inverting -> 11011101 and adding 1 -> 11011110.

This makes converting from decimal to binary easier, as you can convert negative decimal numbers as their positive equivalents (using the method shown above) and then make the binary number negative at the end.

Binary numbers are written in PureBasic with a percent symbol in front of them, and obviously all the bits in the number must be a '0' or a '1'. For example, you could use the value %110011 in PureBasic to mean 51. Note that you do not need to put in the leading '0's (that number would really be %00110011) but it might help the readability of your source if you put in the full amount of bits.

Hexadecimal System

Hexadecimal (for base 16, symbols '0'-'9' then 'A'-'F') is a number base which is most commonly used when dealing with computers, as it is probably the easiest of the non-base 10 number bases for humans to understand, and you do not end up with long strings of symbols for your numbers (as you get when working in binary).

Hexadecimal mathematics follows the same rules as with decimal, although you now have 16 symbols per column until you require a carry/borrow. Conversion between hexadecimal and decimal follows the same patterns as between binary and decimal, except the weights are in multiples of 16 and divides are done by 16 instead of 2:

```
Column number              3      2      1      0
Weight (index)          16^3   16^2   16^1   16^0
Weight (actual value)   4096    256     16      1
```

Converting the hexadecimal value BEEF (Base 16) to decimal would be done like this:

```
  B * 4096 = 11 * 4096
+ E *  256 = 14 *  256
+ E *   16 = 14 *   16
+ F *    1 = 15 *    1
=                48879
```

And converting the value 666 to hexadecimal would be done like this:

```
666 / 16 = 41 r 10     ^
 41 / 16 =  2 r  9    /|\    Read digits in this direction,
 remembering to convert
  2 / 16 =  0 r  2     |     to hex digits where required
(Stop here, as result is 0)
Hexadecimal value of 666 is 29A
```

The really good thing about hexadecimal is that it also allows you to convert to binary very easily. Each hexadecimal digit represents 4 bits, so to convert between hexadecimal and binary, you just need to convert each hex digit to 4 bits or every 4 bits to one hex digit (and remembering that 4 is an equal divisor of all the common lengths of binary numbers in a CPU). For example:

```
Hex number          5        9        D        F          4E
Binary value     0101     1001     1101     1111    01001110
```

When using hexadecimal values in PureBasic, you put a dollar sign in front of the number, for example $BEEF.

To finish this topic you find here a table with the numbers 0 till 255 in decimal, binary and hexadecimal format.

Chapter 21

Break : Continue

Syntax

```
Break [Level]
```

Description

Break provides the ability to exit during any interation, for the following loops: Repeat , For , ForEach and While . The optional 'level' parameter specifies the number of loops to exit from, given several nested loops. If no parameter is given, Break exits from the current loop.

Example: Simple loop

```
For k=0 To 10
  If k=5
    Break  ; Will exit directly from the For/Next loop
  EndIf
  Debug k
Next
```

Example: Nested loops

```
For k=0 To 10
  Counter = 0
  Repeat
    If k=5
      Break 2 ; Will exit directly from the Repeat/Until and For/Next loops
    EndIf
    Counter+1
  Until Counter > 1
  Debug k
Next
```

Syntax

```
Continue
```

Description

Continue provides the ability to skip straight to the end of the current iteration (bypassing any code in between) in the following loops: Repeat , For , ForEach , and While .

Example:

```
For k=0 To 10
  If k=5
    Continue  ; Will skip the 'Debug 5' and go directly to the next iteration
  EndIf
  Debug k
Next
```

Chapter 22

Using the command line compiler

Introduction

The command line compiler is located in the subdirectory 'Compilers\' from the PureBasic folder. The easier way to access this is to add this directory in the windows PATH variable, which will give access to all the commands of this directory at all times.

Command switchs

/? : display a quick help about the compiler.

/COMMENTED : create a commented '.asm' output file when creating an executable. This file can be re-assembled later when the needed modifications have been made. This option is for advanced programmers only.

/DEBUGGER : enable the debugger support.

/EXE "Filename" : create a standalone Executable or DLL specified by the filename at the desired path location.

/ICON "IconName" : add the specified icon to the executable.

/INLINEASM : enable the inline asm feature (asm routines can be written directly in the BASIC source code).

/RESIDENT "Filename" : create a resident file specified by the filename.

/CONSOLE: output file in Console format. Default format is Win32.

/DLL: output file is a DLL .

/XP: Add the Windows XP theme support to the executable.

/REASM: Reassemble the PureBasic.asm file into an executable. This allow to use the /COMMENTED function, modify the asm output and creates the executable again.

/LINENUMBERING: Add lines and files information to the executable, which can slow it considerably. This allows the OnError library to report the file and linenumber of an error.

/QUIET: Disable all unnecessary text output, very useful when using another editor.

/STANDBY: Loads the compiler in memory and wait for external commands (editor, scripts...).

/MMX, /3DNOW, /SSE or /SSE2: Creates a processor specific executable. This means than if a processor specific routine is available it will be used for this executable. Therefore, the executable will run correctly only on computer with this kind of CPU.

/DYNAMICCPU: Creates a executable containing all the available processor specific routines. When the program is started, it looks for the processor type and then select the more appropriates routines to use. This makes a bigger executable, but result in the fastest possible program.

/RESOURCE "Filename": Add a Windows resource file (.rc) to the created executable or DLL. This should be not a compiled resource file, but an ascii file with the directives in it. It's possible to specify only one resource, but this file can include other resource script if needed.

/LINKER "ResponseFile": Specify a file which contains commands to be passed directly to the linker. On Windows, PureBasic use the PellesC linker (polink), more informations about the possible options can be found in the related documentation.

/IGNORERESIDENT "Filename": Doesn't load the specified resident file when the compiler starts. It's mostly useful when updating a resident which already exists, so it won't load it.

/CONSTANT Name=Value: Creates the specified constant with the given expression. Example: 'pbcompiler test.pb /CONSTANT MyConstant=10'. The constant #MyConstant will be created on the fly with a value of 10 before the program gets compiled.

/UNICODE: Use unicode instead ascii for strings management.

/THREAD: Use thread safe runtime for strings and general routines.

/SUBSYSTEM "Name": Uses the specific subsystem to replace a set of internal functions.

The following two compiler options are needed for creating programs running on Microsoft Vista OS. They are both options for the included manifest, so they are ignored on older windows versions. With none of these switches, the exe will run as a normal user as well, but with virtualisation turned on (i.e. registry and file redirection). It is recommended to use the /USER switch to turn of virtualisation for all programs that comply to the standard user priviledges, as it is only intended for compatibility for older programs. These options are also available in the IDE compiler options .
/ADMINISTRATOR: Will cause the program to request admin priviledges at start. The program will not run without it. This option is needed. Note: You can also debug programs with this flag, but only with the standalone gui debugger (as it must run in elevated mode as well).
/USER: The program will run as the user who started it. Virtualisation for the exe is turned off.

Example:

```
CLI> pbcompiler sourcecode.pb
```

The compiler will compile the source code and execute it.

```
CLI> pbcompiler sourcecode.pb /DEBUGGER /INLINEASM
```

The compiler will compile the source code and execute it with debugger and inline asm support.

```
CLI> pbcompiler "C:\Project\Source\DLLSource.pb" /EXE
 "C:\Project\project.dll" /DLL
```

The compiler will compile the source code (here with full path) and create the dll "project.dll" in the given directory.

Chapter 23

Compiler Directives

Syntax

```
CompilerIf <constant expression>
  ...
[CompilerElse]
  ...
CompilerEndIf
```

Description

If the <constant expression> result is true, the code inside the CompilerIf will be compiled, else it will be totally ignored. It's useful when building multi-OSes programs to customize some programs part by using OS specific functions. The And and Or Keywords can be used in <constant expression> to combine multiple conditions.

Example:

```
1   CompilerIf #PB_Compiler_OS = #PB_OS_Linux And #PB_Compiler_Processor
     = #PB_Processor_x86
2     ; some Linux and x86 specific code..
3   CompilerEndIf
```

Syntax

```
CompilerSelect <numeric constant>
  CompilerCase <numeric constant>
    ...
  [CompilerDefault]
    ...
CompilerEndSelect
```

Description

Works like a regular Select : EndSelect except that only one numeric value is allowed per case. It will tell the compiler which code should be compiled. It's useful when building multi-OSes programs to customize some programs part by using OS specific functions.

Example:

```
CompilerSelect #PB_Compiler_OS
  CompilerCase #PB_OS_AmigaOS
    ; some Amiga specific code
  CompilerCase #PB_OS_Linux
    ; some Linux specific code
CompilerEndSelect
```

Syntax

```
CompilerError <string constant>
```

Description

Generates an error, as if it was a syntax error and display the associated message. It can be useful when doing specialised routines, or to inform than a source code is not available on an particular OS.

Example:

```
CompilerIf #PB_Compiler_OS = #PB_OS_AmigaOS
  CompilerError "AmigaOS isn't supported, sorry."
CompilerElse
  CompilerError "OS supported, you can now comment me."
CompilerEndIf
```

Syntax

```
EnableExplicit
DisableExplicit
```

Description

Enables or disables the explicit mode. When enabled, all the variables which are not explicitly declared with Define , Global , Protected or Static are not accepted and the compiler will raise an error. It can help to catch typo bugs.

Example:

```
EnableExplicit

Define a

a = 20 ; Ok, as declared with 'Define'
b = 10 ; Will raise an error here
```

Syntax

```
EnableASM
DisableASM
```

Description

Enables or disables the inline assembler. When enabled, all the assembler keywords are available directly in the source code, see the inline assembler section for more informations.

Example:

```
; x86 assembly example
;
Test = 10

EnableASM
  MOV dword [v_Test],20
DisableASM

Debug Test ; Will be 20
```

Reserved Constants

The PureBasic compiler has several reserved constants which can be useful to the programmer:

```
#PB_Compiler_OS : Determines on which OS the compiler is currently
 running. It can be one of the following values:
  - #PB_OS_Windows : The compiler is running on Windows
  - #PB_OS_Linux   : The compiler is running on Linux
  - #PB_OS_AmigaOS : The compiler is running on AmigaOS
  - #PB_OS_MacOS   : The compiler is running on Mac OS X

#PB_Compiler_Processor : Determines the processor type for which the
 program is compiling. It can be one of the following:
  - #PB_Processor_x86     : x86 processor architecture (also called
 IA-32 or x86-32)
  - #PB_Processor_x64     : x86-64 processor architecture (also
 called x64, AMD64 or Intel64)
  - #PB_Processor_PowerPC : PowerPC processor architecture
  - #PB_Processor_mc68000 : Motorola 68000 processor architecture

#PB_Compiler_Date     : Current date, at the compile time, in the
 PureBasic  date
format.
#PB_Compiler_File     : Full path and name of the file being
 compiled, useful for debug
purpose.
#PB_Compiler_FilePath : Full path of the file being compiled, useful
 for debug
purpose.
#PB_Compiler_Line     : Line number of the file being compiled,
 useful for debug
purpose.
#PB_Compiler_Procedure: Current procedure name, if the line is inside
 a procedure
.
```

```
 #PB_Compiler_Version  : Compiler version, in integer format in the
  form '420' for 4.20.
 #PB_Compiler_Home     : Full path of the PureBasic directory, can be
  useful to locate include files

 #PB_Compiler_Debugger : Set to 1 if the runtime debugger
is enabled, set to 0 else. When an executable
                        is created, the debugger is always disabled
  (this constant will be 0).
 #PB_Compiler_Thread   : Set to 1 if the executable is compiled in
  thread safe mode, set to 0 else.
 #PB_Compiler_Unicode  : Set to 1 if the executable is compiled in
  unicode
mode, set to 0 else.
```

Chapter 24

Compiler Functions

Syntax

```
Size = SizeOf(Type)
```

Description

SizeOf can be used to find out the size of any complex Structure (it does not work on the simple built-in types such as word and float), Interface or even variables . This can be useful in many areas such as calculating memory requirements for operations, using API commands, etc.

Note: In unicode mode one character uses 2 bytes. In Ascii mode one character uses 1 byte. Sizeof(Character) allows to return the size (in bytes), which is taken by one character according to the active mode.

Example:

```
Structure Person
  Name.s
  ForName.s
  Age.w
EndStructure

Debug "The size of my friend is "+Str(Sizeof(Person))+" bytes" ; will be 10 (4+4+2)

John.Person\Name = "John"

Debug SizeOf(John) ; will be also 10
```

Note: if a variable and a structure have the same name, the structure will have the priority over the variable.

Syntax

```
Index = OffsetOf(Structure\Field)
Index = OffsetOf(Interface\Function())
```

Description

OffsetOf can be used to find out the index of a Structure field or the index of an Interface function. When used with an Interface , the function index is the memory offset, so it will be IndexOfTheFunction*4.

Example:

```
Structure Person
  Name.s
  ForName.s
  Age.w
EndStructure

Debug OffsetOf(Person\Age) ; will be 8 as a string is 4 byte in memory


Interface ITest
  Create()
  Destroy(Flags)
EndInterface

Debug OffsetOf(ITest\Destroy()) ; will be 4
```

Syntax

```
Result = Subsystem(<constant string expression>)
```

Description

Subsystem can be used to find out if a subsystem is in use for the program being compiled. The name of the subsystem is not case sensitive.

Example:

```
CompilerIf Subsystem("OpenGL")
  Debug "Compiling with the OpenGL subsystem"
CompilerEndIf
```

Syntax

```
Result = Defined(Name, Type)
```

Description

Defined checks if a particular object of a code source like structure , interface , variables etc. is already defined or not. The 'Name' parameter has to be specified without any extra decoration (ie: without the '#' for a constant , without '()' for an array , a linkedlist or a map).

The 'Type' parameter can be one of the following values:

```
#PB_Constant
#PB_Variable
#PB_Array
#PB_LinkedList
#PB_Map
#PB_Structure
#PB_Interface
#PB_Procedure
#PB_Function
#PB_OSFunction
```

Example:

```
#PureConstant = 10

CompilerIf Defined(PureConstant, #PB_Constant)
  Debug "Constant 'PureConstant' is already declared"
CompilerEndIf

Test = 25

CompilerIf Defined(Test, #PB_Variable)
  Debug "Variable 'Test' is already declared"
CompilerEndIf
```

Syntax

```
InitializeStructure(*Pointer, Structure)
```

Description

InitializeStructure initialize the specified structured memory area. This is useful when the structure contains dynamic objects like array ", list or map which have to be allocated internally by PureBasic. 'Structure' is the name of the structure which should be used to perform the initialization. There is no internal check to ensures than the structure match the memory area. This function is for advanced users only and should be used with care.

Example:

```
Structure People
  Name$
  Age.l
  List Friends.s()
EndStructure

*Student.People = AllocateMemory(SizeOf(People))
InitializeStructure(*Student, People)

; Now the list is ready to use
;
AddElement(*Student\Friends())
*Student\Friends() = "John"

```

```
AddElement(*Student\Friends())
*Student\Friends() = "Yann"

; Print out the list content
;
ForEach *Student\Friends()
  Debug *Student\Friends()
Next
```

Syntax

```
CopyStructure(*Source, *Destination, Structure)
```

Description

CopyStructure copy the memory of a structured memory area to another. This is useful when dealing with dynamic allocations, trough pointers. Every field will be duplicated, even array, list, and map. There is no internal check to ensures than the structure match the two memory area. This function is for advanced users only and should be used with care.

Example:

```
Structure People
  Name$
  LastName$
  Map Friends$()
  Age.l
EndStructure

Student.People\Name$ = "Paul"
Student\LastName$ = "Morito"
Student\Friends$("Tom") = "Jones"
Student\Friends$("Jim") = "Doe"

CopyStructure(@Student, @StudentCopy.People, People)

Debug StudentCopy\Name$
Debug StudentCopy\LastName$
Debug StudentCopy\Friends$("Tom")
Debug StudentCopy\Friends$("Jim")
```

Syntax

```
ClearStructure(*Pointer, Structure)
```

Description

ClearStructure free the memory of a structured memory area. This is useful when the structure contains strings which have been allocated internally by PureBasic. 'Structure' is the name of the structure which should be used to perform the clearing. There is no internal check to ensures than the structure match the memory area. This function is for advanced users only and should be used with care.

Example:

```
Structure People
  Name$
  LastName$
  Age.l
EndStructure

Student.People\Name$ = "Paul"
Student\LastName$ = "Morito"
Student\Age = 10

ClearStructure(@Student, People)

; Will print empty strings as the whole structure has been cleared. All other fields have been resetted to zero.
;
Debug Student\Name$
Debug Student\LastName$
Debug Student\Age
```

Chapter 25

Data

Introduction

PureBasic allows the use of Data, to store predefined blocks of information inside of your program. This is very useful for default values of a program (language string for example) or, in a game, to define the sprite way to follow (precalculated).

DataSection must be called first to indicate than a data section follow. This mean than all labels, and data component will be stored in the data section of the program, which has a much faster access than the code section. Data will be used to enter the data. EndDataSection must be specified if some code follows the data declaration. One of good stuff is you can define different Data sections in the code without any problem. Restore and Read command will be used to retrieve the data.

Commands

Syntax

```
DataSection
```

Description

Start a data section.

Syntax

```
EndDataSection
```

Description

End a data section.

Syntax

```
Data.TypeName
```

Description

Defines data. The type can only be a native basic type (integer, long, word, byte, ascii, unicode, float, double, quad, character, string). Any number of data can be put on the same line, each one delimited with a comma ',.

Example:

```
Data.l 100, 200, -250, -452, 145
Data.s "Hello", "This", "is ", "What ?"
```

For advanced programmers: it's also possible to put a procedure address or a label address inside Data when its type is set to integer (.i). (Using the 'integer' type will store the (different) addresses accurate on both 32-bit and 64-bit environments.) This can be used to build easy virtual function tables for example.

Example:

```
Procedure Max(Number, Number2)
EndProcedure

Label:

DataSection
  Data.i ?Label, @Max()
EndDataSection
```

Example:

```
Interface MyObject
  DoThis()
  DoThat()
EndInterface

Procedure This(*Self)
  MessageRequester("MyObject", "This")
EndProcedure

Procedure That(*Self)
  MessageRequester("MyObject", "That")
EndProcedure

m.MyObject = ?VTable

m\DoThis()
m\DoThat()


DataSection
  VTable:
    Data.i ?Procedures
  Procedures:
    Data.i @This(), @That()
EndDataSection
```

Syntax

```
Restore label
```

Description

This keyword is useful to set the start indicator for the Read to a specified label. All labels used for this purpose should be placed within the DataSection because the data is treated as a separate block from the program code when it is compiled and may become disassociated from a label if the label were placed outside of the DataSection.

Example:

```
Restore StringData
Read.s MyFirstData$
Read.s MySecondData$

Restore NumericalData
Read.l a
Read.l b

Debug MyFirstData$
Debug a

End

DataSection
  NumericalData:
    Data.l 100, 200, -250, -452, 145

  StringData:
    Data.s "Hello", "This", "is ", "What ?"
EndDataSection
```

Syntax

```
Read[.<type>] <variable>
```

Description

Read the next available data. The next available data can be changed by using the Restore command. By default, the next available data is the first data declared. The type of data to read is determined by the type suffix. The default type will be used if it is not specified.

Chapter 26

Debugger keywords in PureBasic

Overview

A complete description of all functions of the powerful debugger you will find in the extra chapters Using the debugger or Using the debugging Tools .

Following is a list of special keywords to control the debugger from your source code. There is also a Debugger library which provides further functions to modify the behavior of the debugger should it be present.

Syntax

```
CallDebugger
```

Description

This invokes the ”debugger” and freezes the program immediately.

Syntax

```
Debug <expression> [, DebugLevel]
```

Description

Display the DebugOutput window and the result inside it. The expression can be any valid PureBasic expression, from numeric to string. An important point is than the Debug command and its associated expression is totally ignored (not compiled) when the debugger is deactivated.

Note: This is also true, if you’re using complete command lines after Debug (e.g. Debug LoadImage(1,”test.bmp”)). They will not be compiled with disabled debugger!

This means than this command can be used to trace easily in the program without having to comment all the debug commands when creating the final executable.

The ’DebugLevel’ is the level of priority of the debug message. All normal debug message (without specifying a debug level) are automically displayed. When a level is specified then the message will be only displayed if the current DebugLevel (set with the following DebugLevel command) is equals or above this number. This allows hierarchical debug mode, by displaying more and more precise informations in function of the used DebugLevel.

Syntax

```
DebugLevel <constant expression>
```

Description

Set the current debug level for the 'Debug' message.

Note: The debug level is set at compile time, which means than you have to put the DebugLevel command before any other Debug commands to be sure it affects them all.

Syntax

```
DisableDebugger
```

Description

This will disable the debugger checks on the source code which follow this command.

Syntax

```
EnableDebugger
```

Description

This will enable the debugger checks on the source code which follow this command (if the debugger was previously disabled with DisableDebugger).

Note: EnableDebugger doesn't have an effect, if the debugger is completely disabled in the IDE (look at Compiler settings).

Chapter 27

Define

Syntax

```
Define.<type> [<variable> [= <expression>], <variable> [=
    <expression>], ...]
```

Description

If no <variables> are specified, Define is used to change the default type for future untyped variables (including procedure parameters, interface method parameters and data to read with the Read keyword). The initial default type is integer (.i). Each variable can have a default value directly assigned to it.

Example:

```
d = e + f
Define.w
a = b + c
```

d, e and f will be created as integer type variables, since there was no type specified. a, b and c will be signed word typed (.w) as no type is specified and the default type had been changed to the word type.

If variables are specified, Define only declares these variables as "defined type" and will not change the default type. If you don't assign a value to the variables at this time, the value will be 0.

Example:

```
Define.b a, b = 10, c = b*2, d ; these 4 variables will be byte typed
  (.b)
```

Syntax

```
Define <variable>.<type> [= <expression>] [, <variable>.<type> [=
    <expression>], ...]
```

Description

Alternative possibility for the variable declaration using Define.

Example:

```
Define MyChar.c
Define MyLong.l
Define MyWord.w

Debug SizeOf(MyChar)    ; will give 1 byte
Debug SizeOf(MyLong)    ; will give 4 bytes
Debug SizeOf(MyWord)    ; will give 2 bytes
```

Chapter 28

Dim

Syntax

```
Dim name.<type>(<expression>, [<expression>], ...)
```

Description

Dim is used to create new arrays (the initial value of each element will be zero). An array in PureBasic can be of any types, including structured , and user defined types. Once an array is defined it can be resized with ReDim. Arrays are dynamically allocated which means than a variable or an expression can be used to size them. To view all commands used to manage arrays, see the Array library.

When you define a new array, please note that it will have one more element than you used as parameter, because the numbering of the elements in PureBasic (like in other BASIC's) starts at element 0. For example when you define Dim(10) the array will have 11 elements, elements 0 to 10. This behaviour is different for static arrays in structures .

The new arrays are always locals, which means than Global or Shared commands have to be used if an array declared in the main source need to be used in procedures. It is also possible to pass an array as parameter to a procedure - by using the keyword Array. It will be passed "by reference" (which means, that the array will not be copied, instead the functions in the procedure will manipulate the original array).

To delete the content of an array and release its used memory during program flow, call Dim with array name and 0 elements.

If Dim is used on an existing array, it will reset its contents to zero.

For fast swapping of array contents the Swap keyword is available.

Note: Array bound checking is only done when the runtime Debugger is enabled.

Example:

```
Dim MyArray.l(41)
MyArray(0) = 1
MyArray(1) = 2
```

Example: Multidimensional array

```
Dim MultiArray.b(NbColumns, NbLines)
MultiArray(10, 20) = 10
MultiArray(20, 30) = 20
```

Example: Array as procedure parameter

```
Procedure fill(Array A.l(1), length)  ; the 1 stays for the number of dimensions in the array
  For i = 0 To length
    A(i) = i
  Next i
EndProcedure

Dim A.l(10)
fill(A(), 10)  ; the array A() will be passed as parameter here

Debug A(5)
Debug A(10)
```

Syntax

```
ReDim name.<type>(<expression>, [<expression>], ...)
```

Description

ReDim is used to 'resize' an already declared array while preserving its content. The new size can be larger or smaller, but the number of dimension of the array can not be changed.

If ReDim is used with a multi-dimension array, only its last dimension can be changed.

Example:

```
Dim MyArray.l(1) ; We have 2 elements
MyArray(0) = 1
MyArray(1) = 2

ReDim MyArray(4) ; Now we want 5 elements
MyArray(2) = 3

For k = 0 To 2
  Debug MyArray(k)
Next
```

Chapter 29

Building a DLL

PureBasic allows to create standard Microsoft Windows DLL (Dynamic Linked Library), shared objects (.so) on Linux, and dynamic libraries (.dylib) on MacOS X. The DLL code is like a PureBasic code except than no real code should be written outside of procedure.

When writting a DLL, all the code is done inside procedures. When a procedure should be public (ie: accessible by third programs which will use the DLL), the keyword ProcedureDLL (or ProcedureCDLL if the procedure needs to be in 'CDecl' format, which is not the case of regular Windows DLL) is used instead of Procedure (and DeclareDLL or DeclareCDLL if are used instead of Declare). This is the only change to do to a program.

When this is done, select 'Shared DLL' as output format ('Compiler Option' window in the purebasic editor or /DLL switch in command line) and a DLL with the name you set (in the save-requester when using the IDE) will be created in the selected directory.

Example:

```
ProcedureDLL MyFunction()
  MessageRequester("Hello", "This is a PureBasic DLL !", 0)
EndProcedure

; Now the client program, which use the DLL
;
If OpenLibrary(0, "PureBasic.dll")
  CallFunction(0, "MyFunction")
  CloseLibrary(0)
EndIf
```

For advanced programmers: there is 4 special procedures which are called automatically by Windows when one of the following events happen:

- DLL is attached to a new process
- DLL is detached from a process
- DLL is attached to a new thread
- DLL is detached from a thread

To handle that, it's possible to declare 4 special procedures called: AttachProcess(Instance), DetachProcess(Instance), AttachThread(Instance) and DetachThread(Instance). This means than these 4 procedures names are reserved and can't be used by the programmer for other purposes.

Notes about creating DLL's:

- The declaration of arrays, linked lists or map with Dim , NewList or NewMap must always be done inside the procedure AttachProcess.
- Don't write program code outside procedures. The only exception is the declaration of variables or structures.
- DirectX initialization routines must not be written in the AttachProcess procedure.

Note about returning strings from DLL's:

If you want to return a string out of a DLL, the string has to be declared as Global before using it.

Example:

```
Global ReturnString$

ProcedureDLL.s MyFunction(var.s)
  ReturnString$ = var + " test"
  ProcedureReturn ReturnString$
EndProcedure
```

Without declaring it as Global first, the string is local to the ProcedureDLL and cant be accessed from outside.

When using CallFunction() (or one of its similar CallXXX functions) on a DLL function you will get a pointer on the return string, which you could read with PeekS() .

Example:

```
String.s = PeekS(CallFunction(0, "FunctionName", Parameter1,
 Parameter2))
```

Here a complete code example:

Chapter 30

Enumerations

Syntax

```
Enumeration [<constant> [Step <constant>]]
  #Constant1
  #Constant2 [= <constant>]
  #Constant3
  ...
EndEnumeration
```

Description

Enumerations are very handy to declare a sequence of constants quickly without using fixed numbers. The first constant found in the enumeration will get the number 0 and the next one will be 1 etc.. It's possible to change the first constant number and adjust the step for each new constant found in the enumeration. If needed, the current constant number can be altered by affecting with '=' the new number to the specified constant. As Enumerations only accept integer numbers, floats will be rounded to the nearest integer. The reserved constant #PB_Compiler_EnumerationValue store the next value which will be used in the enumeration. Can be useful to chain several enumerations.

Example: Simple enumeration

```
1  Enumeration
2    #GadgetInfo ; Will be 0
3    #GadgetText ; Will be 1
4    #GadgetOK   ; Will be 2
5  EndEnumeration
```

Example: Enumeration with step

```
1  Enumeration 20 Step 3
2    #GadgetInfo ; Will be 20
3    #GadgetText ; Will be 23
4    #GadgetOK   ; Will be 26
5  EndEnumeration
```

Example: Enumeration with dynamic change

```
Enumeration
  #GadgetInfo       ; Will be 0
  #GadgetText = 15  ; Will be 15
  #GadgetOK         ; Will be 16
EndEnumeration
```

Example: Chained enumerations

```
Enumeration
  #GadgetInfo ; Will be 0
  #GadgetText ; Will be 1
  #GadgetOK   ; Will be 2
EndEnumeration

Enumeration #PB_Compiler_EnumerationValue
  #GadgetCancel ; Will be 3
  #GadgetImage  ; Will be 4
  #GadgetSound  ; Will be 5
EndEnumeration
```

Chapter 31

For : Next

Syntax

```
For <variable> = <expression1> To <expression2> [Step <constant>]
  ...
Next [<variable>]
```

Description

For : Next is used to create a loop within a program with the given parameters. At each loop the <variable> value is increased by a 1, (or of the "Step value" if a Step value is specified) and when the <variable> value is above the <expression2> value, the loop stop.

With the Break command its possible to exit the For : Next loop at any moment, with the Continue command the end of the current iteration can be skiped.

The For : Next loop works only with integer values, at the expressions as well at the Step constant. The Step constant can also be negative.

Example:

```
1  For k = 0 To 10
2    ...
3  Next
```

In this example, the program will loop 11, time (0 to 10), then quit.

Example:

```
1  For k = 10 To 1 Step -1
2    Debug k
3  Next
```

In this example, the program will loop 10 times (10 to 1 backwards), then quit.

Example:

```
a = 2
b = 3
For k = a+2 To b+7 Step 2
  ...
Next k
```

Here, the program will loop 4 times before quitting, (k is increased by a value of 2 at each loop, so the k value is: 4-6-8-10). The "k" after the "Next" indicates that "Next" is ending the "For k" loop. If another variable, is used the compiler will generate an error. It can be useful to nest several "For/Next" loops.

Example:

```
For x=0 To 320
  For y=0 To 200
    Plot(x,y)
  Next y
Next x
```

Note: Be aware, that in PureBasic the value of <expression2> ('To' value) can also be changed inside the For : Next loop. This can lead to endless loops when wrongly used.

Chapter 32

ForEach : Next

Syntax

```
ForEach LinkedList() Or Map()
  ...
Next [LinkedList() Or Map()]
```

Description

ForEach loops through all elements in the specified linked list or map starting from the first element up to the last. If the list or the map is empty, ForEach : Next exits immediately. To view all commands used to manage lists, please click here . To view all commands used to manage maps, please click here .

When used with linked list, it's possible to delete or add elements during the loop. As well it's allowed to change the current element with ChangeCurrentElement() . After one of the mentioned changes the next loop continues with the element following the current element.

With the Break command its possible to exit the ForEach : Next loop at any moment, with the Continue command the end of the current iteration can be skiped.

Example: Linked list

```
NewList Number()

AddElement(Number())
Number() = 10

AddElement(Number())
Number() = 20

AddElement(Number())
Number() = 30

ForEach Number()
  Debug Number() ; Will output 10, 20 and 30
Next
```

Example: Map

```
NewMap Country.s()

Country("US") = "United States"
Country("FR") = "France"
Country("GE") = "Germany"

ForEach Country()
  Debug Country()
Next
```

Chapter 33

General Rules

PureBasic has established rules which never change. These are:

Comments

Comments are marked by ;. All text entered after ; is ignored by the compiler.

Example:

```
  If a = 10 ; This is a comment to indicate something.
```

Keywords

All **keywords** are used for general things inside PureBasic, like creating arrays (Dim) or lists (NewList), or controlling the program flow (If : Else : EndIf). They are not followed by the brackets '()', which are typically for PureBasic **functions**.

Example:

```
    If a = 1       ; If, Else and EndIf are keywords; while 'a = 1'
      ...          ; is a variable used inside an expression.
    Else
      ...
    EndIf
```

Keywords are regularly described in the chapters on the left side of the index page in the reference manual.

Functions

All **functions** must be followed by a (or else it will not be considered as a function, (even for null parameter functions).

Example:

```
EventWindow() ; it is a function.
EventWindow   ; it is a variable.
```

Functions are regularly included in the PureBasic "Command libraries", described on the right side of the index page in the reference manual.

Constants

All constants are preceded by #. They can only be declared once in the source and always keep their predefined values. (The compiler replaces all constant names with their corresponding values when compiling the executable.)

Example:

```
#Hello = 10 ; it is a constant.
Hello  = 10 ; it is a variable.
```

Labels

All labels must be followed by a : (colon). Label names may not contain any operators (+,-,...) as well special characters (ß,ä,ö,ü,...).

Example:

```
I_am_a_label:
```

Expressions

An expression is something which can be evaluated. An expression can mix any variables, constants, or functions, of the same type. When you use numbers in an expression, you can add the Keyword $ sign in front of it to mean a hexadecimal number, or a Keyword % sign sign to mean a binary number. Without either of those, the number will be treated as decimal. Strings must be enclosed by inverted commas.

Example:

```
a = a + 1 + (12 * 3)

a = a + WindowHeight(#Window) + b/2 + #MyConstant

If a <> 12 + 2
  b + 2 >= c + 3
EndIf

a$ = b$ + "this is a string value" + c$

Foo = Foo + $69 / %1001  ; Hexadecimal and binary number usage
```

Concatenation of commands

Any number of commands can be added to the same line by using the : option.

Example:

```
1  If IsCrazy = 0 : MessageRequester("Info", "Not Crazy") : Else :
    MessageRequester("Info", "Crazy") : EndIf
```

Typical words in this manual

Words used in this manual:

<variable> : a basic variable.
<expression> : an expression as explained above.
<constant> : a numeric constant.
<label> : a program label.
<type> : any type, (standard or structured).

Others

- In this guide, all topics are listed in alphabetical order to decrease any search time.

- **Return values** of commands are always Integer if no other type is specified in the Syntax line of the command description.

Chapter 34

Global

Syntax

```
Global[.<type>] <variable[.<type>]> [= <expression>] [, ...]
```

Description

Global provides the ability for variables to be defined as global, i.e., variables defined as such may then be accessed within a Procedure . In this case the command Global must be called for the according variables, **before** the declaration of the procedure. Each variable may have a default value directly assigned to it. If a type is specified for a variable after Global, the default type is changed through the use of this declaration. Global may also be used with arrays , linkedlists and maps .

In order to use local variables within a procedure, which have the same name as global variables, take a look at the Protected and Static keywords.

Example: With variables

```
Global a.l, b.b, c, d = 20

Procedure Change()
  Debug a  ; Will be 10 as the variable is global
EndProcedure

a = 10
Change()
```

Example: With array

```
Global Dim Array(2)

Procedure Change()
  Debug Array(0)  ; Will be 10 as the array is global
EndProcedure

Array(0) = 10
Change()
```

Example: With default type

```
; 'Angle' and 'Position' will be a float, as they didn't have a specified type
;
Global.f Angle, Length.b, Position
```

Chapter 35

Gosub : Return

Syntax

```
Gosub MyLabel

MyLabel:
  ...
Return
```

Description

Gosub stands for 'Go to sub routine'. A label must be specified after Gosub, at that point the program execution continues immediately after the position defined by that label, and will do so until encountering a Return. When a return is reached, the program execution is then transferred immediately below the Gosub.
Gosub is useful when building fast structured code.

Another technique which may be used in order to insert a sub routine into a standalone program component is to use procedures . Gosub may only be used within the main body of the source code, and may not be used within procedures .

Example:

```
a = 1
b = 2
Gosub ComplexOperation
Debug a
End

ComplexOperation:
  a = b*2+a*3+(a+b)
  a = a+a*a
Return
```

Syntax

```
FakeReturn
```

Description

If the command Goto is used within the body of a sub routine, FakeReturn must be used. FakeReturn simulates a return without actually executing a return, and if it is not used, the program will crash. Note: To exit a loop safely, Break should be used instead of Goto.

Example:

```
Gosub SubRoutine1

SubRoutine1:
  ...
  If a = 10
    FakeReturn
    Goto Main_Loop
  EndIf
Return
```

Chapter 36

Handles and Numbers

Numbers

All created objects are identified by an arbitrary number (which is not the object's handle, as seen below). In this manual, these numbers are marked as #Number (for example, each gadget created have a #Gadget number).

The numbers you assign to them do not need to be constants, but they need to be unique for each object in your program (an image can get the same number as a gadget, because these are different types of objects). These numbers are used to later access these objects in your program.

For example, the event handling functions return these numbers:

```
EventGadget()
EventMenu()
EventWindow()
```

Handles

All objects also get a unique number assigned to them by the system. These identifiers are called handles. Sometimes, a PureBasic function doesn't need the number as argument, but the handle. In this manual, such things are mentioned, as an ID.

Example:

```
ImageGadget(#Gadget, x, y, Width, Heigth, ImageID [, Flags])
; #Gadget needs to be the number you want to assign to the Gadget
; ImageID needs to a handle to the image.
```

To get the handle to an object, there are special functions like:

```
FontID()
GadgetID()
ImageID()
ThreadID()
WindowID()
```

Also, most of the functions that create these objects also return this handle as a result, if they were successful. This is only the case if #PB_Any was not used to create the object. If #PB_Any is used, these commands return the object number that was assigned by PB for them, not the handle.

Example:

```
GadgetHandle = ImageGadget(...)
```

Chapter 37

If : Else : EndIf

Syntax

```
If <expression>
  ...
[ElseIf <expression>]
  ...
[Else]
  ...
EndIf
```

Description

The If structure is used to achieve tests, and/or change the programmes direction, depending on whether the test is true or false. ElseIf optional command is used for any number of additional tests if the previous test was not true. The Else optional command is used to execute a part of code, if all previous tests were false. Any number of If structures may be nested together.

Example:

```
1   If a=10
2     Debug "a=10"
3   Else
4     Debug "a<>10"
5   EndIf
```

Example:

```
1   If a=10 And b>=10 Or c=20
2     If b=15
3       Debug "b=15"
4     Else
5       Debug "Other possibility"
6     EndIf
7   Else
8     Debug "Test failure"
9   EndIf
```

Chapter 38

Import : EndImport

Syntax

```
Import "Filename"
  FunctionName.<type>(<parameter>, [, <parameter> [= DefaultValue]...])
   [As "SymbolName"]
  ...
  VariableName.<type> [As "SymbolName"]
EndImport
```

Description

For advanced programmers. Import : EndImport allows to easy declare external functions and variables from a library (.lib) or an object (.obj) file.

Once declared, the imported functions are directly available for use in the program, like any other commands. The compiler doesn't check if the functions really exists in the imported file, so if an error occurs, it will be reported by the linker.

This feature can replace the OpenLibrary() /CallFunction() sequence as it has some advantages: type checking is done, number of of parameters is validated. Unlike CallFunction(), it can deal with double, float and quad without any problem.

The last parameters can have a default value (need to be a constant expression), so if these parameters are omitted when the function is called, the default value will be used.

By default the imported function symbol is 'decorated' in the following way: _FunctionName@callsize.That should work for most of the functions which use the standard call convension (stdcall). If the library is a C one, and the function are not stdcall, the ImportC variant should be used instead. In this case, the default function symbol is decorated like: _FunctionName.

The pseudotypes can be used for the parameters, but not for the returned value.

Example:

```
1   Import "User32.lib"
2
3     ; No need to use 'As' as PureBasic decores the function correctly
4     ; We also define the 'Flags' as optional, with a default value of 0
    (when ommited)
5     ;
6     MessageBoxA(Window.l, Body$, Title$, Flags.l = 0)
```

```
7
8       ; This time PureBasic can't find it alone as the function
9       ; name isn't the same than the one used by the symbol
10      ;
11      MsgBox(Window.l, Body$, Title$, Flags.l) As "_MessageBoxA@16"
12
13    EndImport
14
15    MessageBoxA(0, "Hello", "World") ; We don't specify the flags
16    MsgBox(0, "Hello", "World 2", 0)
```

Example: With pseudotypes

```
1     Import "User32.lib"
2
3       ; We use the 'p-unicode' pseudotype for the string parameters, as
4       ; MessageBoxW() is an unicode only function. The compiler will
5       ; automatically converts the strings to unicode when needed.
6       ;
7       MessageBoxW(Window.l, Body.p-unicode, Title.p-unicode, Flags.l = 0)
8
9     EndImport
10
11    ; It will work correctly in ascii and in unicode mode, even if the
       API is unicode
12    ; only as the compiler will take care of the conversion itself.
13    ;
14    MessageBoxW(0, "Hello", "World")
```

Chapter 39

Includes Functions

Syntax

```
IncludeFile  "Filename"
XIncludeFile "Filename"
```

Description

IncludeFile will include any named source file, at the current place in the code. XIncludeFile is exactly the same except it avoids having to include the same file many times.

Example:

```
1  XIncludeFile "Sources\myfile.pb" ; This will be inserted.
2  XIncludeFile "Sources\myfile.pb" ; This will be ignored along with
    all subsequent calls..
```

This commands are useful, if you want to split your sourcecode into several files, to be able to reuse parts e.g. in different projects.

Syntax

```
IncludeBinary "filename"
```

Description

IncludeBinary will include the named file at the current place in the code. Including should be done inside a Data block.

Example:

```
1  DataSection
2    MapLabel:
3    IncludeBinary "Data\map.data"
4  EndDataSection
```

This command is especially useful in combination with the Catch-commands (currently there are /CatchImage(), /CatchSound(), /CatchSprite()) to include images, sounds, sprites etc. into the executable.

Syntax

```
IncludePath "path"
```

Description

IncludePath will specify a default path for all files included after the call of this command. This can be very handy when you include many files which are in the same directory.

Example:

```
IncludePath  "Sources\Data"
IncludeFile  "Sprite.pb"
XIncludeFile "Music.pb"
```

Chapter 40

Inline x86 ASM

Introduction

PureBasic allows you to include any x86 assembler commands (including MMX and FPU one) directly in the source code, as if it was a real assembler. And it gives you even more: you can use directly any variables or pointers in the assembler keywords, you can put any ASM commands on the same line, ... The syntax is the FAsm (http://flatassembler.net) one, so if you want more informations about the ASM syntax, just read the FAsm guide.

There are serveral ways to activate the inline assembler:
- check the "inline assembler" compiler option via the IDE
- use the /INLINEASM command line compiler switch
- use the compiler directives EnableASM and DisableASM

Rules

You have several rules to closely follow if you want to include asm in a Basic code :

- The used variables or pointers must be declared **before** to use them in an assembler keyword
- When you reference a label , you must put the prefix 'l_' before the name. This is because PureBasic add a 'l_' before a BASIC label to avoid conflit with internal labels. The label need to be referenced in lowercase when using the inlined ASM.

Example:

```
MOV ebx,l_mylabel
...
MyLabel:
```

- The errors in an asm part are not reported by PureBasic but by FAsm. Just check your code if a such error happen.
- With enabled InlineASM you can't use ASM keywords as label names in your source.
- The available volatile registers are: eax, edx and ecx. All others must be always preserved.
- An ASM helpfile could be downloaded here. If you place the 'ASM.HLP' in the 'Help/' folder of PureBasic, you can also get help on Asm keywords with F1. (Note: this feature is only enabled, when InlineASM is enabled).

When using assembler in a procedure , you have to be aware of several important things:

- To return directly the 'eax' register content, just use ProcedureReturn, without any expression. It will let the eax register content untouched and use it as return value.

Example:

```
Procedure.l MyTest()
  MOV eax, 45
  ProcedureReturn  ; The returned value will be 45
EndProcedure
```

- Local variables in PureBasic are directly indexed by the stack pointer, which means than if the stack pointer change via an ASM instruction (like PUSH, POP etc..) the variable index will be wrong and direct variable reference won't work anymore.

- It's possible to pass directly an assembly line to the assembler without being processed by the compiler by using the '!' character at the line start. This allow to have a full access to the assembler directives. When using this, it's possible to reference the local variables using the notation 'p.v_variablename' for a regular variable or 'p.p_variablename' for a pointer

Example:

```
Procedure Test(*Pointer, Variable)
  ! MOV dword [p.p_Pointer], 20
  ! MOV dword [p.v_Variable], 30
  Debug *Pointer
  Debug Variable
EndProcedure

Test(0, 0)
```

Chapter 41

Interfaces

Syntax

```
Interface <name> [Extends <name>]
  <Method[.<type>]()>
  ...
EndInterface
```

Description

Interfaces are used to access Object Oriented modules, such as COM (Component Object Model) or DirectX dynamic libraries (DLL). These types of libraries are becoming more and more common in Windows, and through the use of interfaces, the ability to access these modules easily (and without any performance hit) is realized. It also introduces the necessary basis for Object Oriented programming within PureBasic, but the use of interfaces requires some advanced knowledge. Most of the standard Windows interfaces have already been implemented within a resident file and this allows direct use of these objects.

The optionnal Extends parameter may be used to extend another interface with new functions (Theses functions are commonly called 'methods' in Object Orientied (OO) languages such as C++ or Java). All functions contained within the extended interface are then made available within the new interface and will be placed before the new functions. This is useful in order to do basic inheritance of objects.

A return type may be defined in the interface declaration by adding the type after the method.

SizeOf may be used with Interfaces in order to get the size of the interface and OffsetOf may be used to retrieve the index of the specified function.

The pseudotypes may be used for the parameters of the functions, but not for the return value.

Note: The concept of objects, and the capability provided within PureBasic for their use, has been developed for, and mainly targeted towards, experienced programmers. However, an understanding of these concepts and capabilities are in no way a prerequisite for creating professional software or games.

Example: Basic Example of object call

```
; In order to access an external object (within a DLL for example),
; the objects' interface must first be declared:

Interface MyObject
  Move(x,y)
  MoveF(x.f,y.f)
```

```
    Destroy()
  EndInterface

  ; CreateObject is the function which creates the object, from the DLL,
  ; whose interface has just been defined.
  ; Create the first object...
  ;
  Object1.MyObject = MyCreateObject()

  ; And the second one.
  ;
  Object2.MyObject = MyCreateObject()

  ; Then the functions which have just been defined, may
  ; be used, in order to act upon the desired object.
  ;
  Object1\Move(10, 20)
  Object1\Destroy()

  Object2\MoveF(10.5, 20.1)
  Object2\Destroy()
```

Example: Example with 'Extends'

```
  ; Define a basic Cube interface object.
  ;
  Interface Cube
    GetPosition()
    SetPosition(x)
    GetWidth()
    SetWidth(Width)
  EndInterface

  Interface ColoredCube Extends Cube
    GetColor()
    SetColor(Color)
  EndInterface

  Interface TexturedCube Extends Cube
    GetTexture()
    SetTexture(TextureID)
  EndInterface

  ; The interfaces for 3 different objects have now been defined, these
   objects include:
  ;
  ; - 'Cube' which has the: Get/SetPosition() and Get/SetWidth()
   functions.
  ; - 'ColoredCube' which has the: Get/SetPosition(), Get/SetWidth()
   and Get/SetColor() functions.
  ; - 'TexturedCube' which has the: Get/SetPosition(), Get/SetWidth()
   and Get/SetTexture() functions.
  ;
```

Chapter 42

Macros

Syntax

```
Macro <name> [(Parameter [, ...])]
  ...
EndMacro
```

Description

Macros are a very powerful feature, mainly useful for advanced programmers. A macro is a placeholder for some code (one keyword, one line or even many lines), which will be directly inserted in the source code at the place where a macro is used. In this, it differs from procedures , as the procedures doesn't duplicate the code when they are called.

The Macro : EndMacro declaration must be done before the macro will be called for the first time. Because macros will be completely replaced by their related code at compile time, they are not local to a procedure.

A macro can not have a return type nor typed parameters. When a macro has some parameters, they are replaced in the macro code by the literal expression which is passed to the called macro. No evaluation is done as this stage, which is very important to understand: the evaluation of a line is started once all the macros found on this line are expanded.

The macros are divided into two categories: simple (without parameters) and complex (with parameters, needs the parentheses when calling it). When using no parameters, it's possible to replace any word with another word (or any expression). The macros can be used recursively, but if the parameter passed contain the concatenation character '#', it won't be expanded.

Example: Simple macro

```
Macro MyNot
  Not
EndMacro

a = 0
If MyNot a   ; Here the line will be expanded to : 'If Not a'
  Debug "Ok"
EndIf
```

When using parameters, it's possible to do very flexible macros. The special concatenation character '#' can be used to create new labels or keyword by mixing the macro code and the parameter expression

(spaces are not accepted between each words by the concatenation character). It's also possible to define default values for parameters, so they can be ommited when calling the macro.

Example: Macro with parameter

```
Macro UMsgBox(Title, Body)
  MessageRequester(Title, UCase(Body), 0)
EndMacro

Text$ = "World"
UMsgBox("Hello", "-"+Text$+"-") ; Here the line will be expanded like that:
                                ; 'MessageRequester("Hello", UCase("-"+Text$+"-"), 0)'
```

Example: Macro with default parameter

```
Macro UMsgBox(Title, Body = "Ha, no body specified")
  MessageRequester(Title, UCase(Body), 0)
EndMacro

UMsgBox("Hello") ; Here the line will be expanded like that:
                 ; 'MessageRequester("Hello", UCase("Ha, no body specified"), 0)'
```

Example: Macro parameter concatenation

```
Macro XCase(Type, Text)
  Type#Case(Text)
EndMacro

Debug XCase(U, "Hello")
Debug XCase(L, "Hello")
```

Example: Advanced multiline macro

```
Macro DoubleQuote
  "
EndMacro

Macro Assert(Expression)
  CompilerIf #PB_Compiler_Debugger  ; Only enable assert in debug mode
    If Expression
      Debug "Assert (Line " + Str(#PB_Compiler_Line) + "): " + DoubleQuote#Expression#DoubleQuote
    EndIf
  CompilerEndIf
EndMacro

Assert(10 <> 10) ; Will display nothing
Assert(10 <> 15) ; Should display the assert
```

Chapter 43

Pointers and memory access

Pointers

Using pointers is possible by placing one * in front of the name of the variable . A pointer is a variable that stores a memory address and which is generally associated to a structure .

Example:

```
*MyScreen.Screen = OpenScreen(0, 320, 200, 8, 0)
mouseX = *MyScreen\MouseX  ; Assuming than the Screen structure contains a MouseX field
```

There are only three valid methods to set the value of a pointer:
- Get the result from a function (as shown in the above example)
- Copy the value from another pointer
- Find the address of a variable, procedure or label (as shown below)

Note: Other than in C/C++ in PureBasic the * is **always** part of the variable name. Therefore *ptr and ptr are two different variables. ptr is a variable (regular one) storing a value, *ptr is another variable of pointer type storing an address.

Pointers and memory size

Because pointers receive only addresses as values, the memory size of a pointer is the space allowing to store an absolute address of the processor:
- On 32 bits processors the address space is limited to 32 bits, so a pointer takes 32 bits (4 bytes, like a 'long') in memory
- On newer 64 bits processors it takes 64 bits (8 bytes, like a 'quad') in memory, because the absolute address is on a 64 bits range.

As a consequence the type of a pointer depends of the CPU address mode, (long' on 32 bits CPU and quad' on 64 bits one for example), so a pointer is a variable of type pointer.
It results from this that assigning a type to a pointer (*Pointer.l , *Pointer.b) makes no sense.

Note:
- Every time a memory address needs to be stored in a variable, it should be done through a pointer. This guaranteed address integrity at the compilation time whatever the CPU address mode is.
- PureBasic x86 does not generate 64 bits executables. For PureBasic programs compiled with this, the system grants them only an addressing with 32 bit pointers.

Pointers and structures

By assigning a structure to a pointer (for example *MyPointer.Point) it allows to access any memory address in a structured way (with the operator \').

Example:

```
Define Point1.Point, Point2. Point
*CurrentPoint.Point = @Point1   ; Pointer declaration, associated to a structure and initialized with Point1's address
*CurrentPoint \x = 10           ; Assign value 10 to Point1\x
*CurrentPoint.Point = @Point2   ; move to Point2's address
*CurrentPoint \x = 20           ; Assign value 20 to Point2\x
Debug Point1\x
Debug Point2\x
```

Pointers allow to move, to read and to write easily in memory. Furthermore they allow programmers to reach big quantities of data without supplementary cost further to data duplication. Copying a pointer is much faster.

Pointers and character strings

All variables have a permanent size in memory (2 bytes for Word, 4 bytes for a Long, etc.) except for strings variables with lengths that can change. So string variables are managed by a different way of other variables.
Thus a structure field, that makes reference to a string, store only the memory address of the string instead of the string itself: a such structure field is a pointer towards a string.

Example:

```
Text$ = "Hello"
*Text = @Text$              ; *Text store the address of the string in memory
*Pointer.String = @*Text    ; *Pointer points on *Text
Debug *Pointer\s            ; Display the string living at the address stored in *Pointer (i.e. @Text$)
```

Pointers Arithmetic

Arithmetic operations on the pointers are possible and practical by using SizeOf() .

Example:

```
Dim Array.Point(1)          ; Array of points

*Pointer.Point = @Array()   ; Store the array address
*Pointer\x = 10             ; Change the first array element values
*Pointer\y = 15

*Pointer + SizeOf(Point)    ; Move to the next array element

*Pointer\x = 7              ; Change the second array element values
*Pointer\y = 9

; Display results
For i = 0 To 1
  Debug Array(i)\x
```

```
    Debug Array(i)\y
  Next i
```

Addresses of variables

To find the address of a variable in your code, you use the at symbol (@).A common reason for using this is when you want to pass a structured type variable to a procedure . You must pass a pointer to this variable as you cannot pass structured variables directly.

Example:

```
  Structure astruct
    a.w
    b.l
    c.w
  EndStructure

  Procedure SetB(*myptr.astruct)
    *myptr\b = 69
  EndProcedure

  Define.astruct myvar

  SetB(@myvar)

  Debug myvar\b
```

Addresses of procedures

For advanced programmers. The most common reason to get the address of a procedure is when dealing with the OS at a low-level. Some OSes allow you to specify callback or hook functions (for some operations) which get called by the OS and allows the programmer to extend the ability of the OS routine. The address of a procedure is found in a similar way to variables .

Example:

```
  Procedure WindowCB(WindowID.l, Message.l, wParam.l, lParam.l)
    ; This is where the processing of your callback procedure would be performed
  EndProcedure

  ; A special callback for the Windows OS allowing you to process window events
  SetWindowCallback( @WindowCB() )
```

Addresses of labels

It can also be useful to find the address of labels in your code. This can be because you want to access the code or data stored at that label, or any other good reason you can think of. To find the address of a label, you put a question mark (?) in front of the label name.

Example:

```
Debug "Size of data file = " + Str(?endofmydata - ?mydata)

DataSection
  mydata:
    IncludeBinary "somefile.bin"
  endofmydata:
EndDataSection
```

Chapter 44

NewList

Syntax

```
NewList name.<type>()
```

Description

NewList allows to declare a new dynamic linked list. Each element of the list is allocated dynamically. There are no element limits, so there can be as many as needed. A list can have any Variables standard or structured type. To view all commands used to manage lists, see the LinkedList library.

The new linked list are always locals, which means than Global or Shared commands have to be used if a list declared in the main source need to be used in procedures. It is also possible to pass a linked list as parameter to a procedure by using the keyword List.

For fast swapping of linked list contents the Swap keyword is available.

Example: Simple list

```
NewList MyList.l()

AddElement(MyList())
MyList() = 10

AddElement(MyList())
MyList() = 20

AddElement(MyList())
MyList() = 30

ForEach MyList()
  Debug MyList()
Next
```

Example: List as procedure parameter

```
NewList Test.l()

AddElement(Test())
```

```
Test() = 1
AddElement(Test())
Test() = 2

Procedure DebugList(c.l, List ParameterList.l())

  AddElement(ParameterList())
  ParameterList() = 3

  ForEach ParameterList()
    MessageRequester("List", Str(ParameterList()))
  Next

EndProcedure

DebugList(10, Test())
```

Chapter 45

NewMap

Syntax

```
NewMap name.<type>([Slots])
```

Description

NewMap allows to declare a new map, also known as hashtable or dictionnary. It allows to quickly reference an element based on a key. Each key in the map are unique, which means it can't have two distinct elements with the same key. There are no element limits, so there can be as many as needed. A map can have any standard or structured type. To view all commands used to manage maps, see the Map library.

When using a new key for an assignment, a new element is automatically added to the map. If another element with the same key is already in the map, it will be replaced by the new one. Once an element as been accessed or created, it becomes the current element of the map, and further access to this element can be done without specify the key. This is useful when using structured map, as no more element lookup is needed to access different structure field.

New maps are always locals by default, so Global or Shared commands have to be used if a map declared in the main source need to be used in procedures. It is also possible to pass a map as parameter to a procedure by using the keyword Map.

For fast swapping of map elements the Swap keyword is available.

The optional 'Slots' parameter defines how much slots the map will have have to store its elements. The more slots is has, the faster it will be to access an element, but the more memory it will use. It's a tradeoff depending of how many elements the map will ultimately contains and how fast the random access should be. The default value is 512. This parameter has no impact about the number of elements a map can contain.

Example: Simple map

```
NewMap Country.s()

Country("GE") = "Germany"
Country("FR") = "France"
Country("UK") = "United Kingdom"

Debug Country("FR")

```

```
ForEach Country()
  Debug Country()
Next
```

Example: Map as procedure parameter

```
NewMap Country.s()

Country("GE") = "Germany"
Country("FR") = "France"
Country("UK") = "United Kingdom"

Procedure DebugMap(Map ParameterMap.s())

  ParameterMap("US") = "United States"

  ForEach ParameterMap()
    Debug ParameterMap()
  Next

EndProcedure

DebugMap(Country())
```

Example: Structured map

```
Structure Car
  Weight.l
  Speed.l
  Price.l
EndStructure

NewMap Cars.Car()

; Here we use the current element after the new insertion
;
Cars("Ferrari F40")\Weight = 1000
Cars()\Speed = 320
Cars()\Price = 500000

Cars("Lamborghini Gallardo")\Weight = 1200
Cars()\Speed = 340
Cars()\Price = 700000

ForEach Cars()
  Debug "Car name: "+MapKey(Cars())
  Debug "Weight: "+Str(Cars()\Weight)
Next
```

Chapter 46

Others Commands

Syntax

```
Goto <label>
```

Description

This command is used to transfer the program directly to the labels position. Be cautious when using this function, as incorrect use could cause a program to crash...

Note: To exit a loop safely, you always should use Break instead of Goto.

Syntax

```
End [ExitCode]
```

Description

Ends the program execution correctly. The 'ExitCode' optional parameter can be specified if the program need to returns an error code (widely used in console programs).

The 'ExitCode' can be further used e.g. with the ProgramExitCode() command.

Syntax

```
Swap <expression>, <expression>
```

Description

Swaps the value of the both expression, in an optimized way. The both <expression> have to be a variable , array , linked list or a map element (structured or not) and have to be one of the PureBasic native type like long (.l), quad (.q), string etc.

Example: Swapping of strings

```
Hello$ = "Hello"
World$ = "World"

Swap Hello$, World$

Debug Hello$+" "+World$
```

Example: Swapping of multi-dimensional arrays

```
Dim Array1(5,5)
Dim Array2(5,5)
Array1(2,2) = 10      ; set initial contents
Array2(3,3) = 20

Swap Array1() , Array2()     ; swap the arrays

Debug "Array contents after swapping:"
Debug Array1(3,3)     ; will give 20
Debug Array2(2,2)     ; will give 10
```

Chapter 47

Procedures

Syntax

```
Procedure[.<type>] name(<parameter1[.<type>]> [, <parameter2[.<type>]
   [= DefaultValue]>, ...])
  ...
  [ProcedureReturn value]
EndProcedure
```

Description

A Procedure is a part of code independent from the main code which can have any parameters and it's own variables . In PureBasic, a recurrence is fully supported for the procedures and any procedure can call it itself. At each call of the procedure the variables inside will start with a value of 0 (null). To access main code variables, they have to be shared them by using Shared or Global keywords (see also the Protected and Static keywords).

The last parameters can have a default value (need to be a constant expression), so if these parameters are omitted when the procedure is called, the default value will be used.

Arrays can be passed as parameters using the Array keyword, linked lists using the List keyword and maps using the Map keyword.

A procedure can return a value or a string if necessary. You have to set the type after Procedure and use the ProcedureReturn keyword at any moment inside the procedure. A call of ProcedureReturn exits immediately the procedure, even when its called inside a loop.
ProcedureReturn can't be used to return an array , linked list or a map . For this purpose pass the array, the linked list or the map as parameter to the procedure.

If no value is specified for ProcedureReturn, the returned value will be undefined (see inline assembly for more information).

Note: To return strings from DLLs, see DLLs . For advanced programmers ProcedureC is available and will declare the procedure using 'CDecl' instead of 'StandardCall' calling convension.

Selected procedures can be executed asynchronously to the main program by using of threads .

Example: Procedure with a long variable as return value

```
1  Procedure.l Maximum(nb1.l, nb2.l)
2    If nb1>nb2
3      Result.l = nb1
```

```
    Else
      Result = nb2
    EndIf

    ProcedureReturn Result
  EndProcedure

  Result.l = Maximum(15,30)
  Debug Result
```

Example: Procedure with a string as return value

```
  Procedure.s Attach(String1$, String2$)
    ProcedureReturn String1$+" "+String2$
  EndProcedure

  Result$ = Attach("PureBasic", "Coder")
  Debug Result$
```

Example: Parameter with default value

```
  Procedure a(a, b, c=2)
    Debug c
  EndProcedure

  a(10, 12)       ; 2 will be used as default value for 3rd parameter
  a(10, 12, 15)
```

Example: Linked list as parameter

```
  NewList Test.Point()

  AddElement(Test())
  Test()\x = 1
  AddElement(Test())
  Test()\x = 2

  Procedure DebugList(c.l, List ParameterList.Point())

    AddElement(ParameterList())
    ParameterList()\x = 3

    ForEach ParameterList()
      MessageRequester("List", Str(ParameterList()\x))
    Next

  EndProcedure

  DebugList(10, Test())
```

Example: Array as parameter

```
Dim Table.Point(10, 15)

Table(0,0)\x = 1
Table(1,0)\x = 2

Procedure TestIt(c.l, Array ParameterTable.Point(2))  ; The table support 2 dimensions

  ParameterTable(1, 2)\x = 3
  ParameterTable(2, 2)\x = 4

EndProcedure

TestIt(10, Table())

MessageRequester("Table", Str(Table(1, 2)\x))
```

Syntax

```
Declare[.<type>] name(<parameter1[.<type>]> [, <parameter2[.<type>]>, ...])
```

Description

Sometimes a procedure need to call another procedure which isn't declared before its definition. This is annoying because the compiler will complain 'Procedure <name> not found'. Declare can help in this particular case by declaring only the header of the procedure. Nevertheless, the Declare and real Procedure declaration must be identical (including the correct type).

Example:

```
Declare Maximum(Value1, Value2)

Procedure Operate(Value)
  Maximum(10, 2)      ; At this time, Maximum() is unknown.
EndProcedure

Procedure Maximum(Value1, Value2)
  ProcedureReturn 0
EndProcedure
```

Chapter 48

Protected

Syntax

```
Protected[.<type>] <variable[.<type>]> [= <expression>] [, ...]
```

Description

Protected allows a variable to be accessed only in a Procedure even if the same variable has been declared as Global in the main program. Protected in its function is often known as 'Local' from other BASIC dialects. Each variable can have a default value directly assigned to it. If a type is speficied after Protected, the default type is changed for this declaration. Protected can also be used with arrays , linkedlists and maps .

The value of the local variable will be reinitialized at each procedure call. To avoid this, you can use the keyword Static , to separate global from local variables while keeping their values.

Example: With variable

```
Global a
a = 10

Procedure Change()
  Protected a
  a = 20
EndProcedure

Change()
Debug a ; Will print 10, as the variable has been protected.
```

Example: With array

```
Global Dim Array(2)
Array(0) = 10

Procedure Change()
  Protected Dim Array(2) ; This array is protected, it will be local.
  Array(0) = 20
EndProcedure

```

```
Change()
Debug Array(0) ; Will print 10, as the array has been protected.
```

Chapter 49

Prototypes

Syntax

```
Prototype.<type> name(<parameter>, [, <parameter> [= DefaultValue]...])
```

Description

For advanced programmers. Prototype allows to declare a type which will map a function. It's useful when used with a variable, to do a function pointer (the variable value will be the address the function to call, and can be changed at will).

This feature can replace the OpenLibrary() and CallFunction() sequence as it has some advantages: type checking is done, number of of parameters is validated.

Unlike CallFunction(), it can deal with double, float and quad variables without any problem. To get easily the pointer of a library function, use GetFunction() .

The last parameters can have a default value (need to be a constant expression), so if these parameters are omitted when the function is called, the default value will be used.

By default the function will use the standard call convention (stdcall). If the function pointer will be a C one, the PrototypeC variant should be used instead.

The pseudotypes can be used for the parameters, but not for the returned value.

Example:

```

Prototype.l ProtoMessageBox(Window.l, Body$, Title$, Flags.l = 0)

If OpenLibrary(0, "User32.dll")

  ; 'MsgBox' is a variable with a 'ProtoMessageBox' type
  ;
  MsgBox.ProtoMessageBox = GetFunction(0, "MessageBoxA")

  MsgBox(0, "Hello", "World") ; We don't specify the flags
EndIf
```

Example: With pseudotypes

```
; We use the 'p-unicode' pseudotype for the string parameters, as
; MessageBoxW() is an unicode only function. The compiler will
; automatically converts the strings to unicode when needed.
;
Prototype.l ProtoMessageBoxW(Window.l, Body.p-unicode,
 Title.p-unicode, Flags.l = 0)

If OpenLibrary(0, "User32.dll")

  ; 'MsgBox' is a variable with a 'ProtoMessageBoxW' type
  ;
  MsgBox.ProtoMessageBoxW = GetFunction(0, "MessageBoxW")

  MsgBox(0, "Hello", "World") ; We don't specify the flags
EndIf
```

Chapter 50

Pseudotypes

Description

For advanced programmers. The pseudotypes are a way to ease the programming when dealing with external libraries which needs datatypes which are not internally supported by PureBasic. In this case, it is possible to use the predefined pseudotype which will do the appropriate conversion on the fly, without extra work. As they are not 'real' types, the choosen naming scheme is explicitly different: a 'p-' prefix (for 'pseudo') is part of the type name. The available pseudotypes are:

```
  p-ascii: acts as a string type, but will always convert the string to
    ascii before
            calling the function, even when the program is compiled in
    unicode
 mode.
            It is very useful when accessing a shared library which
    doesn't have unicode
            support in an unicode program for example.

  p-utf8: acts as a string type, but will always convert the string to
    utf8 before calling the
           function. It is very useful when accessing a shared library
    which needs its unicode

           string be passed as UTF8 instead of ascii or unicode strings.

  p-bstr: acts as a string type, but will always convert the string to
    bstr before calling the
           function. It is very useful when accessing a shared library
    which needs bstr parameters,
           like COM components.

  p-unicode: acts as a string type, but will always convert the string
    to unicode
 before
              calling the function, even when the program is compiled in
    ascii mode.
              It is very useful when accessing a shared library which
    only supports unicode
              in an ascii program for example.

  p-variant: acts as a numeric type, will adjust the function call to
    use the variant parameter
```

```
            correctly. It is very useful when accessing a shared
 library which needs 'variant' parameters,
            like COM components.
```

The pseudotypes can only be used with prototypes , interfaces and imported functions. The pseudotypes does the appropriate conversion only if it is necessary: for example using a 'p-ascii' pseudotype in an ascii compiled program makes no difference with regular string type.

Example:

```
Import "User32.lib"

  ; We use the 'p-unicode' pseudotype for the string parameters, as
  ; MessageBoxW() is an unicode only function. The compiler will
  ; automatically converts the strings to unicode when needed.
  ;
  MessageBoxW(Window.l, Body.p-unicode, Title.p-unicode, Flags.l = 0)

EndImport

; It will work correctly in ascii and in unicode mode, even if the API is unicode
; only as the compiler will take care of the conversion itself.
;
MessageBoxW(0, "Hello", "World")
```

Chapter 51

PureBasic objects

Introduction

The purpose of this section is to describe the behaviour, creation, and handling of objects in PureBasic. For the demonstration, we will use the Image object, but the same logic applies to all other PureBasic objects. When creating an Image object, we can do it in two ways: indexed and dynamic.

I. Indexed numbering

The static, indexed way, allows you to reference an object by a predefined numeric value. The first available index number is 0 and subsequent indexes are allocated sequentially. This means that if you use the number 0 and then the number 1000, 1001 indexes will be allocated and 999 (from 1 to 999) will be unused, which is not an efficient way to use indexed objects. If you need a more flexible method, use the dynamic way of allocating objects, as described in section II. The indexed way offers several advantages:

- Easier handling, since no variables or arrays are required.
- 'Group' processing, without the need to use an intermediate array.
- Use the object in procedures without declaring anything in global (if using a constant or a number).
- An object that is associated with an index is automatically freed when re-using that index.

The maximum index number is limited to an upper bound, depending of the object type (usually from 5000 to 60000). Enumerations are strongly recommended if you plan to use sequential contants to identify your objects (which is also recommended).

Example:

```
CreateImage(0, 640, 480) ; Create an image, the n°0
ResizeImage(0, 320, 240) ; Resize the n°0 image
```

Example:

```
CreateImage(2, 640, 480) ; Create an image, the n°2
ResizeImage(2, 320, 240) ; Resize the n°2 image
```

```
CreateImage(2, 800, 800) ; Create a new image in the n°2 index, the old one is automatically free'ed
```

Example:

```
For k = 0 To 9
  CreateImage(k, 640, 480) ; Create 10 differents images, numbered from 0 to 9
  ResizeImage(k, 320, 240) ; Create a new image in the n°2 index, the old one is automatically free'ed
Next
```

Example:

```
#ImageBackground = 0
#ImageButton     = 1

CreateImage(#ImageBackground, 640, 480) ; Create an image (n°0)
ResizeImage(#ImageBackground, 320, 240) ; Resize the background image
CreateImage(#ImageButton    , 800, 800) ; Create an image (n°1)
```

II. Dynamic numbering

Sometimes, indexed numbering isn't very handy to handle dynamic situations where we need to deal with an unknown number of objects. PureBasic provides an easy and complementary way to create objects in a dynamic manner. Both methods (indexed and dynamic) can be used together at the same time without any conflict. To create a dynamic object, you just have to specify the #PB_Any constant instead of the indexed number, and the dynamic number will be returned as result of the function. Then just use this number with the other object functions in the place where you would use an indexed number (except to create a new object). This way of object handling can be very useful when used in combination with a linked list , which is also a dynamic way of storage. A complete example of dynamic objects and linked lists can be found here: MDI ImageViewer.pb.

Example:

```
DynamicImage1 = CreateImage(#PB_Any, 640, 480) ; Create a dynamic image
ResizeImage(DynamicImage1, 320, 240) ; Resize the DynamicImage1
```

Chapter 52

Repeat : Until

Syntax

```
Repeat
  ...
Until <expression> [or Forever]
```

Description

This function will loop until the <expression> becomes true. The number of loops is unlimited. If an endless loop is needed then use the Forever keyword instead of Until. With the Break command, it is possible to exit the Repeat : Until loop during any iteration. With Continue command, the end of the current iteration may be skiped.

Example:

```
1  a=0
2  Repeat
3    a=a+1
4  Until a>100
```

This will loop until "a" takes a value > to 100, (it will loop 101 times).

Chapter 53

Select : EndSelect

Syntax

```
Select <expression1>
  Case <expression> [, <expression> [<numeric expression> To <numeric
   expression>]]
     ...
  [Case <expression>]
     ...
  [Default]
     ...
EndSelect
```

Description

Select provides the ability to determine a quick choice. The program will execute the <expression1> and retain its' value in memory. It will then compare this value to all of the Case <expression> values and if a given Case <expression> value is true, it will then execute the corresponding code and quit the Select structure. Case supports multi-values and value ranges through the use of the optional To keyword (numeric values only). If none of the Case values are true, then the Default code will be executed (if specified).

Note: Select will accept floats as <expression1> but will round them down to the nearest integer (comparisons will be done only with integer values).

Example: Simple example

```
Value = 2

Select Value
  Case 1
    Debug "Value = 1"

  Case 2
    Debug "Value = 2"

  Case 20
    Debug "Value = 20"

  Default
    Debug "I don't know"
```

```
EndSelect
```

Example: Multicase and range example

```
Value = 2

Select Value
  Case 1, 2, 3
    Debug "Value is 1, 2 or 3"

  Case 10 To 20, 30, 40 To 50
    Debug "Value is between 10 and 20, equal to 30 or between 40 and 50"

  Default
    Debug "I don't know"

EndSelect
```

Chapter 54

Using several PureBasic versions on Windows

Overview

It is possible to install several PureBasic versions on your hard disk at the same time. This is useful to finish one project with an older PureBasic version, and start developing a new project with a new PureBasic version.

How to do it

Create different folders like "PureBasic_v3.94" and "PureBasic_v4" and install the related PureBasic version in each folders.

When one "PureBasic.exe" is started, it will assign all ".pb" files with this version of PureBasic. So when a source code is loaded by double-clicking on the related file, the currently assigned PureBasic version will be started. Beside PureBasic will change nothing, which can affect other PureBasic versions in different folders.

To avoid the automatic assignment of ".pb" files when starting the IDE, a shortcut can be created from PureBasic.exe with "/NOEXT" as parameter. The command line options for the IDE are described here .

Note: Since PureBasic 4.10, the settings for the IDE are no longer saved in the PureBasic directory but rather in the %APPDATA%\PureBasic directory. To keep the multiple versions from using the same configuration files, the /P /T and /A switches can be used. Furthermore there is the /PORTABLE switch which puts all files back into the PureBasic directory and disabled the creation of the .pb extension.

Chapter 55

Shared

Syntax

```
Shared <variable> [, ...]
```

Description

Shared allows a variable , an array , a linkedlist or a map to be accessed within a procedure . When Shared is used with an array, a linkedlist or a map, only the name followed by '()' must be specified.

Example: With variable

```
a = 10

Procedure Change()
  Shared a
  a = 20
EndProcedure

Change()
Debug a   ; Will print 20, as the variable has been shared.
```

Example: With array and linkedlist

```
Dim Array(2)
NewList List()
AddElement(List())

Procedure Change()
  Shared Array(), List()
  Array(0) = 1
  List() = 2
EndProcedure

Change()
Debug Array(0)  ; Will print 1, as the array has been shared.
Debug List()    ; Will print 2, as the linkedlist has been shared.
```

Chapter 56

Static

Syntax

```
Static[.<type>] <variable[.<type>]> [= <constant expression>] [, ...]
```

Description

Static allows to create a local persistent variable in a Procedure even if the same variable has been declared as Global in the main program. If a type is speficied after Static, the default type is changed for this declaration. Static can also be used with arrays , linkedlists and maps . When declaring a static array, the dimension parameter has to be a constant value.

The value of the variable isn't reinitialized at each procedure call, means you can use local variables parallel to global variables (with the same name), and both will keep their values. Each variable can have a default value directly assigned to it, but it has to be a constant value.

Beside Static you can use the keyword Protected , to separate global from local variables, but with Protected the local variables will not keep their values.

Example: With variable

```
Global a
a = 10

Procedure Change()
  Static a
  a+1
  Debug "In Procedure: "+Str(a) ; Will print 1, 2, 3 as the variable increments at each procedure call.
EndProcedure

Change()
Change()
Change()
Debug a ; Will print 10, as the static variable doesn't affect global one.
```

Example: With array

```
Global Dim Array(2)
Array(0) = 10

Procedure Change()
  Static Dim Array(2)
  Array(0)+1
  Debug "In Procedure: "+Str(Array(0)) ; Will print 1, 2, 3 as the value of the array field increments at each procedure call.
EndProcedure

Change()
Change()
Change()
Debug Array(0) ; Will print 10, as the static array doesn't affect global one.
```

Chapter 57

Structures

Syntax

```
Structure <name> [Extends <name>]
  ...
EndStructure
```

Description

Structure is useful to define user type, and access some OS memory areas. Structures can be used to enable faster and easier handling of data files. It is very useful as you can group into the same object the informations which are common. Structures fields are accessed with the \ option. Structures can be nested. Statics arrays are supported inside structures.

Dynamic objects like arrays, lists and maps are supported inside structure and are automatically initialized when the object using the structure is created. To declare such field, use the following keywords: Array, List and Map.

The optionnal Extends parameter allows to extends another structure with new fields. All fields found in the extended structure will be available in the new structure and will be placed before the new fields. This is useful to do basic inheritance of structures.

SizeOf can be used with structures to get the size of the structure and OffsetOf can be used to retrieve the index of the specified field.

Please note, that in structures a static array[] doesn't behave like the normal BASIC array (defined using Dim) to be conform to the C/C++ structure format (to allow direct API structure porting). This means that a[2] will allocate an array from 0 to 1 where Dim a(2) will allocate an array from 0 to 2.

When using a lot of structure fields you can use the With : EndWith keywords to reduce the amount of code to type and ease its readibility.

It's possible to perform a full structure copy by using the equal affectation between two structure element of the same type.

ClearStructure can be used to clear a structured memory area. It's for advanced use only, when pointers are involved.

Example:

```
1   Structure Person
2     Name.s
```

```
  ForName.s
  Age.w
EndStructure

Dim MyFriends.Person(100)

; Here the position '0' of the array MyFriend()
; will contain one person and it's own informations

MyFriends(0)\Name = "Andersson"
MyFriends(0)\Forname = "Richard"
MyFriends(0)\Age = 32
```

Example: A more complex structure (Nested and static array)

```
Structure Window
  *NextWindow.Window  ; Points to another window object
  x.w
  y.w
  Name.s[10]  ; 10 Names available (from 0 to 9)
EndStructure
```

Example: Extended structure

```
Structure MyPoint
  x.l
  y.l
EndStructure

Structure MyColoredPoint Extends MyPoint
  color.l
EndStructure

ColoredPoint.MyColoredPoint\x = 10
ColoredPoint.MyColoredPoint\y = 20
ColoredPoint.MyColoredPoint\color = RGB(255, 0, 0)
```

Example: Structure copy

```
Structure MyPoint
  x.l
  y.l
EndStructure

LeftPoint.MyPoint\x = 10
LeftPoint\y = 20

RightPoint.MyPoint = LeftPoint

Debug RightPoint\x
Debug RightPoint\y
```

Example: Dynamic object

```
Structure Person
  Name$
  Age.l
  List Friends$()
EndStructure

John.Person
John\Name$ = "John"
John\Age   = 23

; Now, add some friends to John
;
AddElement(John\Friends$())
John\Friends$() = "Jim"

AddElement(John\Friends$())
John\Friends$() = "Monica"

ForEach John\Friends$()
  Debug John\Friends$()
Next
```

Syntax

```
StructureUnion
  Field1.Type
  Field2.Type
  ...
EndStructureUnion
```

Description

Structure union are only useful for advanced programmers who want to save some memory by sharing some fields inside the same structure. It's like the 'union' keyword in C/C++.

Note: Each field in the StructureUnion declaration can be of a different type .

Example:

```
Structure Type
  Name$
  StructureUnion
    Long.l       ; Each field (Long, Float and Byte) resides at the
    Float.f      ; same address in memory.
    Byte.b       ;
  EndStructureUnion
EndStructure
```

Example: Extended example (date handling)

```
Structure date
  day.s{2}
  pk1.s{1}
  month.s{2}
  pk2.s{1}
  year.s{4}
EndStructure

Structure date2
  StructureUnion
    s.s{10}
    d.date
  EndStructureUnion
EndStructure

Dim d1.date2(5)

d1(0)\s = "05.04.2008"
d1(1)\s = "07.05.2009"

Debug d1(0)\d\day
Debug d1(0)\d\month
Debug d1(0)\d\year

Debug d1(1)\d\day
Debug d1(1)\d\month
Debug d1(1)\d\year

d2.date2\s = "15.11.2010"

Debug d2\d\day
Debug d2\d\month
Debug d2\d\year
```

Chapter 58

Threaded

Syntax

```
Threaded[.<type>] <variable[.<type>]> [= <constant expression>] [, ...]
```

Description

Threaded allows to create a thread based persistent variable , arrays , linkedlists or maps . That means than every thread will have its own version of the object. This is only useful when writing multithreaded programs. If a type is speficied after Threaded, the default type is changed for this declaration.

Each variable can have a default value directly assigned to it, but it has to be a constant value. When declaring a threaded array, the dimension parameter has to be a constant value.

A Threaded object can't be declared in a procedure, its scope is always global.

Example: With variables

```
Threaded Counter

Counter = 128

Procedure Thread(Parameter)

  Debug Counter ; Will display zero as this thread doesn't have used this variable for now
  Counter = 256
  Debug Counter ; Will display 256

EndProcedure

Thread = CreateThread(@Thread(), 0)
WaitThread(Thread) ; Wait for thread ending

Debug Counter ; Will display 128, even if Counter has been changed in the thread
```

Chapter 59

Unicode

Introduction

Unicode is the term used for extended character sets which allows displaying text in many languages (including those with a lot of different characters like Japanese, Korean etc.). It solves the ASCII problem of having a lot of table dedicated for each language by having a unified table where each character has its own place. If an application needs to be used by such countries, it's highly recommended to use the unicode mode when starting the development, as it will reduce the number of bugs due to the conversion from ascii to unicode.

To simplify, unicode can be seen as a big ascii table, which doesn't have 256 characters but up to 65536. Thus, each character in unicode will need 2 bytes in memory (this is important when dealing with pointers for example). The special 'character' (.c) PureBasic type automatically switches from 1 byte to 2 bytes when unicode is enabled.

Here are some links to have a better understanding about unicode (must have reading):

General unicode information
Unicode and BOM

To check if the program is compiled in "Unicode mode", you can use the compiler directives with #PB_Compiler_Unicode constant. To compile a program in "Unicode mode" the related option can be set in the Compiler options .

Unicode and Windows

On Windows, PureBasic internally uses the UCS2 encoding which is the format used by the Windows unicode API, so no conversions are needed at runtime when calling an OS function. When dealing with an API function, PureBasic will automatically use the unicode one if available (for example MessageBox_() will map to MessageBoxW() in unicode mode and MessageBoxA() in Ascii mode). All the API structures and constants supported by PureBasic will also automatically switch to their unicode version. That means than same API code can be compiled either in unicode or ascii without any change.

Unicode is only natively supported on Windows NT and later (Windows 2000/XP/Vista): a unicode program won't work on Windows 95/98/Me. There is solution by using the 'unicows' wrapper dll but it is not yet supported by PureBasic. If the Windows 9x support is needed, the best is to provide two version of the executable: one compiler in ascii, and another in unicode. As it's only a switch to specify, it should be quite fast.

UTF-8

UTF-8 is another unicode encoding, which is byte based. Unlike UCS2 which always takes 2 bytes per characters, UTF-8 uses a variable length encoding for each character (up to 4 bytes can represent one character). The biggest advantage of UTF-8 is the fact it doesn't includes null characters in its coding, so it can be edited like a regular text file. Also the ASCII characters from 1 to 127 are always preserved, so the text is kind of readable as only special characters will be encoded. One drawback is its variable length, so it all string operations will be slower due to needed pre-processing to locate a character is in the text.

PureBasic uses UTF-8 by default when writing string to files in unicode mode (File and Preference libraries), so all texts are fully cross-platform.

The PureBasic compiler also handles both Ascii and UTF-8 files (the UTF-8 files need to have the correct BOM header to be handled correctly). Both can be mixed in a single program without problem: an ascii file can include an UTF-8 file and vice-versa. When developing a unicode program, it's recommended to set the IDE in UTF-8 mode, so all the source files will be unicode ready. As the UTF-8 format doesn't hurt as well when developing ascii only programs, it is not needed to change this setting back.

Chapter 60

Variables and Types

Variables declaration

To declare a variable in PureBasic, simply type its name. You can also specify the type you want this variable to be. Variables do not need to be explicitly declared, as they can be used as "variables on-the-fly". The Define keyword can be used to declare multiple variables in one statement. If you don't assign an initial value to the variable, their value will be 0.

Example:

```
a.b          ; Declare a variable called 'a' from byte (.b) type.
c.l = a*d.w  ; 'd' is declared here within the expression !
```

Notes:

Variable names must not start with a number (0,1,...) and contain no operators (+,-,...) as well special characters (ß,ä,ö,ü,...).

The variables in PureBasic are not case sensitive, so "pure" and "PURE" are the same variable.

If you don't need to change the content of a variable during the program flow (e.g. you're using fixed values for ID's etc.), you can also take a look at constants as an alternative.

To avoid typing errors etc. it's possible to force the PureBasic compiler to always want a declaration of variables, before they are first used. Just use EnableExplicit keyword in your source code to enable this feature.

Basic types

PureBasic allows type variables. Theses types are signed and can be standard integers, float, double, quad and char numbers or even string characters. Here is the list of the native supported types and a brief description :

```
Name          | Extension   | Memory consumption | Range
-------------+------------+--------------------+-----------------------------------
Byte          |    .b       | 1 byte             | -128 to +127
Ascii         |    .a       | 1 byte             | 0 to +255
Character     |    .c       | 1 byte (ascii)     | 0 to +255
Word          |    .w       | 2 bytes            | -32768 to +32767
Unicode       |    .u       | 2 bytes            | 0 to +65535
```

```
Character     |     .c       | 2 bytes (unicode)  | 0 to +65535
Long          |     .l       | 4 bytes            | -2147483648 to
 +2147483647
Integer       |     .i       | 4 bytes (32 bits)  | -2147483648 to
 +2147483647
Integer       |     .i       | 8 bytes (64 bits)  | -9223372036854775808
 to +9223372036854775807
Float         |     .f       | 4 bytes            | unlimited (see below)
Quad          |     .q       | 8 bytes            | -9223372036854775808
 to +9223372036854775807
Double        |     .d       | 8 bytes            | unlimited (see below)
String        |     .s       | string length + 1  | unlimited
Fixed String | .s{Length} | string length      | unlimited
```

Unsigned variables: Purebasic offers native support to unsigned variables for byte and word types trough the ascii (.a) and unicode (.u) types. The character (.c) type (which is an unsigned byte in ascii, and unsigned word in unicode) can also be used as unsigned type.

Notation of string variables: it is possible to use the '$' as last char of a variable name to mark it as string. This way you can use 'a$' and 'a.s' as different string variables. Please note, that the '$' belongs to the variable name and must be always attached, unlike the '.s' which is only needed when the string variable is declared the first time.

```
a.s = "One string"
a$ = "Another string"
Debug a    ; will give "One string"
Debug a$   ; will give "Another string"
```

Note: The floating numbers (floats + doubles) can also be written like this: 123.5e-20

```
value.d = 123.5e-20
Debug value    ; will give 0.00000000000000001235
```

Operators

Operators are the functions you can use in expressions to combine the variables, constants, and whatever else. The table below shows the operators you can use in PureBasic, in no particular order (LHS = Left Hand Side, RHS = Right Hand Side).

Operator = (Equals)

This can be used in two ways. The first is to assign the value of the expression on the RHS to the variable on the LHS. The second way is when the result of the operator is used in an expression and is to test whether the values of the expressions on the LHS and RHS are the same (if they are the same this operator will return a true result, otherwise it will be false).

Example:

```
a = b+c       ; Assign the value of the expression "b+c" to the
 variable "a"
If abc = def ; Test if the values of abc and def are the same, and
 use this result in the If command
```

Operator + (Plus)

Gives a result of the value of the expression on the RHS added to the value of the expression on the LHS. If the result of this operator is not used and there is a variable on the LHS, then the value of the expression on the RHS will be added directly to the variable on the LHS.

Example:

```
number=something+2 ; Adds the value 2 to "something" and uses the result with the equals operator
variable+expression ; The value of "expression" will be added directly to the variable "variable"
```

Operator - (Minus)

Subtracts the value of the expression on the RHS from the value of the expression on the LHS. When there is no expression on the LHS this operator gives the negative value of the value of the expression on the RHS. If the result of this operator is not used and there is a variable on the LHS, then the value of the expression on the RHS will be subtracted directly from the variable on the LHS. This operator cannot be used with string type variables.

Example:

```
var=#MyConstant-foo ; Subtracts the value of "foo" from "#MyConstant" and uses the result with the equals operator
another=another+ -var ; Calculates the negative value of "var" and uses the result in the plus operator
variable-expression ; The value of "expression" will be subtracted directly from the variable "variable"
```

Operator * (Multiplication)

Multiplies the value of the expression on the LHS by the value of the expression on the RHS. If the result of this operator is not used and there is a variable on the LHS, then the value of the variable is directly multipled by the value of the expression on the RHS. This operator cannot be used with string type variables.

Example:

```
total=price*count ; Multiplies the value of "price" by the value of "count" and uses the result with the equals operator
variable*expression ; "variable" will be multiplied directly by the value of "expression"
```

Operator / (Division)

Divides the value of the expression on the LHS by the value of the expression on the RHS. If the result of this operator is not used and there is a variable on the LHS, then the value of the variable is directly

divided by the value of the expression on the RHS. This operator cannot be used with string type variables.

Example:

```
count=total/price ; Divides the value of "total" by the value of "price" and uses the result with the equals operator
variable/expression ; "variable" will be divided directly by the value of "expression"
```

Operator & (Bitwise AND)

You should be familiar with binary numbers when using this operator. The result of this operator will be the value of the expression on the LHS anded with the value of the expression on the RHS, bit for bit. The value of each bit is set according to the table below. Additionally, if the result of the operator is not used and there is a variable on the LHS, then the result will be stored directly in that variable. This operator cannot be used with strings.

```
LHS | RHS | Result
------------------
 0  |  0  |    0
 0  |  1  |    0
 1  |  0  |    0
 1  |  1  |    1
```

Example:

```
; Shown using binary numbers as it will be easier to see the result
a.w = %1000 & %0101 ; Result will be 0
b.w = %1100 & %1010 ; Result will be %1000
bits = a & b ; AND each bit of a and b and use result in equals operator
a & b ; AND each bit of a and b and store result directly in variable "a"
```

Operator || (Bitwise OR)

You should be familiar with binary numbers when using this operator. The result of this operator will be the value of the expression on the LHS or'ed with the value of the expression on the RHS, bit for bit. The value of each bit is set according to the table below. Additionally, if the result of the operator is not used and there is a variable on the LHS, then the result will be stored directly in that variable. This operator cannot be used with strings.

```
LHS | RHS | Result
------------------
 0  |  0  |    0
 0  |  1  |    1
 1  |  0  |    1
 1  |  1  |    1
```

Example:

```
; Shown using binary numbers as it will be easier to see the result
a.w = %1000 | %0101 ; Result will be %1101
b.w = %1100 | %1010 ; Result will be %1110
bits = a | b ; OR each bit of a and b and use result in equals operator
a | b ; OR each bit of a and b and store result directly in variable "a"
```

Operator ! (Bitwise XOR)

You should be familiar with binary numbers when using this operator. The result of this operator will be the value of the expression on the LHS xor'ed with the value of the expression on the RHS, bit for bit. The value of each bit is set according to the table below. Additionally, if the result of the operator is not used and there is a variable on the LHS, then the result will be stored directly in that variable. This operator cannot be used with strings.

```
LHS | RHS | Result
------------------
 0  |  0  |    0
 0  |  1  |    1
 1  |  0  |    1
 1  |  1  |    0
```

Example:

```
; Shown using binary numbers as it will be easier to see the result
a.w = %1000 ! %0101 ; Result will be %1101
b.w = %1100 ! %1010 ; Result will be %0110
bits = a ! b ; XOR each bit of a and b and use result in equals operator
a ! b ; XOR each bit of a and b and store result directly in variable "a"
```

Operator * * (Bitwise NOT)

You should be familiar with binary numbers when using this operator. The result of this operator will be the not'ed value of the expression on the RHS, bit for bit. The value of each bit is set according to the table below. This operator cannot be used with strings.

```
RHS | Result
----------
 0  |    1
 1  |    0
```

Example:

```
; Shown using binary numbers as it will be easier to see the result
a.w = ~%1000 ; Result will be %0111
b.w = ~%1010 ; Result will be %0101
```

Operator () (Brackets)

You can use sets of brackets to force part of an expression to be evaluated first, or in a certain order.

Example:

```
a = (5 + 6) * 3 ; Result will be 33 since the 5+6 is evaluated first
b = 4 * (2 - (3 - 4)) ; Result will be 12 since the 3-4 is evaluated
  first, then the 2-result, then the multiplication
```

Operator < (Less than)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is less than the value of the expression on the RHS this operator will give a result of true, otherwise the result is false.

Operator > (More than)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is more than the value of the expression on the RHS this operator will give a result of true, otherwise the result is false.

Operator <= (Less than or equal to)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is less than or equal to the value of the expression on the RHS this operator will give a result of true, otherwise the result is false.

Operator >= (More than or equal to)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is more than or equal to the value of the expression on the RHS this operator will give a result of true, otherwise the result is false.

Operator <> (Not equal to)

This is used to compare the values of the expressions on the LHS and RHS. If the value of the expression on the LHS is equal to the value of the expression on the RHS this operator will give a result of false, otherwise the result is true.

Operator And (Logical AND)

Can be used to combine the logical true and false results of the comparison operators to give a result shown in the following table.

```
  LHS   |   RHS   | Result
-----------------------
 false | false | false
```

```
 false |  true | false
  true | false | false
  true |  true |  true
```

Operator Or (Logical OR)

Can be used to combine the logical true and false results of the comparison operators to give a result shown in the following table.

```
  LHS   |  RHS   | Result
-----------------------
 false | false | false
 false |  true |  true
  true | false |  true
  true |  true |  true
```

Operator XOr (Logical XOR)

Can be used to combine the logical true and false results of the comparison operators to give a result shown in the following table.

```
  LHS   |  RHS   | Result
-----------------------
 false | false | false
 false |  true |  true
  true | false |  true
  true |  true | false
```

Operator Not (Logical NOT)

The result of this operator will be the not'ed value of the logical on the RHS. The value is set according to the table below. This operator cannot be used with strings.

```
  RHS   | Result
---------------
 false |  true
  true | false
```

Operator « (Arithmetic shift left)

Shifts each bit in the value of the expression on the LHS left by the number of places given by the value of the expression on the RHS. Additionally, when the result of this operator is not used and the LHS contains a variable, that variable will have its value shifted. It probably helps if you understand binary numbers when you use this operator, although you can use it as if each position you shift by is multiplying by an extra factor of 2.

Example:

```
1   a=%1011 << 1 ; The value of a will be %10110. %1011=11, %10110=22
2   b=%111 << 4 ; The value of b will be %1110000. %111=7, %1110000=208
3   c.l=$8000000 << 1 ; The value of c will be 0. Bits that are shifted
     off the left edge of the result are lost.
```

Operator » (Arithmetic shift right)

Shifts each bit in the value of the expression on the LHS right by the number of places given by the value of the expression on the RHS. Additionally, when the result of this operator is not used and the LHS contains a variable, that variable will have its value shifted. It probably helps if you understand binary numbers when you use this operator, although you can use it as if each position you shift by is dividing by an extra factor of 2.

Example:

```
d=16 >> 1 ; The value of d will be 8. 16=%10000, 8=%1000
e.w=%10101010 >> 4 ; The value of e will be %1010. %10101010=170,
  %1010=10. Bits shifted out of the right edge of the result are lost
  (which is why you do not see an equal division by 16)
f.b=-128 >> 1 ; The value of f will be -64. -128=%10000000,
  -64=%11000000. When shifting to the right, the most significant bit
  is kept as it is.
```

Operator % (Modulo)

Returns the remainder of the RHS by LHS integer division.

Example:

```
a=16 % 2 ; The value of a will be 0 as 16/2 = 8 (no remainder)
b=17 % 2 ; The value of a will be 1 as 17/2 = 8*2+1 (1 is remaining)
```

Operators priorities

```
Priority Level |      Operators
---------------+---------------------
     8 (high)  |       ~, -
     7         |     <<, >>, %, !
     6         |        |, &
     5         |        *, /
     4         |        +, -
     3         | >, >=, <, <=, =, <>
     2         |        Not
     1 (low)   |  And, Or, XOr
```

Structured types

Build structured types, via the Structures option. More information can be located in the structures chapter .

Special information about Floats and Doubles

A floating point number is stored in a way that makes the binary point ”float” around the number, so that it is possible to store very large numbers or very small numbers. However, you cannot store very large numbers with very high accuracy (big and small numbers at the same time, so to speak).

Another limitation of floating point numbers is that they still work in binary, so they can only store numbers exactly which can be made up of multiples and divisions of 2. This is especially important to realise when you try to print a floating point number in a human readable form (or when performing operations on that float) - storing numbers like 0.5 or 0.125 is easy because they are divisions of 2. Storing numbers such as 0.11 are more difficult and may be stored as a number such as 0.10999999. You can try to display to only a limited range of digits, but do not be surprised if the number displays different from what you would expect!

This applies to floating point numbers in general, not just those in PureBasic.

Like the name says the doubles have double-precision (64 bit) compared to the single-precision of the floats (32 bit). So if you need more accurate results with floating point numbers use doubles instead of floats.

The exact range of values, which can be used with floats and doubles to get correct results from arithmetic operations, looks as follows:

Float: +- 1.175494e-38 till +- 3.402823e+38
Double: +- 2.2250738585072013e-308 till +- 1.7976931348623157e+308

More informations about the 'IEEE 754' standard you can get on Wikipedia.

Chapter 61

We start programming...

Overview

The following topics in this chapter should give you some ideas, where to start with PureBasic. It shouldn't replace any larger tutorial or the massive informations, which can be found on the popular PureBasic forums. So the following information texts are short, but they include some "keywords" and links to futher informations, which are included in this reference manual. This chapter covers only a small part of the 1100+ commands available in PureBasic! :)

Topics in this chapter:

- First steps
- Decisions and Conditions
- Loops
- Storing data in memory
- Input & Output
- Displaying text output
- Displaying graphics output & simple drawing
- Procedures
- Reading and writing files
- ...

First steps with the Debug output (variables & operators)

Normally we would present here the typical "Hello World". You want to see it? Ok here we go with two examples:

"Hello World" in the Debug output window :

```
Debug "Hello World!"
```

"Hello World" in a MessageRequester :

```
MessageRequester("", "Hello World!")
```

We now continue with a short example using the available variable types, arithmetic operators and displaying the result:

```
a = 5
b = 7
```

```
c = 3

d = a + b    ; we use the values stored in variables 'a' and 'b' and
  save the sum of them in 'd'
d / c        ; we directly use the value of 'd' (= 12) divided by the
  value of 'c' (= 3) and save the result in 'd'

Debug d     ; will give 4
```

This way you have used variables "on the fly". To force the compiler always declaring variables before their first use, just use the keyword EnableExplicit .

Decisions and Conditions

... There are different ways of processing data you got via user input or on another way (loading from a file, ...). The common arithmetic functions (+, -, *, /, ...) can be combined with ...conditions (Bedingungen). You can use the "If : Else/ElseIf : EndIf" set of keywords or the the "Select : Case/Default : EndSelect" keywords, just you what is the best for your situation!

Loops

... Data, Events or many other things can also be processed using loops, which are always checking for a specific condition. This can be: "Repeat : Until/Forever", "While : Wend", "For : Next", ForEach : Wend", ...

Storing data in memory

...

Input & Output

Every PureBasic application can communicate and interact with the user on different ways.

Thereby we distinguish between

a) the pure output of informations
b) the interaction of the PureBasic application with the user, when user-input will be taken and the results will be outputted again.

It's not possible anymore, to use a simple "PRINT" command to output some things on the screen, like it was possible on DOS operating systems (OS) without a graphical user interface (GUI) years ago. Today such a GUI is always present, when you use an actual OS like Windows, Mac OSX oder Linux.

For the output of informations we have different possibilities:

- debug window (only possible during programming with PureBasic)
- MessageRequester (output of shorter text messages in a requester window)
- files (for saving the results in a text-file, etc.)
- console (for simple and almost non-graphic text output, most similar to earlier DOS times)
- windows and gadgets (standard windows with GUI elements on the desktop of the OS, e.g. for applications)
- Screen (Output of text and graphics directly on the screen, e.g. for games)

To be able to record and process input by the user of the PureBasic application, the three last-mentioned output options have also the possibility to get user-input:

- in the console using Input()
- in the window using WaitWindowEvent() / WindowEvent() , which can get the events occured in the window, like clicking on a button or the entered text in a StringGadget
- in the graphics screen using the keyboard
- the is as well the possibility to get user-input using the InputRequester

Displaying text output

In previous chapter "Input & Output" you already got an overview about the different possibilities to output text to the user.

First we will store some data in memory, which will be used later in the output examples:

And now we will see several examples of "How to output" text (using the previous stored data):

Displaying graphics output & simple drawing

...

Procedures

...

Reading and writing files

...

...

...

Chapter 62

While : Wend

Syntax

```
While <expression>
  ...
Wend
```

Description

Wend will loop until the <expression> becomes false. A good point to keep in mind with a While test is that if the first test is false, then the program will never enter the loop and will skip this part. A Repeat loop is executed at least once, (as the test is performed after each loop).

With the Break command it is possible to exit the While : Wend loop during any interation, with the Continue command the end of the current iteration may be skiped.

Example:

```
1   b = 0
2   a = 10
3   While a = 10
4     b = b+1
5     If b=10
6       a=11
7     EndIf
8   Wend
```

This program loops until the 'a' value is <> 10. The value of 'a' becomes 11 when b=10, the program will loop 10 times.

Chapter 63

Windows Message Handling

Introduction

The messages for your program will be sent by Windows into a queue, which is worked off only if you want this. Windows sends thousand messages to your program without noticing this directly.

For example if you change the status of a gadget (identical whether you add a new entry or change the image of an ImageGadget), a message is sent to the queue of your program.

There are two possibilities to receive and to process the Windows messages in PureBasic: WaitWindowEvent() and WindowEvent() . The difference is, that WaitWindowEvent() waits till a message arrives and WindowEvent() continues to work also so. The messages in the queue will be processed however only, after you have called WindowEvent() or WaitWindowEvent().

Specials of WindowEvent()

The command WindowEvent() don't wait, until a message arrives, but checks only whether one is in the queue. If yes, the message is processed and WindowEvent() gives back the number of the message. If no message lines up, then zero (0) is given back.

Example:

```
While WindowEvent() : Wend
```

cause, that WindowEvent() becomes called as long till it gives back 0, i.e. until all messages of the queue are processed.

It doesn't reach, if you insert a simple 'WindowEvent()' after a SetGadgetState() to process this 1 message. Firstly there still can be other messages in the queue, which have arrived before, and secondly Windows also sends quite a number of other messages, we don't have to take care of... which nevertheless are in the queue.

A simple call of

```
WindowEvent()
```

doesn't reach, the code then runs correct under circumstances on one Windows version, but on another version not. The different Windows versions are internally very different, so that one version only sends 1 message but another Windows version sends 5 messages for the same circumstance.

Because of this one always uses for updating:

```
While WindowEvent() : Wend
```

Of course there is also the alternative

```
Repeat : Until WindowEvent() = 0
```

possible, what isn't rather usual however.

The described method While : WindowEvent() : Wend is frequently useful in connection with the command Delay() , where the loop is inserted BEFORE the Delay(), e.g. to firstly wait for the update of an ImageGadget after changing an image with SetGadgetState().

Chapter 64

With : EndWith

Syntax

```
With <expression>
  ...
EndWith
```

Description

With : EndWith blocks may be used with structure fields in order to reduce the quantity of code and to improve its' readability. This is a compiler directive, and works similarly to a macro , i.e., the specified expression is automatically inserted before any anti-slash '\' character which does not have a space or an operator preceding it. The code behaves identically to its' expanded version. With : EndWith blocks may not be nested, as this could introduce bugs which are difficult to track under conditions where several statements have been replaced implicitly.

Example:

```
Structure Person
  Name$
  Age.l
  Size.l
EndStructure

Friend.Person

With Friend
  \Name$ = "Yann"
  \Age   = 30
  \Size  = 196

  Debug \Size+\Size
EndWith
```

Example: Complex example

```
Structure Body
  Weight.l
  Color.l
```

```
  Texture.l
EndStructure

Structure Person
  Name$
  Age.l
  Body.Body[10]
EndStructure

Friend.Person

For k = 0 To 9
  With Friend\Body[k]
    \Weight = 50
    \Color  = 30
    \Texture = \Color*k

    Debug \Texture
  EndWith
Next
```

Part IV

Library Reference

Chapter 65

2DDrawing

The 2D drawing library contains all the 2D rendering operations that can be performed on a visual area. Drawing a line, a box, a circle or even text is considered a 2D rendering operation.

The output of the drawing functions is possible on a window, a screen, a sprite, an image or the printer. More information can be found at StartDrawing() .

Note: The drawing process starts after calling StartDrawing() and must end with StopDrawing() .

65.1 BackColor

Syntax

```
BackColor(Color)
```

Description

Set the default background color for graphic functions and text display. RGB() or RGBA() can be used to get a valid color value. Whether or not the alpha channel is used depends on the drawing mode .

A color table with common colors is available here .

65.2 Box

Syntax

```
Box(x, y, Width, Height [, Color])
```

Description

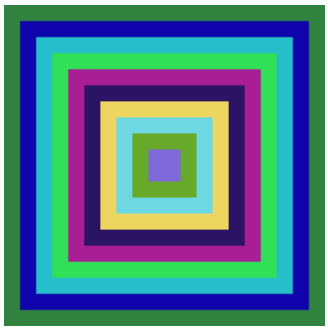
Draw a box of given dimensions on the current output. The filling mode is determined by DrawingMode() . The current output is set with StartDrawing() .

If the 'Color' parameter is not specified, the default color set with FrontColor() will be used. RGB() or RGBA() can be used to get a valid color value.

The RoundBox() command can be used to draw a box with rounded corners.

Example:

```
If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
    y = 0
    For x = 0 To 95 Step 10
      Box(x, y, 200-2*x, 200-2*y, RGB(Random(255), Random(255),
 Random(255)))
      y + 10        ; the same as y = y + 10
    Next x
    StopDrawing()
    ImageGadget(0, 0, 0, 200, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.3 RoundBox

Syntax

```
RoundBox(x, y, Width, Height, RoundX, RoundY [, Color])
```

Description

Draw a box of the given dimensions with rounded corners on the current output. The filling mode is determined by DrawingMode() . The current output is set with StartDrawing() .

The 'RoundX' and 'RoundY' parameters specify the radius of the rounded corners in the x and y direction. If the 'Color' parameter is not specified, the default color set with FrontColor() will be used. RGB() or RGBA() can be used to get a valid color value.

The Box() command can be used to draw a box without rounded corners.

Example:

```
If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
```

```
    If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
      y = 0
      For x = 0 To 95 Step 10
        RoundBox(x, y, 200-2*x, 200-2*y, 20, 20, RGB(Random(255),
  Random(255), Random(255)))
        y + 10
      Next x
      StopDrawing()
      ImageGadget(0, 0, 0, 200, 200, ImageID(0))
    EndIf

    Repeat
      Event = WaitWindowEvent()
    Until Event = #PB_Event_CloseWindow
  EndIf
```

65.4 Circle

Syntax

```
Circle(x, y, Radius [, Color])
```

Description

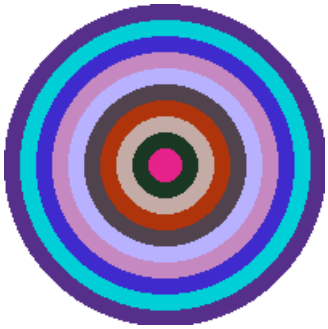
Draw a circle on the current output at the position x,y with the size of Radius. The filling mode is determined by DrawingMode() . The current output is set with StartDrawing() .

If the 'Color' parameter is not specified, the default color set with FrontColor() will be used. RGB() or RGBA() can be used to get a valid color value.

Example:

```
  If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
   #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
    If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
      Box(0, 0, 200, 200, RGB(255, 255, 255))
      For Radius = 100 To 10 Step -10
        Circle(100, 100, Radius, RGB(Random(255), Random(255),
  Random(255)))
      Next
      StopDrawing()
      ImageGadget(0, 0, 0, 200, 200, ImageID(0))
    EndIf
```

```
10
11      Repeat
12        Event = WaitWindowEvent()
13      Until Event = #PB_Event_CloseWindow
14    EndIf
```



65.5 DrawImage

Syntax

```
DrawImage(ImageID, x, y [, Width, Height])
```

Description

Draw the specified image at the given x,y position. The image ID can be easily obtained by using the ImageID() function from the image library.

If the optional 'Width' and 'Height' parameters are specified, then the image can be resized on-the-fly before being displayed. The original image size doesn't change.

65.6 DrawAlphaImage

Syntax

```
DrawAlphaImage(ImageID, x, y [, ConstAlpha])
```

Description

Draw the specified image at the given x,y position. The image ID can be easily obtained by using the ImageID() function from the image library.

The Image will be alpha-blended onto the drawing background (even if the current drawing mode is not #PB_2DDrawing_AlphaBlend). The optional "ConstAlpha" parameter can be used to adjust the overall transparency of the image. The value can range from 0 (fully transparent) to 255 (fully opaque). Even images that have no alpha channel themselves can be drawn with transparency this way.

This command cannot be used to draw Icons (loaded from .ico files).

65.7 DrawingBuffer

Syntax

```
MemoryAddress = DrawingBuffer()
```

Description

Returns the MemoryAddress of the drawing buffer. It can be directly in video memory if the output is ScreenOutput() or SpriteOutput() and allows very fast pixel manipulation. With ImageOutput() this command allows direct access to the pixels of the target image.

If the 'MemoryAddress' is 0, then direct memory access isn't available on the current output. This function is for advanced programmers only. To get more information about the buffer, the following functions are available: DrawingBufferPixelFormat() and DrawingBufferPitch() .

This function has to be called again if other commands of this library where used since the last pixel manipulation.

For an example of how the drawing buffer can be used to create a nice visual effect, take a look here:

65.8 DrawingBufferPitch

Syntax

```
Result = DrawingBufferPitch()
```

Description

Returns the real length of one line (in bytes) of the current buffer. The result can be larger than the width of the output multiplied by the pixel size because of additional padding.

DrawingBuffer() must be called before using this function.

65.9 DrawingBufferPixelFormat

Syntax

```
Result = DrawingBufferPixelFormat()
```

Description

Returns the pixel format of the current output. It can be a combination (with bitwise or) of the following values:

```
#PB_PixelFormat_8Bits        ; 1 bytes per pixel, palletised
#PB_PixelFormat_15Bits       ; 2 bytes per pixel
#PB_PixelFormat_16Bits       ; 2 bytes per pixel
#PB_PixelFormat_24Bits_RGB   ; 3 bytes per pixel (RRGGBB)
#PB_PixelFormat_24Bits_BGR   ; 3 bytes per pixel (BBGGRR)
#PB_PixelFormat_32Bits_RGB   ; 4 bytes per pixel (RRGGBB)
#PB_PixelFormat_32Bits_BGR   ; 4 bytes per pixel (BBGGRR)
```

```
#PB_PixelFormat_ReversedY    ; The Y-Coordinate of the output is
 reversed in memory (the bottom row is stored first).
```

Important: DrawingBuffer() must be called before using this function.

The following examples show how to handle the result:

```
If DrawingBufferPixelFormat() =
 #PB_PixelFormat_32Bits_RGB|#PB_PixelFormat_ReversedY
  ; 32bit RGB with reversed Y coordinate
EndIf

If DrawingBufferPixelFormat() = #PB_PixelFormat_32Bits_RGB
  ; 32bit RGB without reversed Y coordinate
EndIf

If DrawingBufferPixelFormat() & #PB_PixelFormat_32Bits_RGB
  ; 32bit RGB, with or without reversed Y coordinate
EndIf
```

65.10 DrawingFont

Syntax

```
DrawingFont(FontID)
```

Description

Sets the current font to the given FontID. All new text will be rendered with the new specified font.

The FontID can be easily obtained with the FontID() function from the font library. The font should be loaded with LoadFont() before using this function. To restore the default system font, #PB_Default can be used as FontID.

65.11 DrawingMode

Syntax

```
DrawingMode(Mode)
```

Description

Change the drawing mode for text and graphics output.

Mode can be a combination of the following flags:

#PB_2DDrawing_Default

> This is the default drawing mode when the drawing starts. Text is displayed with a solid background and graphic shapes are filled. If the current output has an alpha channel, the drawing operations will only modify the color components and leave the alpha channel unchanged.

#PB_2DDrawing_Transparent

If this flag is set then the background will be transparent with the DrawText() command.

#PB_2DDrawing_XOr

Enables the XOR mode. All graphics will be XOR'ed with the current background. This mode cannot be combined with the below alpha channel modes.

Note: This mode does not work with PrinterOutput() .

#PB_2DDrawing_Outlined

If this flag is set then shapes will be drawn as outlines only and not filled. This applies to commands such as Circle , Box , etc.

Note: The following modes only work with ImageOutput() . They are ignored for all other outputs:

#PB_2DDrawing_AlphaBlend

The drawing operations will be alpha-blended onto the background. The RGBA() command can be used to specify colors with alpha transparency in commands like FrontColor() , Box() , DrawText() etc.

#PB_2DDrawing_AlphaClip

The drawing operations will be alpha-blended onto the background like with the #PB_2DDrawing_AlphaBlend mode, with the addition that the alpha channel of the drawing output acts as a mask. This means that areas of the output that are transparent before the blending will also remain transparent afterwards. If the drawing output has no alpha channel then this mode acts just like the #PB_2DDrawing_AlphaBlend mode.

#PB_2DDrawing_AlphaChannel

The drawing operations will only modify the alpha channel of the drawing output. All color information is ignored. For example drawing a circle with a color value of RGBA (0, 0, 0, 0) will "cut" a hole into the drawing output by making the circle area fully transparent. If the drawing output has no alpha channel then no drawing will have an effect in this mode.

#PB_2DDrawing_AllChannels

The drawing operations will modify the color channels and the alpha channel of the drawing output. The content of the channels is replaced by the drawing operation without any blending. Drawing in this mode has the same effect as drawing first using the #PB_2DDrawing_Default mode and then drawing the same operation using the #PB_2DDrawing_AlphaChannel mode. If the drawing output has no alpha channel then this mode is equivalent to the #PB_2DDrawing_Default mode.

#PB_2DDrawing_Gradient

This mode allows drawing with a gradient instead of a solid color. The gradient shape can be defined with commands such as LinearGradient() , CircularGradient() etc. and the colors used in the gradient can be set with GradientColor() . The color parameters given to the individual drawing commands will be ignored in this mode. This mode can be combined with the above alpha channel modes to have gradients with semi-transparent colors.

#PB_2DDrawing_CustomFilter

In this mode, the drawing of the pixels can be defined by a custom procedure with the CustomFilterCallback() command. This allows the implementation of custom drawing effects while still using the default functions to do the actual drawing.

To use several modes at once, you have to use the '||' (OR) operator. The following is an example for XOR'ed outlined shapes:

```
DrawingMode(#PB_2DDrawing_Outlined | #PB_2DDrawing_XOr)
```

65.12 DrawRotatedText

Syntax

```
DrawRotatedText(x, y, Text$, Angle.f [, Color])
```

Description

Displays the given string on the current output at the given angle.

The x and y parameter refer to the top/left corner of the string to display. This is also the point around which the rotation will take place. The Angle.f parameter specifies the angle in degrees to rotate counter-clockwise starting from the normal text orientation . The 'Color' parameter and can be used to override the color set with FrontColor() . The background of the rotated text is always transparent.

Example:

```
If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 200, 200, RGB(255, 255, 255))
    For Angle = 0 To 360 Step 45
      DrawRotatedText(100, 100, "Hello World!", Angle, RGB(0, 0, 0))
    Next Angle
    StopDrawing()
    ImageGadget(0, 0, 0, 200, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```

65.13 FillArea

Syntax

```
FillArea(x, y, BorderColor [, Color])
```

Description

Fill an arbitrary area starting from x,y position until the BorderColor is encountered. This is useful for filling any kind of shape.

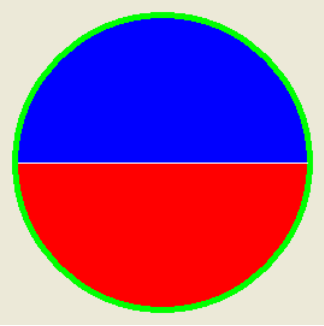
If the 'Color' parameter is not specified, the default color set with FrontColor() will be used. RGB() or RGBA() can be used to get a valid color value.

If the 'BorderColor' parameter is set to -1, then the area defined by the color found at the (x,y) coordinates will be filled. On 32bit images, the alpha channel is ignored when determining whether a pixel counts as a border pixel or not.

Note: This command does not work with PrinterOutput() .

Example:

```
If OpenWindow(0, 0, 0, 300, 300, "2DDrawing Example", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 300, 300) And StartDrawing(ImageOutput(0))
    Box(0, 0, 300, 300, RGB(255, 255, 255))

    Circle(150, 150, 125 ,$00FF00)
    Circle(150, 150, 120 ,$FF0000)
    LineXY(30, 150, 270, 150, $FFFFFF)
    FillArea(150, 155, -1, $0000FF) ; Replace -1 by $00FF00, and compare the result

    StopDrawing()
    ImageGadget(0, 0, 0, 300, 300, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.14 GrabDrawingImage

Syntax

```
Result = GrabDrawingImage(#Image, x, y, Width, Height)
```

Description

Create a new #Image with the content of the given area in the current output. If #PB_Any is used as '#Image' parameter, the new image number will be returned as 'Result'.

Any parts of the specified area that are outside of the drawing output will be undefined in the created images. Also if the current output is WindowOutput() , any part of the window that is currently not visible may be undefined in the resulting image.

Note: This command does not work with PrinterOutput() .

65.15 StartDrawing

Syntax

```
Result = StartDrawing(OutputID)
```

Description

Change the current drawing output to the specified output. After setting this, all drawing functions are rendered to this output. Once all drawing operations are finished, StopDrawing() must be called.

The 'Result' should be always evaluated before proceeding with drawing operations. If the 'Result' is zero, then the output can't be used for drawing, else it's ok to commence drawing.

The back color is set to black (ie: RGB(0,0,0)) and the front color is set to white (ie: RGB(255,255,255)).

OutputID can be obtained with the following functions:

```
WindowOutput()
: Graphics will be rendered directly on the Window
ScreenOutput()
: Graphics will be rendered directly on the Screen (for games)
SpriteOutput()
: Graphics will be rendered directly on the Sprite (for games)
ImageOutput()
```

```
    : Graphics will be rendered directly on the Image data (see
    CreateImage()
)
  PrinterOutput()
 : Graphics will be rendered directly on the Printer
```

65.16 DrawText

Syntax

```
Result = DrawText(x, y, Text$ [, FrontColor [, BackColor]])
```

Description

Display the given string on the current output at the given x,y position. The current output is set with StartDrawing() .

DrawingMode() can be used to set the transparent background mode. DrawingFont() can be used to set the font. This function uses the default colors set with FrontColor() and BackColor() . If DrawingMode() is set to non-transparent background and the current drawing mode uses the alpha channel then the text is first blended onto the background and then applied to the drawing output. RGB() or RGBA() can be used to get a valid color value.

The new x position of the text cursor (ie: just after the last printed character) is returned in 'Result' to allow easy text concatenation, if required.

The 'FrontColor' and 'BackColor' are optional parameters and can be used to override the colors set with FrontColor() and BackColor() . The current back and front colors aren't changed.

Example:

```
1   If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
     #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2     If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
3       DrawingMode(#PB_2DDrawing_Transparent)
4       Box(0, 0, 200, 200, RGB(255, 255, 255))
5       For i = 1 To 30
6         DrawText(Random(200), Random(200), "Hello World!",
     RGB(Random(255), Random(255), Random(255)))
7       Next i
8       StopDrawing()
9       ImageGadget(0, 0, 0, 200, 200, ImageID(0))
10    EndIf
11
12    Repeat
13      Event = WaitWindowEvent()
14    Until Event = #PB_Event_CloseWindow
15  EndIf
```

65.17 Ellipse

Syntax

```
Ellipse(x, y, RadiusX, RadiusY [, Color])
```

Description

Draw an ellipse at the position x,y with size of RadiusX and RadiusY. The RadiusX and RadiusY do not include the center point of the ellipse. The filling mode is determined by DrawingMode() . The current output is set with StartDrawing() .

If the 'Color' parameter is not specified, the default color set with FrontColor() will be used. RGB() or RGBA() can be used to get a valid color value.

Example:

```
If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 200, 200, RGB(255, 255, 255))
    For radius=50 To 10 Step -10
      Ellipse(100, 100, radius*2, radius, RGB(Random(255),
 Random(255), Random(255)))
    Next radius
    StopDrawing()
    ImageGadget(0, 0, 0, 200, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```

65.18 FrontColor

Syntax

```
FrontColor(Color)
```

Description

Set the default color for graphic functions and text display. RGB() or RGBA() can be used to get a valid color value. Whether or not the alpha channel is used depends on the drawing mode .

A color table with common colors is available here .

65.19 Line

Syntax

```
Line(x, y, Width, Height [, Color])
```

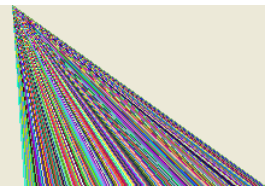
Description

Draw a line of given dimensions on the current output. The Width and Height include the starting point of the line, so a Height of 1 draws a horizontal line while a Height of 0 draws nothing at all. The current output is set with StartDrawing() .

If the 'Color' parameter is not specified, the default color set with FrontColor() will be used. RGB() or RGBA() can be used to get a valid color value.

Example:

```
If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 200, 200, RGB(255, 255, 255))
    For Width = 1 To 180 Step 5
      Line(10, 10, Width, 180, RGB(Random(255), Random(255),
 Random(255)))
    Next Width
    StopDrawing()
    ImageGadget(0, 0, 0, 200, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.20 LineXY

Syntax

```
LineXY(x1, y1, x2, y2 [, Color])
```

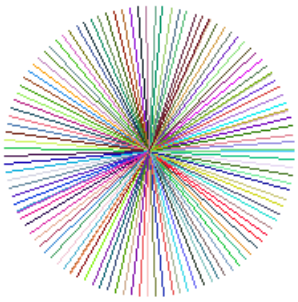
Description

Draw a line of given dimensions (from point x1,y1 to x2,y2) on the current output. The current output is set with StartDrawing() .

If the 'Color' parameter is not specified, the default color set with FrontColor() will be used. RGB() or RGBA() can be used to get a valid color value.

Example:

```
If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 200, 200, RGB(255, 255, 255))
    For Angle = 0 To 360 Step 3
      LineXY(100, 100, 100+Cos(Angle)*90, 100+Sin(Angle)*90,
 RGB(Random(255), Random(255), Random(255)))
    Next Angle
    StopDrawing()
    ImageGadget(0, 0, 0, 200, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.21 Plot

Syntax

```
Plot(x, y [, Color])
```

Description

Draw a plot at the given x,y position in the current output. The current output is set with StartDrawing() . If the 'Color' parameter is not specified, the default color set with FrontColor() will be

used. RGB() or RGBA() can be used to get a valid color value.

The given x, y coordinate must be inside the current drawing area, as there is no bounds checking for speed reasons.

Example:

```
If OpenWindow(0, 0, 0, 200, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
    For x = 0 To 199
      For y = 0 To 199
        Plot(x, y, RGB(Random(255), Random(255), Random(255)))
      Next y
    Next x
    StopDrawing()
    ImageGadget(0, 0, 0, 200, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```

65.22 Point

Syntax

```
Color = Point(x, y)
```

Description

Return the color used at the x,y position in the current output. The color can be split in their RGB and alpha values by using the Red() , Green() , Blue() and Alpha() functions. The given x, y coordinate must be inside the current drawing area, as there is no bounds checking for speed reasons.

The returned color will only contain alpha information if the output has a 32bit color depth and the current DrawingMode() is set to one of the alpha channel modes. Otherwise the alpha component of the color is set to 0.

Note: This command does not work with PrinterOutput() .

65.23 StopDrawing

Syntax

```
StopDrawing()
```

Description

Once all the needed graphics operations (started with StartDrawing()) have been performed, this function must be called to finish the drawing and free all associated resources.

Typically, a normal drawing sequence would look like:

```
If StartDrawing(WindowOutput(0))
  Box(10,10,20,20)
  Line(30,50,100,100)
  ....
  StopDrawing()
EndIf
```

65.24 TextHeight

Syntax

```
Height = TextHeight(Text$)
```

Description

Return the height (in pixels) of the given string in the current output using the current font.

65.25 TextWidth

Syntax

```
Width = TextWidth(Text$)
```

Description

Return the width (in pixels) of the given string in the current output using the current font. This function is handy for getting the real length of a string when using a non-proportional font.

65.26 OutputDepth

Syntax

```
Result = OutputDepth()
```

Description

Returns the color depth (in bits per pixel) of the current drawing output.

65.27 OutputWidth

Syntax

```
Result = OutputWidth()
```

Description

Returns the width (in pixels) of the current drawing output.

65.28 OutputHeight

Syntax

```
Result = OutputHeight()
```

Description

Returns the height (in pixels) of the current drawing output.

65.29 CustomFilterCallback

Syntax

```
CustomFilterCallback(@FilterCallback())
```

Description

Specify a callback that will be called for every pixel that is part of a drawing operation in #PB_2DDrawing_CustomFilter drawing mode .

The callback must have the following form:

```
Procedure CustomCallback(x, y, SourceColor, TargetColor)
  ;
  ; Calculate ResultColor from the given input
  ;
  ProcedureReturn ResultColor
EndProcedure
```

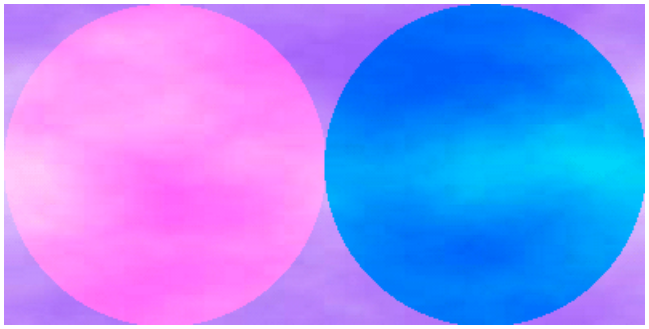
The callback will be called for every pixel that is drawn as a result of a call to drawing functions like Line() , Box() or DrawText() . The SourceColor parameter specifies the color given in the drawing operation and the TargetColor parameter specifies the color of the target pixel in the drawing area. Both colors are always 32bit with alpha channel independent of the color depth of the output. The callback has to calculate the color that the target pixel should have after the drawing and return that.

This callback will be called many times (for every pixel to draw) so it should be very small and fast to not have a too big impact on the drawing performance.

Note: The #PB_2DDrawing_CustomFilter drawing mode only works on ImageOutput() .

Example:

```
Procedure FilterCallback(x, y, SourceColor, TargetColor)
  ; Take only the Red component from the Source, do not modify the others
  ProcedureReturn RGBA(Red(SourceColor), Green(TargetColor), Blue(TargetColor), Alpha(TargetColor))
EndProcedure

UseJPEGImageDecoder()

If OpenWindow(0, 0, 0, 400, 200, "2DDrawing Example", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  LoadImage(1, #PB_Compiler_Home + "examples/sources/data/clouds.jpg")

  If CreateImage(0, 400, 200) And StartDrawing(ImageOutput(0))
    DrawImage(ImageID(1), 0, 0, 400, 200)

    DrawingMode(#PB_2DDrawing_CustomFilter)
    CustomFilterCallback(@FilterCallback())
    Circle(100, 100, 100, $0000FF)
    Circle(300, 100, 100, $000000)

    StopDrawing()
    ImageGadget(0, 0, 0, 400, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.30 GradientColor

Syntax

```
GradientColor(Position.f, Color)
```

Description

Adds the given Color at the given Position to the spectrum of the drawing gradient. Position must be a float value between 0.0 and 1.0.

By default, the drawing gradient ranges from the current background color at position 0.0 to the current front color at position 1.0. With this command, additional colors can be added in-between, or the colors at 0.0 and 1.0 can be overwritten.

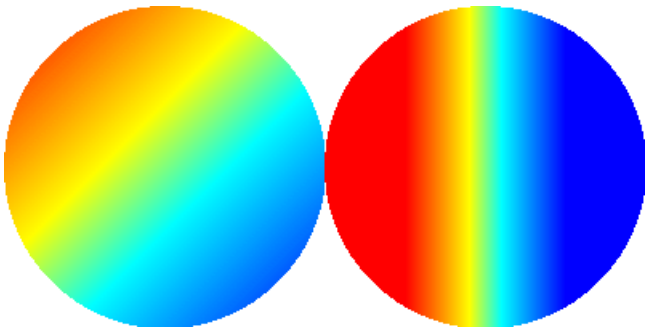
The ResetGradientColors() command can be used to revert back to the default gradient.

The following commands can be used to specify the shape of the drawing gradient:
- LinearGradient()
- CircularGradient()
- EllipticalGradient()
- BoxedGradient()
- ConicalGradient()
- CustomGradient()

Note: This command only has an effect with ImageOutput() . The gradient only affects the drawing commands when #PB_2DDrawing_Gradient is set with the DrawingMode() command.

Example:

```
If OpenWindow(0, 0, 0, 400, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 400, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 400, 200, $FFFFFF)

    DrawingMode(#PB_2DDrawing_Gradient)
    BackColor($0000FF)
    GradientColor(0.4, $00FFFF)
    GradientColor(0.6, $FFFF00)
    FrontColor($FF0000)

    LinearGradient(0, 0, 200, 200)
    Circle(100, 100, 100)
    LinearGradient(350, 100, 250, 100)
    Circle(300, 100, 100)

    StopDrawing()
    ImageGadget(0, 0, 0, 400, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.31 ResetGradientColors

Syntax

```
ResetGradientColors()
```

Description

Removes all colors from the drawing gradient and reverts back to a gradient from the current background color to the current front color .

The GradientColor() command can be used to add additional colors to the gradient.

Note: This command only has an effect with ImageOutput() . The gradient only affects the drawing commands when #PB_2DDrawing_Gradient is set with the DrawingMode() command.

65.32 LinearGradient

Syntax

```
LinearGradient(x1, y1, x2, y2)
```

Description

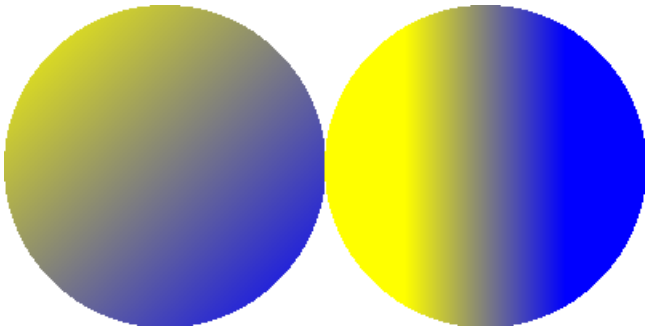
Sets the drawing gradient to have a linear shape defined by the two points x1,y1 and x2,y2.

By default the gradient ranges from the current background color at x1,y1 to the current front color at x2, y2. Additional colors can be added to the gradient with the GradientColor() command.

Note: This command only has an effect with ImageOutput() . The gradient only affects the drawing commands when #PB_2DDrawing_Gradient is set with the DrawingMode() command.

Example:

```
If OpenWindow(0, 0, 0, 400, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 400, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 400, 200, $FFFFFF)

    DrawingMode(#PB_2DDrawing_Gradient)
    BackColor($00FFFF)
    FrontColor($FF0000)

    LinearGradient(0, 0, 200, 200)
    Circle(100, 100, 100)
    LinearGradient(350, 100, 250, 100)
    Circle(300, 100, 100)

    StopDrawing()
    ImageGadget(0, 0, 0, 400, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.33 CircularGradient

Syntax

```
CircularGradient(x, y, Radius)
```

Description

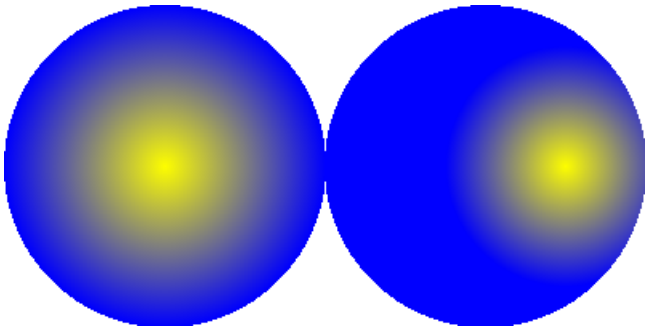
Sets the drawing gradient to have a circular shape around the x,y point with the given radius.

By default the gradient ranges from the current background color at x,y to the current front color at the distance given by Radius away from the x,y point. Additional colors can be added to the gradient with the GradientColor() command.

Note: This command only has an effect with ImageOutput() . The gradient only affects the drawing commands when #PB_2DDrawing_Gradient is set with the DrawingMode() command.

Example:

```
If OpenWindow(0, 0, 0, 400, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 400, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 400, 200, $FFFFFF)

    DrawingMode(#PB_2DDrawing_Gradient)
    BackColor($00FFFF)
    FrontColor($FF0000)

    CircularGradient(100, 100, 100)
    Circle(100, 100, 100)
    CircularGradient(350, 100, 75)
    Circle(300, 100, 100)

    StopDrawing()
    ImageGadget(0, 0, 0, 400, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.34 EllipticalGradient

Syntax

```
EllipticalGradient(x, y, RadiusX, RadiusY)
```

Description

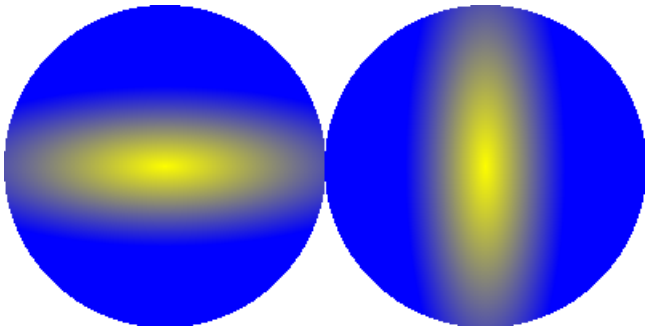
Sets the drawing gradient to have an elliptical shape around the x,y point with the given RadiusX and RadiusY.

By default the gradient ranges from the current background color at x,y to the current front color at the ellipse defined by RadiusX and RadiusY. Additional colors can be added to the gradient with the GradientColor() command.

Note: This command only has an effect with ImageOutput() . The gradient only affects the drawing commands when #PB_2DDrawing_Gradient is set with the DrawingMode() command.

Example:

```
If OpenWindow(0, 0, 0, 400, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 400, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 400, 200, $FFFFFF)

    DrawingMode(#PB_2DDrawing_Gradient)
    BackColor($00FFFF)
    FrontColor($FF0000)

    EllipticalGradient(100, 100, 150, 50)
    Circle(100, 100, 100)
    EllipticalGradient(300, 100, 50, 150)
    Circle(300, 100, 100)

    StopDrawing()
    ImageGadget(0, 0, 0, 400, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.35 BoxedGradient

Syntax

```
BoxedGradient(x, y, Width, Height)
```

Description

Sets the drawing gradient to have a box shape defined by the given parameters.

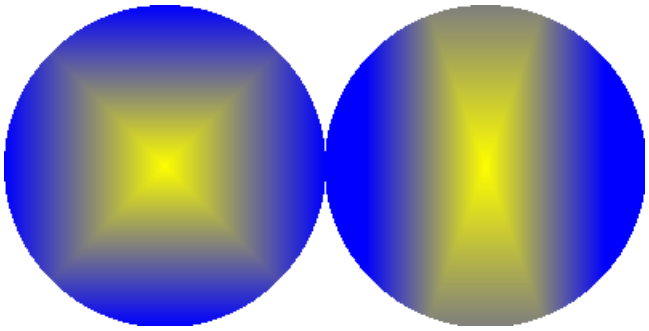
By default the gradient ranges from the current background color at the center of the box to the current front color at the edges of the box. Additional colors can be added to the gradient with the GradientColor() command.

Note: This command only has an effect with ImageOutput() . The gradient only affects the drawing commands when #PB_2DDrawing_Gradient is set with the DrawingMode() command.

Example:

```
If OpenWindow(0, 0, 0, 400, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 400, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 400, 200, $FFFFFF)

    DrawingMode(#PB_2DDrawing_Gradient)
    BackColor($00FFFF)
    FrontColor($FF0000)

    BoxedGradient(0, 0, 200, 200)
    Circle(100, 100, 100)
    BoxedGradient(225, -100, 150, 400)
    Circle(300, 100, 100)

    StopDrawing()
    ImageGadget(0, 0, 0, 400, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```

@newline



65.36 ConicalGradient

Syntax

```
ConicalGradient(x, y, Angle.f)
```

Description

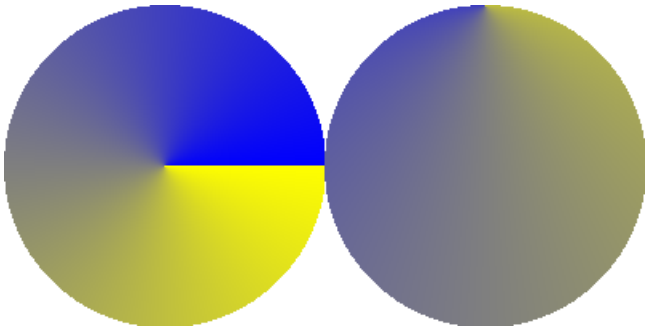
Sets the drawing gradient to have a conical shape around the x,y point starting at the given Angle (in degrees).

By default the gradient starts with the current background color at the given angle and then changes towards the current front color counter-clockwise until the angle is reached again. Additional colors can be added to the gradient with the GradientColor() command.

Note: This command only has an effect with ImageOutput() . The gradient only affects the drawing commands when #PB_2DDrawing_Gradient is set with the DrawingMode() command.

Example:

```
If OpenWindow(0, 0, 0, 400, 200, "2DDrawing Example",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateImage(0, 400, 200) And StartDrawing(ImageOutput(0))
    Box(0, 0, 400, 200, $FFFFFF)

    DrawingMode(#PB_2DDrawing_Gradient)
    BackColor($00FFFF)
    FrontColor($FF0000)

    ConicalGradient(100, 100, 0.0)
    Circle(100, 100, 100)
    ConicalGradient(300, 0, 90.0)
    Circle(300, 100, 100)

    StopDrawing()
    ImageGadget(0, 0, 0, 400, 200, ImageID(0))
  EndIf

  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```



65.37 CustomGradient

Syntax

```
CustomGradient(@GradientCallback())
```

Description

Sets the drawing gradient to have a custom shape, defined by the given callback procedure.

The callback must have the following form:

```
Procedure.f GradientCallback(x, y)
  ;
  ; Return a value between 0.0 and 1.0 to define the gradient for the x/y position.
  ;
  ProcedureReturn 1.0
EndProcedure
```

The callback will be called for every pixel that is part of a drawing operation. The callback has to return a value between 0.0 and 1.0 (not a color value) to define the gradient value at the given position.

By default the value 0.0 represents the current background color and the value 1.0 represents the current front color . Additional colors can be added to the gradient with the GradientColor() command.

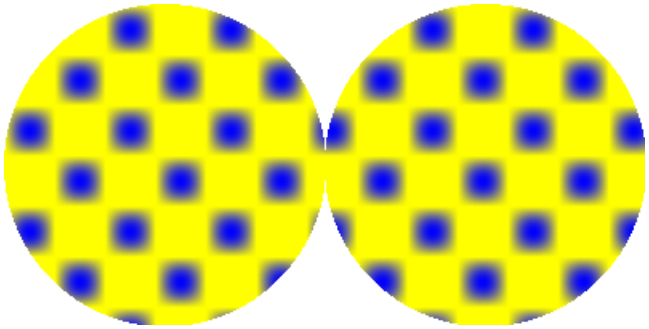
This callback will be called many times (for every pixel to draw) so it should be very small and fast to not have a too big impact on the drawing performance.

Note: This command only has an effect with ImageOutput() . The gradient only affects the drawing commands when #PB_2DDrawing_Gradient is set with the DrawingMode() command.

Example:

```
Procedure.f GradientCallback(x, y)
  ProcedureReturn Sin(x * 0.1) * Sin(y * 0.1) ; will be between 0 and 1
EndProcedure

If OpenWindow(0, 0, 0, 400, 200, "2DDrawing Example", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
```

```
    If CreateImage(0, 400, 200) And StartDrawing(ImageOutput(0))
      Box(0, 0, 400, 200, $FFFFFF)

      DrawingMode(#PB_2DDrawing_Gradient)
      BackColor($00FFFF)
      FrontColor($FF0000)

      CustomGradient(@GradientCallback())
      Circle(100, 100, 100)
      Circle(300, 100, 100)

      StopDrawing()
      ImageGadget(0, 0, 0, 400, 200, ImageID(0))
    EndIf

    Repeat
      Event = WaitWindowEvent()
    Until Event = #PB_Event_CloseWindow
  EndIf
```



Chapter 66

Array

Arrays are structures to store indexed elements. Unlike a list or a map , the elements are allocated in a contiguous manner in memory. Therefore, it's not possible to easily insert or remove an element. On other hand, it provides a very fast direct access to a random element.

To work with Arrays, they have to be declared first. This could be done with the keyword Dim .

66.1 ArraySize

Syntax

```
Result = ArraySize(Array() [, Dimension])
```

Description

Returns the size of the array, as specified with Dim or ReDim. For multi-dimensional arrays, 'Dimension' can be specified to get a specific dimension size. The first dimension starts from 1. If the array isn't yet declared (or its allocation has failed), it will return -1.

Example:

```
Dim MyArray.l(10)
Debug ArraySize(MyArray()) ; will print '10'

Dim MultiArray.l(10, 20, 30)
Debug ArraySize(MultiArray(), 2) ; will print '20'
```

Example:

```
Dim Test.q(99999999999999999)

If ArraySize(Test()) <> -1
  Test(12345) = 123  ; everything fine
Else
```

```
    Debug "Array 'Test()' couldn't be initialized."
  EndIf
```

66.2 CopyArray

Syntax

```
Result = CopyArray(SourceArray(), DestinationArray())
```

Description

Copy every elements of 'SourceArray()' into 'DestinationArray()'. If the two arrays are not of the same type (native or structured) the copy will fail and 'Result' will be 0. Every elements previously found in 'DestinationArray()' will be deleted. After a successful copy, the two arrays are identicals.

Example:

```
Dim Numbers(5)
Dim NumbersCopy(10)

Numbers(0) = 128
Numbers(5) = 256

Debug "Array size before copy: "+Str(ArraySize(NumbersCopy())) ; will print 10

CopyArray(Numbers(), NumbersCopy())

Debug "Array size after copy: "+Str(ArraySize(NumbersCopy())) ; will print 5
Debug NumbersCopy(0)
Debug NumbersCopy(5)
```

66.3 FreeArray

Syntax

```
Result = FreeArray(Array())
```

Description

Free the specified 'Array()' and release all its associated memory. To use it again Dim has to be called.

Chapter 67

AudioCD

Audio CDs are an excellent way to play very high quality music during a game or an application, without consuming a lot of system resources. This library provides all the functions necessary to manage audio CD playback within PureBasic programs.

67.1 AudioCDLength

Syntax

```
Result = AudioCDLength()
```

Description

Returns the total length of an entire audio CD in seconds.

Supported OS

Windows, MacOS X

67.2 AudioCDName

Syntax

```
Result = AudioCDName()
```

Description

Returns the system-dependent name associated with the current audio CD drive. For example, it would return 'D:\' on a Windows-based system, if the audio CD is associated as drive letter 'D'.

67.3 AudioCDTrackLength

Syntax

```
Result = AudioCDTrackLength(TrackNumber)
```

Description

Returns the length of the specified track in seconds.

67.4 AudioCDStatus

Syntax

```
Result = AudioCDStatus()
```

Description

Returns the actual state of the current audio CD drive. The 'Return' value may take the following values:

-1: CD drive not ready (it is either empty or opened)
0: CD drive stopped (but the CD is in the drive and has been detected)
>0: Audio track which is currently being played.

67.5 AudioCDTracks

Syntax

```
Result = AudioCDTracks()
```

Description

Return the total number of tracks on the CD available to be played.

67.6 AudioCDTrackSeconds

Syntax

```
Result = AudioCDTrackSeconds()
```

Description

Return the number of seconds elapsed since the current track began to play.

67.7 EjectAudioCD

Syntax

```
EjectAudioCD(State)
```

Description

Eject (open) or close the current CD drive. If State = 1, the CD drive is opened, else it is closed.

Supported OS

Windows, Linux

67.8 InitAudioCD

Syntax

```
Result = InitAudioCD()
```

Description

Try to initialize all the necessary resources in order to handle audio CD functions. This function must be called before any other audio CD function. If Result is 0, the initialization has failed, or there are no CD drives available, else Result contains the number of CD drives attached to the system.

67.9 PauseAudioCD

Syntax

```
PauseAudioCD()
```

Description

Pause audio CD playback. The playback may be resumed by using the ResumeAudioCD() function.

67.10 PlayAudioCD

Syntax

```
PlayAudioCD(StartTrack, EndTrack)
```

Description

Start to play the audio CD from 'StartTrack' until the end of 'EndTrack'. The first track is indexed to 1.

67.11 ResumeAudioCD

Syntax

```
ResumeAudioCD()
```

Description

Resume audio CD playback, previously paused by the use of the PauseAudioCD() function.

67.12 StopAudioCD

Syntax

```
StopAudioCD()
```

Description

Stop audio CD playback.

67.13 UseAudioCD

Syntax

```
UseAudioCD(AudioCDDrive)
```

Description

Just as the InitAudioCD() function is used to determine the number of audio CD drives which are available on the system, the UseAudioCD() function is used in order to change the current audio CD drive. So with PureBasic, it is possible to play several CDs at the same time.

Chapter 68

Billboard

Billboards are 3D planes (composed of 2 triangles) which always face the camera . Billboards are useful to quickly render a high number of elements , such as rain, snow, trees, vegetation or any particle-based effects. InitEngine3D() should be called successfully before using any billboard functions.

Billboards are added to billboard groups, which are collections of similarly sized and positioned billboards, so that they may be displayed.

68.1 AddBillboard

Syntax

```
AddBillboard(#Billboard, #BillboardGroup, x, y, z)
```

Description

Adds a billboard to the specified billboard group at the given position, relative to the position of the billboard group. The billboard group must have previously been created using the CreateBillboardGroup() function.

#Billboard - The unique numeric value used to identify this billboard.

#BillboardGroup - The number of the billboard group you want to attach the new billboard to.

x - X co-ordinate to create billboard at, relative to the X co-ordinate of the billboard group.

y - Y co-ordinate to create billboard at, relative to the Y co-ordinate of the billboard group.

z - Z co-ordinate to create billboard at, relative to the Z co-ordinate of the billboard group.

This function does not return any value.

68.2 BillboardGroupID

Syntax

```
BillboardGroupID = BillboardGroupID(#BillboardGroup)
```

Description

Returns the unique ID which identifies the given '#BillboardGroup' in the operating system. This function is very useful when another library needs a billboard-group reference.

68.3 BillboardGroupLocate

Syntax

```
BillboardGroupLocate(#BillboardGroup, x, y, z)
```

Description

Translates (moves) the billboard group to a new absolute position in the world. This means that after this function has been performed, the billboard group will be at the position specified by the parameters passed to this function. Use the MoveBillboardGroup() function to move a billboard group using relative co-ordinates.

#BillboardGroup - The number of the billboard group you want to move.

x - New X co-ordinate to move to.

y - New Y co-ordinate to move to.

z - New Z co-ordinate to move to.

This function does not return any value.

68.4 BillboardGroupMaterial

Syntax

```
BillboardGroupMaterial(#BillboardGroup, MaterialID)
```

Description

Assigns a material to the specified billboard group. This material will be used by all the billboards added to this group. A group can only have one material assigned at any point in time.

#BillboardGroup - The number of the billboard group you wish to assign a new material to.

MaterialID - The ID number of the material to assign to the billboard group (and billboards). A valid 'MaterialID' can be easily obtained by using the MaterialID() function.

This function does not return any value.

68.5 BillboardGroupX

Syntax

```
Result = BillboardGroupX(#BillboardGroup)
```

Description

Determines the absolute position of the billboard group in the world.

#BillboardGroup - The number of the billboard group which you want to know the position of.

Result - The absolute X position of the billboard group.

68.6 BillboardGroupY

Syntax

```
Result = BillboardGroupY(#BillboardGroup)
```

Description

Determines the absolute position of the billboard group in the world.

#BillboardGroup - The number of the billboard group which you want to know the position of.

Result - The absolute Y position of the billboard group.

68.7 BillboardGroupZ

Syntax

```
Result = BillboardGroupZ(#BillboardGroup)
```

Description

Determines the absolute position of the billboard group in the world.

#BillboardGroup - The number of the billboard group which you want to know the position of.

Result - The absolute Z position of the billboard group.

68.8 BillboardHeight

Syntax

```
Result = BillboardHeight(#Billboard, #BillboardGroup)
```

Description

Determines the height of a billboard which has been added to a billboard group. The height is measured in the units used by the world.

#Billboard - The number of the billboard which you want to know the height of.

#BillboardGroup - The number of the billboard group which contains the billboard.

Result - The height of the billboard.

68.9 BillboardLocate

Syntax

```
BillboardLocate(#Billboard, #BillboardGroup, x, y, z)
```

Description

Moves a billboard to a new location within the billboard group which is it added to. The position to move to is relative to the position of the billboard group. Use the MoveBillboard() function to move a billboard relative to its own co-ordinates.

#Billboard - The number of the billboard which you want to move.

#BillboardGroup - The number of the billboard group which contains the billboard.

x - The new X position for the billboard.

y - The new Y position for the billboard.

z - The new Z position for the billboard.

This function does not return any value.

68.10 BillboardWidth

Syntax

```
Result = BillboardWidth(#Billboard, #BillboardGroup)
```

Description

Determines the width of a billboard which has been added to a billboard group. The width is measured in the units used by the world.

#Billboard - The number of the billboard which you want to know the width of.

#BillboardGroup - The number of the billboard group which contains the billboard.

Result - The width of the billboard.

68.11 BillboardX

Syntax

```
Result = BillboardX(#Billboard, #BillboardGroup)
```

Description

Determines the position of the billboard, relative to the position of the group which it is in.

#Billboard - The number of the billboard which you want to know the position of.

#BillboardGroup - The number of the billboard group which contains the billboard.

Result - The X position of the billboard.

68.12 BillboardY

Syntax

```
Result = BillboardY(#Billboard, #BillboardGroup)
```

Description

Determines the position of the billboard, relative to the position of the group which it is in.

#Billboard - The number of the billboard which you want to know the position of.

#BillboardGroup - The number of the billboard group which contains the billboard.

Result - The Y position of the billboard.

68.13 BillboardZ

Syntax

```
Result = BillboardZ(#Billboard, #BillboardGroup)
```

Description

Allows you to find out the position of the billboard, relative to the position of the group which it is in.

#Billboard - The number of the billboard which you want to know the position of.

#BillboardGroup - The number of the billboard group which contains the billboard.

Result - The Z position of the billboard.

68.14 ClearBillboards

Syntax

```
ClearBillboards(#BillboardGroup)
```

Description

Removes and destroys all billboards in the specified billboard group.

#BillboardGroup - The number of the billboard group you want to clear.

This function does not return any value.

68.15 CountBillboards

Syntax

```
Result = CountBillboards(#BillboardGroup)
```

Description

Counts the number of billboards contained in a billboard group.

#BillboardGroup - The number of the billboard group you want to count the contained billboards.

Result - The number of billboards which have been added to the billboard group.

68.16 CreateBillboardGroup

Syntax

```
Result = CreateBillboardGroup(#BillboardGroup, MaterialID,
   DefaultBillboardWidth, DefaultBillboardHeight [, x, y, z])
```

Description

Creates a new empty billboard group. If #PB_Any is used as '#BillboardGroup' parameter, the new billboard group number will be returned to Result.

The specified material will be used by all the billboards that are added to the group. A valid 'MaterialID' can be easily obtained by using the MaterialID() function. The default size of future billboards must be specified with the 'DefaultBillboardWidth' and 'DefaultBillboardHeight' parameters. Although the size of each billboard can be set separately with the ResizeBillboard() function, it is important to keep in mind that it will have a negative impact on performance if all billboards are not the same size. The optional x,y and z parameters can be specified to set the position of the newly-created group in the world. If #BillboardGroup was already created, it will be automatically freed (along with its associated billboards) and replaced by the new one.

#BillboardGroup - The number which you want to use to refer to the newly-created billboard group in calls to the other functions in this library.

MaterialID - The ID number of the material to use for all billboards added to this group, using the AddBillboard() function. The material ID can be easily determined by using the MaterialID() function.

DefaultBillboardWidth - The default width (in world units) of billboards added to this group.

DefaultBillboardHeight - The default height (in world units) of billboards added to this group.

The following three parameters are optional, but all must be specified together.

x - The absolute X position of the new billboard group in the world.

y - The absolute Y position of the new billboard group in the world.

z - The absolute Z position of the new billboard group in the world.

Result - The value returned by this function can be used to determing whether the billboard group could be created or not. If the value is zero, then the billboard group could not be created. If the value is non-zero then it was created successfully.

68.17 FreeBillboardGroup

Syntax

```
FreeBillboardGroup(#BillboardGroup)
```

Description

Free the specified billboard group and any billboards contained within it. All its associated memory is released and this object cannot be used anymore.

Note: all remaining billboard groups are automatically freed when the program ends.

#BillboardGroup - The number of the billboard group you want to free.

This function does not return any value.

68.18 HideBillboardGroup

Syntax

```
HideBillboardGroup(#BillboardGroup, State)
```

Description

Changes the visibility of (hides or shows) a billboard group and any billboards it contains.

#BillboardGroup - The number of the billboard group you want to hide or show.

State - The new state of the billboard group visibility. State can take the following values:

```
  0: the billboard group is hidden
  1: the billboard group is shown
```

This function does not return any value.

68.19 IsBillboardGroup

Syntax

```
Result = IsBillboardGroup(#BillboardGroup)
```

Description

Tests if the given #BillboardGroup is a valid and correctly-initialized billboard group.

This function is bullet-proof and can be used with any value. If Result is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that a billboard group is ready to use.

68.20 MoveBillboard

Syntax

```
MoveBillboard(#Billboard, #BillboardGroup, x, y, z)
```

Description

Moves a billboard which is contained in a billboard group by the specified x, y and z values. This is a relative move based on the current location of the billboard. To perform an absolute move (actually, relative to the co-ordinates of the billboard group) use BillboardLocate() .

#Billboard - The number of the billboard which you want to move.

#BillboardGroup - The number of the billboard group which contains the billboard.

x - The offset to add to the current X position of the billboard.

y - The offset to add to the current Y position of the billboard.

z - The offset to add to the current Z position of the billboard.

This function does not return any value.

68.21 MoveBillboardGroup

Syntax

```
MoveBillboardGroup(#BillboardGroup, x, y, z)
```

Description

Moves a billboard group by the specified x, y and z values. This is a relative move based on the current location of the billboard group. To perform an absolute move to new positions within the world, use BillboardGroupLocate() .

#BillboardGroup - The number of the billboard group you want to move.

x - The offset to add to the current X position of the billboard group.

y - The offset to add to the current Y position of the billboard group.

z - The offset to add to the current Z position of the billboard group.

This function does not return any value.

68.22 RemoveBillboard

Syntax

```
RemoveBillboard(#Billboard, #BillboardGroup)
```

Description

Removes a billboard from the specified billboard group.

#Billboard - The number of the billboard which you want to remove.

#BillboardGroup - The number of the billboard group which currently contains the billboard.

This function does not return any value.

68.23 ResizeBillboard

Syntax

```
ResizeBillboard(#Billboard, #BillboardGroup, Width, Height)
```

Description

Resizes a billboard, which is currently contained in a billboard group, to a new width and height, specified in world units. Note that although you can resize billboards independently with this function there will be some performance lost if the billboards within a billboard group are all different sizes.

#Billboard - The number of the billboard which you want to resize.

#BillboardGroup - The number of the billboard group which currently contains the billboard.

Width - The new width of the billboard in world units.

Height - The new height of the billboard in world units.

This function does not return any value.

68.24 RotateBillboardGroup

Syntax

```
RotateBillboardGroup(#BillboardGroup, x, y, z [, Mode])
```

Description

Rotates the #BillboardGroup according to the specified x, y, z angle values. All angles are in degree, with values ranging from 0 to 360. The optional 'Mode' can be one of the following value:

```
PB_Absolute: absolute rotation (default).
PB_Relative: relative rotation based on the previous node rotation.
```

Chapter 69

Camera

Cameras are used to display the 3D world. You can manage them like real life cameras, which means you can rotate, move, change the field of vision and more. At least one camera is absolutely required in order to render the world to the screen. Many cameras can be used at the same time with different positions and views to allow cool effects like: split-screen, rear view, etc. InitEngine3D() must be called successfully before using the camera functions.

69.1 CameraBackColor

Syntax

```
CameraBackColor(#Camera, Color)
```

Description

Changes the #Camera background color. When a new camera is created, there is no background color to allow to mix 2D background and the 3D scene.

#Camera - The number of the camera you want to change the background color of. This is the value which you used when you created the camera using CreateCamera()

Color - New color of the camera background. RGB() can be used to get a valid color value.

This function does not return any value.

69.2 CameraFOV

Syntax

```
CameraFOV(#Camera, Angle)
```

Description

Changes a camera field Of vision (FOV) which allows you to view a larger or smaller area of the scene. Angles above 90 degrees result in a wide-angle (fish-eye like) view. Angles lower than 30 degrees result in a stretched (telescopic) view. Typical values are between 45 and 60 degrees.

#Camera - The number of the camera you want to change the FOV of. This is the value which you used when you created the camera using CreateCamera()

Angle - the new view angle for the camera, in degrees.

This function does not return any value.

69.3 CameraID

Syntax

```
CameraID = CameraID(#Camera)
```

Description

Returns the unique ID which identifies the given '#Camera' in the operating system. This function is very useful when another library needs a camera reference.

69.4 CameraLocate

Syntax

```
CameraLocate(#Camera, x, y, z)
```

Description

Changes the position of a camera to a new absolute location in the world. Only the position is changed, all other values stay as they are (such as direction of camera, field of view, etc). If you want to perform a movement relative to the current camera location you can use the MoveCamera() function.

#Camera - The number of the camera you want to change the position of. This is the value which you used when you created the camera using CreateCamera()

x - The new X position for the camera.

y - The new Y position for the camera.

z - The new Z position for the camera.

This function does not return any value.

69.5 CameraLookAt

Syntax

```
CameraLookAt(#Camera, x, y, z)
```

Description

Changes the direction that a camera is facing so that is pointing towards the specified point. The position of the camera is not changed.

#Camera - The number of the camera you want to change the direction of. This is the value which you used when you created the camera using CreateCamera()

x - The X position to point the camera at.

y - The Y position to point the camera at.

z - The Z position to point the camera at.

This function does not return any value.

69.6 CameraProjectionMode

Syntax

```
CameraProjectionMode(#Camera, Mode)
```

Description

Change the #Camera projection mode.

#Camera - The number of the camera you want to change the projection mode of. This is the value which you used when you created the camera using CreateCamera()

Mode - the value representing the way in which the world should be projected. This should be one of the following values:

```
#PB_Camera_Perspective : Renders the whole scene with perspective
#PB_Camera_Orthographic: Renders the whole scene in Orthographic mode
 (no 3D depth)
```

This function does not return any value.

69.7 CameraProjectionX

Syntax

```
Result = CameraProjectionX(#Camera, x, y, z)
```

Description

Returns the 'x' position, in pixels, of a 3D point on the specified #Camera. If the point is outside of the camera current view, it returns -1. This is very useful to map 3D points to 2D screen.

69.8 CameraProjectionY

Syntax

```
Result = CameraProjectionY(#Camera, x, y, z)
```

Description

Returns the 'y' position, in pixels, of a 3D point on the specified #Camera. If the point is outside of the camera current view, it returns -1. This is very useful to map 3D points to 2D screen.

69.9 CameraRange

Syntax

```
CameraRange(#Camera, Near, Far)
```

Description

Changes camera near and far range.

#Camera - The number of the camera you want to change the range of. This is the value which you used when you created the camera using CreateCamera()

Near - The closest distance to the camera where the world should be rendered.

Far - The largest distance to the camera where the world should be rendered.

This function does not return any value.

69.10 CameraRenderMode

Syntax

```
CameraRenderMode(#Camera, RenderMode)
```

Description

Changes the mode in which the world is displayed through a camera. When you create a new camera, using the CreateCamera() function, the default render mode is to display the world with full detail and textures.

#Camera - The number of the camera you want to change the render mode of. This is the value which you used when you created the camera using CreateCamera()

RenderMode - the value representing the way in which the world should be rendered. This should be one of the following values:

```
#PB_Camera_Plot      : Renders the whole scene by showing only the
 vertices (plots)
#PB_Camera_Wireframe: Renders the whole scene by showing only the
 triangles (lines)
#PB_Camera_Textured : Renders the whole scene at full detail, with
 textures.
```

This function does not return any value.

69.11 CameraX

Syntax

```
Result = CameraX(#Camera)
```

Description

Used to find out the current position of a camera in the world.

#Camera - The number of the camera you want to find the position of. This is the value which you used when you created the camera using CreateCamera()

Result - the X position of the specified camera.

69.12 CameraY

Syntax

```
Result = CameraY(#Camera)
```

Description

Used to find out the current position of a camera in the world.

#Camera - The number of the camera you want to find the position of. This is the value which you used when you created the camera using CreateCamera()

Result - the Y position of the specified camera.

69.13 CameraZ

Syntax

```
Result = CameraZ(#Camera)
```

Description

Used to find out the current position of a camera in the world.

#Camera - The number of the camera you want to find the position of. This is the value which you used when you created the camera using CreateCamera()

Result - the Z position of the specified camera.

69.14 CreateCamera

Syntax

```
Result = CreateCamera(#Camera, x, y, Width, Height)
```

Description

Creates a new #Camera in the current world, at the position x,y with the specified dimensions. Note that these positions and sizes are the position and sizes of the display on your screen, not the position and size of the camera in the world.

Unlike other graphics related functions, these co-ordinates and dimensions are in percentages, from 0 to 100 (float numbers can be used for more precise placement). This could seems odd, but 3D applications should be resolution independant and this is a good way of achieving that. If a camera is created with a height of 50% then it will always fill 50% of the height of the screen, irrespective of whether you use a screen which is 640*480 or 1600*1200. If a camera has already been created with the same number, the previous camera is automatically freed and replaced by the new one.

Cameras can be overlapped. This means that it is possible to display a camera which fills the whole screen and put a little camera inside which shows another place in the world. To achieve this, the order of creation is very important: the camera which will be on top should be created last.

#Camera - The numeric value you want to use to identify the new camera when you use the other functions from this library.

x - The percentage position across the screen of the left edge of the start of the display for this camera. 0% means at the left of the screen, 100% means at the right of the screen.

y - The percentage position down the screen of the top edge of the start of the display for this camera. 0% means at the top of the screen, 100% means at the bottom of the screen.

Width - The percentage of the width of the screen that the display for this camera should take up.

Height - The percentage of the height of the screen that the display for this camera should take up.

Returns '0' if the #Camera can't be created. If #PB_Any is used as '#Camera' parameter, the new camera number will be returned as 'Result'.

Example:

```
CreateCamera(0, 0, 0, 100, 100) ; Creates a camera which fill the whole screen

CreateCamera(0, 0,  0, 100, 50) ; Creates an horizontal split screen effect, for 2 players
CreateCamera(1, 0, 50, 100, 50) ; mode on the same screen

CreateCamera(0, 0, 0, 100, 100) ; Create a full screen camera
CreateCamera(1, 25, 0, 50, 10)  ; And a rear-view mirror effect.
                                ; Note the rear-view is last so that it is displayed on top of the full screen camera
```

69.15 FreeCamera

Syntax

```
FreeCamera(#Camera)
```

Description

Frees a camera and releases all its associated memory. This camera must not be used (by using its number with the other functions in this library) after calling this function, unless you create it again.

Note: all remaining cameras are automatically freed when the program ends.

#Camera - The number of the camera you want to free. This is the value which you used when you created the camera using CreateCamera()

This function does not return any value.

69.16 IsCamera

Syntax

```
Result = IsCamera(#Camera)
```

Description

Tests if the given #Camera is a valid and correctly initialized camera.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that a camera is ready to use.

69.17 MoveCamera

Syntax

```
MoveCamera(#Camera, x, y, z)
```

Description

Moves a camera in the world according to the specified x,y,z values. This is a relative move based on the current camera location and direction. To perform an absolute move in the world, use CameraLocate() .

#Camera - The number of the camera you want to move. This is the value which you used when you created the camera using CreateCamera()

x - Offset to add to the current camera X position.

y - Offset to add to the current camera Y position.

z - Offset to add to the current camera Z position.

This function does not return any value.

69.18 RotateCamera

Syntax

```
RotateCamera(#Camera, x, y, z [, Mode])
```

Description

Rotates the #Camera according to the specified x,y,z angle values. All angles are in degree, with values ranging from 0 to 360. The optional 'Mode' can be one of the following value:

```
PB_Absolute: absolute rotation (default).
PB_Relative: relative rotation based on the previous camera rotation.
```

Chapter 70

Cipher

The cipher library is a set of functions useful to cipher or encode data. For example the MD5 (Message Digest 5) is a very popular fingerprint routine, used in many areas due to its strong resistance to attacks: Linux, BSD and Unix uses it as alternative way to store password on the disk.

70.1 AddCipherBuffer

Syntax

```
AddCipherBuffer(#Cipher, *Input, *Output, Size)
```

Description

Add a new '*Input' buffer of the specified 'Size' to the cipher started with StartAESCipher() and copy the new ciphered data into the '*Output' buffer. The '*Output' buffer has to be at least as large than the '*Input' buffer.

70.2 AESEncoder

Syntax

```
Result\$ = AESEncoder(*Input, *Output, Size, *Key, Bits,
   *InitializationVector [, Mode])
```

Description

Encodes the specified '*Input' buffer using the AES algorithm into the '*Output' buffer. 'Bits' specify the size of the key used by the ciphering. Valid values are 128, 192 and 256. The 'InitializationVector' is a random data block, used to initialize the ciphering to avoid breach in decoding (only needed when using the #PB_Cipher_CBC mode). Its size depends of the 'Bits' parameter: l6 bytes for 128 bit encryption, 24 bytes for 196 bit and 32 bytes for 256 bit.

AES is an industry class cipher algorithm and is good balanced between speed and security. Here is the wikipedia introduction about AES: 'In cryptography, the Advanced Encryption Standard (AES) is an

encryption standard adopted by the U.S. government. The standard comprises three block ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection originally published as Rijndael. Each AES cipher has a 128-bit block size, with key sizes of 128, 192 and 256 bit, respectively. The AES ciphers have been analyzed extensively and are now used worldwide.'

PureBasic uses a RFC compliant implementation of AES. More information can be found in the RFC 3602: http://www.ietf.org/rfc/rfc3602.txt.

The optional 'Mode' parameter can be one of the following value:

```
#PB_Cipher_CBC: Default mode of encoding (Cipher Block Chaining).
 Needs an '*InitializationVector'.
                Recommended as more secure than ECB mode.
#PB_Cipher_ECB: Alternative mode (Electronic CodeBook). It doesn't
 uses random value nor chaining
                (each block is ciphered independently) making it less
 strong the CBC.
```

Example: CBC

```
; Should be compiled in ascii mode
;
String$ = "Hello this is a test for AES"

*CipheredString   = AllocateMemory(Len(String$)+1) ; Space for the string and its
*DecipheredString = AllocateMemory(Len(String$)+1) ; null terminating character (ASCII mode)

If AESEncoder(@String$, *CipheredString, Len(String$), ?Key, 128, ?InitializationVector)
  Debug "Ciphered: "+PeekS(*CipheredString)

  AESDecoder(*CipheredString, *DecipheredString, Len(String$), ?Key, 128, ?InitializationVector)
  Debug "Deciphered: "+PeekS(*DecipheredString)
EndIf

DataSection
  Key:
    Data.b $06, $a9, $21, $40, $36, $b8, $a1, $5b, $51, $2e, $03, $d5, $34, $12, $00, $06

  InitializationVector:
    Data.b $3d, $af, $ba, $42, $9d, $9e, $b4, $30, $b4, $22, $da, $80, $2c, $9f, $ac, $41
EndDataSection
```

Example: ECB

```
; Should be compiled in ascii mode
;
String$ = "Hello this is a test for AES"

*CipheredString   = AllocateMemory(Len(String$)+1) ; Space for the string and its
```

```
*DecipheredString = AllocateMemory(Len(String$)+1) ; null terminating character (ASCII mode)

If AESEncoder(@String$, *CipheredString, Len(String$), ?Key, 128, 0, #PB_Cipher_ECB)
  Debug "Ciphered: "+PeekS(*CipheredString)

  AESDecoder(*CipheredString, *DecipheredString, Len(String$), ?Key, 128, 0, #PB_Cipher_ECB)
  Debug "Deciphered: "+PeekS(*DecipheredString)
EndIf

DataSection
  Key:
    Data.b $06, $a9, $21, $40, $36, $b8, $a1, $5b, $51, $2e, $03, $d5, $34, $12, $00, $06
EndDataSection
```

70.3 AESDecoder

Syntax

```
Result\$ = AESDecoder(*Input, *Output, Size, *Key, Bits, *InitializationVector [, Mode])
```

Description

Decodes the specified '*Input' buffer using the AES algorithm into the '*Output' buffer. 'Bits' specify the size of the key used by the ciphering. Valid values are 128, 192 and 256. The 'InitializationVector' is a random data block, used to initialize the ciphering to avoid breach in decoding (only needed when using the #PB_Cipher_CBC mode). Its size depends of the 'Bits' parameter: l6 bytes for 128 bit encryption, 24 bytes for 196 bit and 32 bytes for 256 bit.

The optional 'Mode' parameter can be one of the following value:

```
#PB_Cipher_CBC: Default mode of encoding (Cipher Block Chaining).
  Needs an '*InitializationVector'.
                Recommended as more secure than ECB mode.
#PB_Cipher_ECB: Alternative mode (Electronic CodeBook). It doesn't
  uses random value nor chaining
                (each block is ciphered independently) making it less
  strong the CBC.
```

For more information about AES and source examples, see AESEncoder() .

70.4 DESFingerprint

Syntax

```
Result\$ = DESFingerprint(Password$, Key$)
```

Description

This algorithm is based on the DES (Data Encryption Standard) crypt method to generates a 13 characters string. This string is mean to be unique and non-reversible implying a strong encryption, hardly crackable when the password is correctly chosen.

The Password$ can be up to 8 characters long (all further characters are simply ignored). MD5Fingerprint() can be used to hash a bigger buffer. The Key$ is also known as the 'Salt' parameter, well know from Linux/Unix/BSD users. When using a 2 characters long Key$, this function returns a 'Salt2' string, compatible which any standard linux hash password (/etc/passwd). This function is based on the opensource crypt() function.

Example:

```
Debug DESFingerprint("Password", "Key007")
Debug DESFingerprint("NewPass", "Key007")
```

70.5 ExamineMD5Fingerprint

Syntax

```
Result = ExamineMD5Fingerprint(#Fingerprint)
```

Description

Initializes the calculation of a MD5 (Message Digest 5) checksum in several steps. Unlike the MD5Fingerprint() function this allows to calculate the checksum of large data without the need to load it all into one continuous memory buffer. See MD5Fingerprint() for more information on the MD5 algorithm.

The '#Fingerprint' value will be used in later calls to refer to this checksum calculation. If #PB_Any is used as '#Fingerprint' parameter, the value returned as a result will be used to identify the calculation.

NextFingerprint() can be used to add memory blocks into the calculation and FinishFingerprint() to finish the calculation and read the MD5 hash.

Example:

```
*Buffer = AllocateMemory(200)                    ; Prepare a buffer with data
If *Buffer
  PokeS(*Buffer, "The quick brown fox jumps over the lazy dog.")
  Length = MemoryStringLength(*Buffer)

  If ExamineMD5Fingerprint(1)                    ; start the calculation
    NextFingerprint(1, *Buffer, Length/2)          ; calculate part 1
    NextFingerprint(1, *Buffer+Length/2, Length/2) ; calculate part 2

    MD5$ = FinishFingerprint(1)                  ; finish calculation
    Debug "MD5 checksum = " + MD5$

    MD5$ = MD5Fingerprint(*Buffer, Length)         ; compare to a calculation in 1 step
```

```
      Debug "MD5 checksum = " + MD5$
    EndIf

    FreeMemory(*Buffer)
  EndIf
```

70.6 ExamineSHA1Fingerprint

Syntax

```
Result = ExamineSHA1Fingerprint(#Fingerprint)
```

Description

Initializes the calculation of a SHA1 (Secure Hash Algorithm 1) checksum in several steps. Unlike the SHA1Fingerprint() function this allows to calculate the checksum of large data without the need to load it all into one continuous memory buffer. See SHA1Fingerprint() for more information on the SHA1 algorithm.

The '#Fingerprint' value will be used in later calls to refer to this checksum calculation. If #PB_Any is used as '#Fingerprint' parameter, the value returned as a result will be used to identify the calculation.

NextFingerprint() can be used to add memory blocks into the calculation and FinishFingerprint() to finish the calculation and read the SHA1 hash.

Example:

```
*Buffer = AllocateMemory(200)                    ; Prepare a buffer with data
If *Buffer
  PokeS(*Buffer, "The quick brown fox jumps over the lazy dog.")
  Length = MemoryStringLength(*Buffer)

  If ExamineSHA1Fingerprint(1)                   ; start the calculation
    NextFingerprint(1, *Buffer, Length/2)          ; calculate part 1
    NextFingerprint(1, *Buffer+Length/2, Length/2) ; calculate part 2

    SHA1$ = FinishFingerprint(1)                 ; finish calculation
    Debug "SHA1 checksum = " + SHA1$

    SHA1$ = SHA1Fingerprint(*Buffer, Length)       ; compare to a calculation in 1 step
    Debug "SHA1 checksum = " + SHA1$
  EndIf

  FreeMemory(*Buffer)
EndIf
```

70.7 FinishCipher

Syntax

```
Result = FinishCipher(#Cipher)
```

Description

Finish a cipher stream previously started with StartAESCipher() .

Note: This command should be called to finish a cipher calculation, even if the cipher is actually no longer needed as it does free any data associated with the cipher calculation.

70.8 NextFingerprint

Syntax

```
NextFingerprint(#Fingerprint, *Buffer, Size)
```

Description

Add a new memory buffer into the calculation of a checksum started by ExamineMD5Fingerprint() or ExamineSHA1Fingerprint() . The checksum returned at the end of the calculation will include all the added buffers as if the checksum was calculated with all of them in one continuous memory buffer.

See ExamineMD5Fingerprint() or ExamineSHA1Fingerprint() for a code example and more information.

70.9 FinishFingerprint

Syntax

```
Result\$ = FinishFingerprint(#Fingerprint)
```

Description

Finishes the calculation of a checksum started by ExamineMD5Fingerprint() or ExamineSHA1Fingerprint() and returns the checksum as a string.

Note: This command should be called to finish a checksum calculation, even if the checksum is actually no longer needed as it frees up any data associated with the checksum calculation as well.

See ExamineMD5Fingerprint() or ExamineSHA1Fingerprint() for a code example and more information.

70.10 IsFingerprint

Syntax

```
Result = IsFingerprint(#Fingerprint)
```

Description

Tests if the given #Fingerprint is a valid fingerprint calculation created by ExamineMD5Fingerprint() or ExamineSHA1Fingerprint() .

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid, else it returns zero.

70.11 MD5FileFingerprint

Syntax

```
Result\$ = MD5FileFingerprint(Filename$)
```

Description

Returns a 32 characters long MD5 (Message Digest 5) hash code for the specified file. If the file isn't found or an error has happened, Result$ will be empty.

See MD5Fingerprint() for more information on the MD5 algorithm.

70.12 MD5Fingerprint

Syntax

```
Result\$ = MD5Fingerprint(*Buffer, Size)
```

Description

Returns a 32 characters long MD5 (Message Digest 5) hash code. Here is a quick explanation taken from the RFC 1321 on MD5:

'The algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest. The MD5 algorithm is intended for digital signature applications.'

MD5 hashs are often used for password encryption, providing a very strong and standard protection. More information can be found in the RFC 1321: http://www.ietf.org/rfc/rfc1321.txt.

MD5FileFingerprint() can be used to calculate the MD5 checksum of a file.
ExamineMD5Fingerprint() can be used to calculate a MD5 checksum in multiple steps.

Example 1 (a string as memory buffer):

```
1  test.s = "This is a test string!"
2  Debug MD5Fingerprint(@test, StringByteLength(test))
```

Example 2 (with memory buffer):

```
1  *Buffer = AllocateMemory(500)
2  If *Buffer
3    PokeS(*Buffer, "The quick brown fox jumps over the lazy dog.")
4    MD5$ = MD5Fingerprint(*Buffer, MemorySize(*Buffer))
5    Debug "MD5 Fingerprint = " + MD5$
6    FreeMemory(*Buffer)  ; would also be done automatically at the end
     of the program
7  EndIf
```

70.13 SHA1Fingerprint

Syntax

```
Result\$ = SHA1Fingerprint(*Buffer, Size)
```

Description

Returns a 40 characters long SHA1 (Secure Hash Algorithm 1) hash code of the given memory buffer. SHA1 can be used to calculate a checksum to verify that a 'message' has not been altered. Unlike CRC32 it is nearly impossible to modify the original message and still produce the same SHA1 fingerprint. Here is a quick explanation taken from the RFC 3174 on SHA1:

'The SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify.'

More information can be found in the RFC 3174: http://www.ietf.org/rfc/rfc3174.txt.

SHA1FileFingerprint() can be used to calculate the SHA1 checksum of a file. ExamineSHA1Fingerprint() can be used to calculate a SHA1 checksum in multiple steps.

Example 1 (a string as memory buffer):

```
1  test.s = "This is a test string!"
2  Debug SHA1Fingerprint(@test, StringByteLength(test))
```

Example 2 (with memory buffer):

```
1  *Buffer = AllocateMemory(500)
2  If *Buffer
3    PokeS(*Buffer, "The quick brown fox jumps over the lazy dog.")
4    SHA1$ = SHA1Fingerprint(*Buffer, MemorySize(*Buffer))
5    Debug "SHA1 Fingerprint = " + SHA1$
6    FreeMemory(*Buffer)  ; would also be done automatically at the end
     of the program
7  EndIf
```

70.14 SHA1FileFingerprint

Syntax

```
Result\$ = SHA1FileFingerprint(Filename$)
```

Description

Returns a 40 characters long SHA1 (Secure Hash Algorithm 1) hash code for the specified file. If the file isn't found or an error has happened, Result$ will be empty.

See SHA1Fingerprint() for more information on the SHA1 algorithm.

70.15 CRC32Fingerprint

Syntax

```
Result = CRC32Fingerprint(*Buffer, Size [, InitialValue])
```

Description

Perform a CRC32 checksum on the given buffer. CRC32 is a 32 bit fingerprint not intended for password storage as it's easily crackable, but for quick data integrity check. For example, zip files have a CRC32 checksum at the end of each file to be sure that the zip is not corrupted. The main advantage of CRC32 over MD5 or other hash algoritm is its very high speed.

The optional InitialValue parameter can be used to pass the checksum of a previous buffer. The result will then be the checksum of both buffers, as if it were calculated from one continuous buffer. This way calculating the checksum of large data can be broken down into a number of calculations on smaller buffers.

Example:

```
*Buffer = AllocateMemory(200)                ; Prepare a buffer with data
If *Buffer
  PokeS(*Buffer, "The quick brown fox jumps over the lazy dog.")
  Length = MemoryStringLength(*Buffer)

  CRC32 = CRC32Fingerprint(*Buffer, Length)    ; Calculate the checksum in 1 step
  Debug "CRC32 checksum = " + Str(CRC32)

  Part1 = CRC32Fingerprint(*Buffer, Length/2) ; Calculate the checksum in 2 steps
  CRC32 = CRC32Fingerprint(*Buffer+Length/2, Length/2, Part1)
  Debug "CRC32 checksum = " + Str(CRC32)

  FreeMemory(*Buffer)
EndIf
```

70.16 CRC32FileFingerprint

Syntax

```
Result = CRC32FileFingerprint(Filename$)
```

Description

Returns the CRC32 checksum of the given file. If the file cannot be read the result is zero. CRC32 is a 32 bit fingerprint not intended for password storage as it's easily crackable, but for quick data integrity checks. The main advantage of CRC32 over MD5 or other hash algorithm is its very high speed.

70.17 Base64Decoder

Syntax

```
Result = Base64Decoder(*InputBuffer, InputSize, *OutputBuffer,
    OutputSize)
```

Description

Decodes the specified Base64 encoded buffer. 'Result' will contain the length (in bytes) of the decoded string.

The output buffer can be up to 33% smaller than the input buffer, with a minimum size of 64 bytes. It's recommended to get a slightly larger buffer, like 30% smaller to avoid overflows. Base64Encoder() can be used to encode data in Base64 format.

Example:

```
1   test.s = "This is a test string!"
2   dec.s = Space(1024)
3   enc.s = Space(1024)
4
5   Debug Base64Encoder(@test, StringByteLength(test), @enc, 1024)
6   Debug enc
7
8   Debug Base64Decoder(@enc, StringByteLength(enc), @dec, 1024)
9   Debug dec
```

70.18 Base64Encoder

Syntax

```
Result = Base64Encoder(*InputBuffer, InputSize, *OutputBuffer,
    OutputSize)
```

Description

Encodes the specified buffer using the Base64 algorithm. This is widely used in e-mail programs but can be useful for any other programs which need an ASCII only (7 bit, only from 32 to 127 characters) encoding for raw binary files. 'Result' will contain the length (in bytes) of the encoded string.

The output buffer should be at last 33% bigger than the input buffer, with a minimum size of 64 bytes. It's recommended to get a slightly larger buffer, like 35% bigger to avoid overflows. The data can be decoded later with Base64Decoder() .

Example:

```
1   test.s = "This is a test string!"
2   dec.s = Space(1024)
3   enc.s = Space(1024)
4
```

```
5   Debug Base64Encoder(@test, StringByteLength(test), @enc, 1024)
6   Debug enc
7
8   Debug Base64Decoder(@enc, StringByteLength(enc), @dec, 1024)
9   Debug dec
```

70.19 StartAESCipher

Syntax

```
Result = StartAESCipher(#Cipher, *Key, Bits, *InitializationVector,
    Mode)
```

Description

Initializes an AES cipher stream using the specified '*Key'. 'Bits' specifies the size of the key used by the ciphering. Valid values are 128, 192 and 256. The 'InitializationVector' is a random data block, used to initialize the ciphering to avoid breach in decoding (only needed when using the #PB_Cipher_CBC mode). Its size depends of the 'Bits' parameter: l6 bytes for 128 bit encryption, 24 bytes for 196 bit and 32 bytes for 256 bit. Once a cipher is finished, FinishCipher() has to be called.

The optional 'Mode' parameter can be a combination of the following value:

```
  #PB_Cipher_Decode: The stream is used to decode data.
  #PB_Cipher_Encode: The stream is used to encode data.

with

  #PB_Cipher_CBC: Default mode of encoding (Cipher Block Chaining).
   Needs an '*InitializationVector'.
                  Recommended as more secure than ECB mode.
  #PB_Cipher_ECB: Alternative mode (Electronic CodeBook). It doesn't
   uses random value nor chaining
                  (each block is ciphered independently) making it less
   strong the CBC.
```

New buffers to be encoded or decoded can be added with AddCipherBuffer() .

For more information about AES, see AESEncoder() .

70.20 OpenCryptRandom

Syntax

```
Result = OpenCryptRandom()
```

Description

Opens the cryptographic safe pseudorandom number generator. The CryptRandom() and CryptRandomData() commands can be used to read data from the opened generator.

This generator provides random data which is strong enough for cryptographic purposes such as generating keys for the AESEncoder() function. The source for the random data is the "/dev/urandom" device on Linux or Mac OSX and the "Microsoft Cryptography API" on Windows.

Returns non-zero if the random number generator could be sucessfully opened. If the result is zero then there is no cryptographic safe random number generator available on the system.

See the CryptRandomData() command for an example.

70.21 CloseCryptRandom

Syntax

```
CloseCryptRandom()
```

Description

Closes the cryptographic safe pseudorandom number generator that was opened with OpenCryptRandom() and frees its resources.

70.22 CryptRandom

Syntax

```
Result = CryptRandom(Maximum)
```

Description

Returns a random number (integer) which lies between (and including) 0 and the Maximum value from the cryptographic safe pseudorandom number generator. The generator has to be opened first with the OpenCryptRandom() command. 'Maximum' may not exceed the positive long value: 2147483647.

Important: Using a 'Maximum' value which is not one less than a power of two will cause certain numbers to be more likely than others which could be used for a statistical attack. This is the result of dividing the generated random number to fit the specified range.

To generate larger amounts of random data, use the CryptRandomData() function. To generate random numbers from the faster but not cryptographic safe pseudorandom number generator, use the Random() function.

70.23 CryptRandomData

Syntax

```
Result = CryptRandomData(*Buffer, Length)
```

Description

Fills the specified memory buffer with random data from the cryptographic safe pseudorandom number generator. The generator has to be opened first with the OpenCryptRandom() command.

To generate random data from the faster but not cryptographic safe pseudorandom number generator, use the RandomData() function.

Example:

```
*Key = AllocateMemory(16)

If OpenCryptRandom() And *Key
  CryptRandomData(*Key, 16)

  Text$ = "Generated Key:"
  For i = 0 To 15
    Text$ + " " + RSet(Hex(PeekB(*Key+i), #PB_Byte), 2, "0")
  Next i

  CloseCryptRandom()
Else
  Text$ = "Key generation is not available"
EndIf

MessageRequester("Example", Text$)
```

; ; Clipboard library documentation ; ; (c) 2010 - Fantaisie Software ;

Chapter 71

Clipboard

The clipboard is the standard way to share information and data between applications, which are currently running on the OS. It also gives the user a temporary location, where information and data may be stored and accessed quicky and easily. For instance, when text is cut within an editor, that text goes to the clipboard, where it may be retrieved later, by pasting it into another application. PureBasic gives the programmer the ability to develop applications capable of cutting, copying and pasting text or images via the standard clipboard.
;———

71.1 ClearClipboard

Syntax

```
ClearClipboard()
```

Description

Clears the clipboard. This means that any data contained within the clipboard, is totally flushed and is no longer available from the clipboard.

None.

This function does not return any value.
;———

71.2 GetClipboardImage

Syntax

```
Result = GetClipboardImage(#Image [, Depth])
```

Description

Creates a new #Image from the clipboard image data (if any). If #PB_Any is used as the '#Image' parameter, the new image number will be returned to Result. If something is wrong a 0 value is returned.

If 'Depth' is omitted or set to #PB_Image_DisplayFormat, the image format is taken from the desktop format. Valid 'Depth' values are: 1, 2, 4, 8, 16, 24 and 32 bits. When manipulating true-color formats, it is advised to use the 24 or 32 bit depth values, in order to avoid color information loss. If the images are to be used only for screen display, #PB_Image_DisplayFormat will provide the fastest performance.

The image may be freed by using the FreeImage() function. For further use of the image, there are functions available such as DrawImage() or ImageGadget() .
;

71.3 GetClipboardText

Syntax

```
Text\$ = GetClipboardText()
```

Description

Returns the text string currently contained within the clipboard.

None.

Text$ - The string containing the clipboard contents. If the clipboard contains no data, the return value is an empty string. If the clipboard contains data, but the clipboard contains no text data, the return value is an empty string. ;

71.4 SetClipboardImage

Syntax

```
SetClipboardImage(#Image)
```

Description

Places a copy of the given #Image into the clipboard. If the clipboard currently contains an image then it will be overwritten.

An image is available for use with this function after being created with CreateImage() or loaded with LoadImage() .

This function does not return any value.

Example:

```
If CreateImage(0, 26, 20)
  StartDrawing(ImageOutput(0))
    Box(0, 0, 26, 20, RGB(255, 255, 255))
    Circle(13, 10, 5, RGB(255, 0, 0))
  StopDrawing()
  SetClipboardImage(0)
EndIf

; now if you open a graphics application and paste the clipboard contents,
; you will see a Japanese flag ;-)
```

;——

71.5 SetClipboardText

Syntax

```
SetClipboardText(Text$)
```

Description

Stores a string into the clipboard. If the clipboard already contains text, it will be overwritten.

Text$ - The string you want to store in the clipboard.

This function does not return any value.

Chapter 72

Console

This library allow the programmer to create console mode applications. This is used to create small programs that don't require a user interface or be executed from the function line. Also, console functions can be really helpful to debug a program by printing out some information to the console, without stopping the program flow.

If your program is intended to be a pure console application (i.e. not a GUI application which sometimes opens a console) then you must remember to set the executable format to 'Console' when you are compiling your programs.

You should start with the OpenConsole() function, since you must use that function to open a console (character mode) display window before you use any other console-related functions.

Please note, that if you create console programs with PureBasic you will still need Windows 95 or later to run them. These programs must be executed only from the Windows function prompt. These programs are not real MS-DOS programs!

72.1 ClearConsole

Syntax

```
ClearConsole()
```

Description

Clears the whole console content using the current background colour. The background color is set with ConsoleColor() . The console has to be in graphical mode, see EnableGraphicalConsole() .

Example:

```
If OpenConsole()
  EnableGraphicalConsole(1)

  PrintN("You will never see me")
  ClearConsole()

  PrintN("Press return to exit")
  Input()
```

```
  EndIf
```

Supported OS

Windows

72.2 CloseConsole

Syntax

```
CloseConsole()
```

Description

Close the console previously opened with OpenConsole() . Once the console has been closed, it's not possible to use any console-related functions unless you open the console again. The console will automatically be closed when your program ends.

Example:

```
  For i = 0 To 4
    If OpenConsole()
      PrintN("This is console #"+Str(i))
      PrintN("Press return to close console")
      Input()
      CloseConsole()
    EndIf
  Next
```

72.3 ConsoleError

Syntax

```
ConsoleError(Message$)
```

Description

Writes the Message string (plus a newline) to the standard error output of the program. This output can be read for example with the ReadProgramError() function of the Process library.

72.4 ConsoleTitle

Syntax

```
ConsoleTitle(Title$)
```

Description

Changes the console title to the specified string. The console title is typically shown in the title bar of the window which the console is in (when you are viewing the console in a graphical environment such as your desktop).

Under Windows, the console title is also the text shown in the start bar and in the task manager representing your console application.

Example:

```
If OpenConsole()
  ConsoleTitle("The ConsoleTitle example program")
  PrintN("Press return to exit")
  Input()
EndIf
```

Supported OS

Windows

72.5 ConsoleColor

Syntax

```
ConsoleColor(CharacterColor, BackgroundColor)
```

Description

Change the colors used by the text display. Any characters printed after calling this function will use the new colors. The color values range from 0 to 15, which are the 16 colors available in console mode.

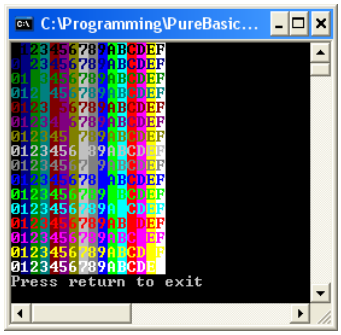
Parameters:
CharacterColor - The foreground color to use. This is the color that the characters are drawn with.
BackgroundColor - The background color to use. This is the color that surrounds each character.

Both colors can have a value from 0 to 15 to represent the color. The colors which each of those values represent are:

```
0 - Black (default background)
1 - Blue
2 - Green
3 - Cyan
4 - Red
5 - Magenta
6 - Brown
7 - Light grey (default foreground)
8 - Dark grey
9 - Bright blue
10 - Bright green
11 - Bright cyan
12 - Bright red
13 - Bright magenta
14 - Yellow
15 - White
```

Example:

```
If OpenConsole()
  For Foreground = 0 To 15
    For Background = 0 To 15
      ConsoleColor(Foreground, Background)
      Print(Right(Hex(Background), 1))
    Next

    PrintN("")
  Next

  ConsoleColor(7, 0)
  PrintN("Press return to exit")
  Input()
EndIf
```



Supported OS

Windows

72.6 EnableGraphicalConsole

Syntax

```
EnableGraphicalConsole(State)
```

Description

Changes the way the characters are drawn on the console. If 'State' is 1, then console will be switched to graphical mode, else if 'State' is 0, it will switch back to text mode.

The default mode of the console is the text mode, which means the text can't be positioned anywhere in the console, but redirections (trough pipes) work correctly.

When being in the graphical mode, functions like ClearConsole() or ConsoleLocate() are available and the text can be positioned anywhere in the console screen, which allow to do console games or text only applications (which can be useful when accessed remotely trough telnet or ssh). The redirection (pipes) doesn't work correctly when being in graphical mode.

Example:

```
If OpenConsole()
  EnableGraphicalConsole(1)
```

```
    ConsoleLocate(7, 8)
    PrintN("Press return to exit")
    Input()
  EndIf
```

Supported OS

Windows

72.7 Inkey

Syntax

```
String\$ = Inkey()
```

Description

Returns a character string if a key is pressed during the call of Inkey(). It doesn't interrupt (halt) the program flow. If special keys (non-ASCII) have to be handled, RawKey() should be called after Inkey().

Example:

```
  If OpenConsole()
    PrintN("Press Escape to exit")

    Repeat
      KeyPressed$ = Inkey()

      If KeyPressed$ <> ""

        PrintN("You pressed: " + KeyPressed$)
        PrintN("It has a raw code of: "+Str(RawKey()))

      ElseIf RawKey()

        PrintN("You pressed a non ASCII key.")
        PrintN("It has a raw code of: "+Str(RawKey()))

      Else
        Delay(20) ; Don't eat all the CPU time, we're on a multitask OS
      EndIf

    Until KeyPressed$ = Chr(27) ; Wait until escape is pressed
  EndIf
```

Notes:

The ASCII codes and numeric values reported in this description may change depending on the code page you have configured at boot time for keyboard input. However, the concepts should be the same and you can use the above example to figure out the real values for your system.

A table with ASCII codes is available here .

72.8 Input

Syntax

```
String\$ = Input()
```

Description

Allows the program to catch an entire line of characters. This function locks the program execution and waits until the user presses the return key.

If the console is in graphical mode (see EnableGraphicalConsole()), the line can't be longer than the console width (no more keys are accepted when the line gets too long).

In non-graphical mode, a special return value of #PB_Input_Eof (equals Chr(4)) will be received if the user enters a single Ctrl+D in the console, or a file that has been redirected to the programs input has reached its end. For compatibility with other console applications on windows, #PB_Input_Eof is also received when Ctrl+Z is entered in the console.

If not line based or raw input is needed, ReadConsoleData() can be used in non-graphical mode.

Example:

```
If OpenConsole()
  Print("Enter your name and press return: ")
  Name$ = Input()

  PrintN("Hello " + Name$ + ", nice to meet you.")
  PrintN("Press return to exit")
  Input()
EndIf
```

72.9 ConsoleLocate

Syntax

```
ConsoleLocate(x, y)
```

Description

Moves the cursor to the given position, in character co-ordinates. Any text you print after calling this function will start from the specified co-ordinates.

The console has to be in graphical mode, see EnableGraphicalConsole() .

Parameters:
x - The horizontal position in the console to move to (starting from 0)
y - The vertical position in the console to move to (starting from 0)

Example:

```
If OpenConsole()
  EnableGraphicalConsole(1)

  For i = 0 To 200
    ConsoleLocate(Random(79), Random(24))
```

```
      Print("*")
    Next

    ConsoleLocate(30, 10)
    PrintN("Press return to exit")
    Input()
  EndIf
```

Supported OS

Windows

72.10 ConsoleCursor

Syntax

```
ConsoleCursor(Height)
```

Description

Changes the display of the cursor, which is the indicator used to show where the next displayed character will be drawn. This function allows you to change the height of the cursor. By default the cursor is a flashing underline in consoles under the Windows OS. Note that you might need to make the console window full-screen to see the true effect of this function.

The console has to be in graphical mode, see EnableGraphicalConsole() .

Parameters:
Height - The new height of the cursor indicator. This value can be zero (for an invisible cursor) or range from 1 to 10 for the height of the cursor. Currently, 3 values are available:

```
1 : Underline cursor (default)
5 : Mid-height cursor
10: Full-height cursor
```

Example:

```
  If OpenConsole()
    EnableGraphicalConsole(1)

    For CursorHeight = 0 To 10
      ConsoleCursor(CursorHeight)
      PrintN("Press return to increase cursor size")
      Input()
    Next

    PrintN("Press return to exit")
    Input()
  EndIf
```

Supported OS

Windows

72.11 Print

Syntax

```
Print(Text$)
```

Description

Displays the specified 'Text$' in the console. In graphical mode the length of the string can't excess the width of the console, otherwise the string will be truncated (have the end cut off).

In graphical mode it's possible to change the position with the function ConsoleLocate() . To change the appearance of the string when it is printed, the function ConsoleColor() should be used.

The cursor will be moved to the next character after the end of the string that is printed. If you print over the right edge of the console the text will wrap around to the left edge on the next line down. If you print off the bottom of the console window, the console window will scroll its contents up.

To output raw data on the non-graphical console (for pipe communication) WriteConsoleData() can be used.

Example:

```
If OpenConsole()
  Print("This is quite a long string.")
  Print("You see how this one joins onto the end of the previous one?")
  Print("That is because the cursor moves to the end of the string and not onto a new line.")
  Print("Hopefully the text will also have been wrapped by now.")

  PrintN("")
  PrintN("")
  PrintN("Press return to exit")
  Input()
EndIf
```

72.12 PrintN

Syntax

```
PrintN(Text$)
```

Description

Displays the specified 'Text$' in the console and adds a new line. In graphical mode the length of the string can't excess the width of the console, otherwise the string will be truncated (have the end cut off).

In graphical mode it's possible to change the position with the function ConsoleLocate() . To change the appearance of the string when it is printed, the function ConsoleColor() should be used.

The cursor will be moved to the start of the next line after the end of the string. If you print over the right edge of the console the text will wrap around to the left edge on the next line down. If you print off the bottom of the console window, the console window will scroll its contents up. You can use Print() instead, if you want to continue the output directly after last character.

To output raw data on the non-graphical console (for pipe communication) WriteConsoleData() can be used.

Example:

```
If OpenConsole()
  PrintN("This is quite a long string.")
  PrintN("You see how this one does not join onto the end of the previous one?")
  PrintN("That is because the cursor moves to the start of the next line after the end of the string.")

  PrintN("Press return to exit")
  Input()
EndIf
```

72.13 OpenConsole

Syntax

```
Result = OpenConsole()
```

Description

Open a console window. This function must be called before any other function of this library. Only one console can be opened at the same time in a PureBasic program.

If Result is 0, it has failed and all further call to console functions must be disabled. The console can be closed using the CloseConsole() function.

With EnableGraphicalConsole() the console can be switched between text and graphics mode.

On Microsoft Windows, there are two different executable formats: Win32 and Console. If you want to create a standard console application, like 'dir', 'del' etc.. you must compile the executable using the 'Console' format (Compiler Option menu in the PureBasic IDE). On Linux and OS X, this command has no effect, as there is no special 'Console' format.

Example:

```
OpenConsole()
PrintN("Waiting 5 secs before quit...")
Delay(5000)
```

72.14 ReadConsoleData

Syntax

```
Result = ReadConsoleData(*Buffer, Size)
```

Description

Reads raw input from the console. This function is only supported in non-graphical mode. It can be used to read in data that is not line-based or text like files that were redirected to the program through a pipe.

The function reads up to 'Size' number of bytes into the specified buffer, which should be reserved before with AllocateMemory .

This function waits until there is some input to read. It will only return without reading data if there was an error or an EOF (End Of File) condition.

Returns the number of bytes actually read from the input. If zero is returned, this means that there is no more input to read. (an end of file was received)

Example:

```
; This reads a passed image from the console and displays it in a window
; Compile this to an exe and run it like "myexe < image.bmp"
;
; (set "Executable format" To "Console" in the compiler options!)
; (works only with Bitmaps and icons unless you use an Image Decoder)
;
OpenConsole()
TotalSize = 0
BufferFree = 10000
*Buffer = AllocateMemory(BufferFree)

Repeat
  ReadSize = ReadConsoleData(*Buffer+TotalSize, BufferFree) ; read a block of data
  TotalSize + ReadSize
  BufferFree - ReadSize
  If BufferFree < 100  ; resize the buffer if it is not large enough
    BufferFree = 10000
    *Buffer = ReAllocateMemory(*Buffer, TotalSize+10000)
  EndIf
Until ReadSize = 0 ; once 0 is returned, there is nothing else to read

If TotalSize > 0 ; display the image if successful
  If CatchImage(0, *Buffer, TotalSize)
    If OpenWindow(0, 0, 0, ImageWidth(0), ImageHeight(0), "Image", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
      ImageGadget(0, 0, 0, ImageWidth(0), ImageHeight(0), ImageID(0))
      Repeat
      Until WaitWindowEvent() = #PB_Event_CloseWindow
      End
    EndIf
  EndIf
EndIf
MessageRequester("Error", "Not a valid image.")
```

72.15 RawKey

Syntax

```
Result = RawKey()
```

Description

Returns the rawkey value of the last Inkey() function call. It's useful for extended (non-ASCII) keys (for example, function keys, arrows, etc).

Notes:
It is not only alphanumeric keys that have an ASCII value. The escape key (27), return key (13) tab key (9) and backspace key (8) are just four examples. A table with ASCII codes you find here .

Example:

```
If OpenConsole()
  PrintN("Press Escape to exit")

  Repeat
    KeyPressed$ = Inkey()

    If KeyPressed$ <> ""

      PrintN("You pressed: " + KeyPressed$)
      PrintN("It has a raw code of: "+Str(RawKey()))

    ElseIf RawKey()

      PrintN("You pressed a non ASCII key.")
      PrintN("It has a raw code of: "+Str(RawKey()))

    Else
      Delay(20) ; Don't eat all the CPU time, we're on a multitask OS
    EndIf

  Until KeyPressed$ = Chr(27) ; Wait until escape is pressed
EndIf
```

Supported OS

Windows

72.16 WriteConsoleData

Syntax

```
Result = WriteConsoleData(*Buffer, Size)
```

Description

Writes raw data to the console output. This function is only supported in non-graphical mode. It can be used to output data other than text to the console that can then be redirected to a file or another program.

The function writes 'Size' number of bytes from the specified buffer into the console output. The memory area can be allocated and filled using functions from the Memory library.

Returns the number of bytes actually written to the output.

Chapter 73

Database

The database library is an easy set of functions to access SQLite, PostgreSQL or any database type (Oracle, MySQL, Access, etc) via ODBC. Accessing and updating data is done using SQL queries, therefore it is necessary to have an understanding of SQL syntax.

Here are some links about SQL syntax:
W3Schools SQL Tutorial
SQLite SQL functions
PostgreSQL manual

Database programming starts by initializing the database environment using UseODBCDatabase() , UseSQLiteDatabase() , and UsePostgreSQLDatabase() .

Note: Under Windows, prior to using an ODBC database, it is necessary to establish an ODBC "User Data Source" which makes your database available via ODBC and useable with this Database library. For more information, refer to Windows's ODBC help document.

73.1 CloseDatabase

Syntax

```
CloseDatabase(#Database)
```

Description

Close the specified #Database (and connections/transactions if any). No further operations are allowed on this database.

Note: all remaining opened databases are automatically closed when the program ends.

73.2 DatabaseColumns

Syntax

```
Result = DatabaseColumns(#Database)
```

Description

Returns the numbers of columns (fields) in the opened #Database.

73.3 DatabaseColumnName

Syntax

```
Text\$ = DatabaseColumnName(#Database, Column)
```

Description

Return the name of the specified column in the #Database.

73.4 DatabaseColumnSize

Syntax

```
Result = DatabaseColumnSize(#Database, Column)
```

Description

Return the size of the specified column in the #Database. It is especially useful when the size of the column can change depending of the records, like a blob or string column.

73.5 DatabaseColumnType

Syntax

```
Result = DatabaseColumnType(#Database, Column)
```

Description

Return the type of the specified column in the #Database. If Result is 0, the type is undefined or the function has failed (e.g. it was not possible to determine the data type)

Type values can be:

```
#PB_Database_Long   : Numeric format (a Long (.l) in PureBasic)
#PB_Database_String: String format (a String (.s) in PureBasic)
#PB_Database_Float  : Numeric float format (a Float (.f) in PureBasic)
#PB_Database_Double: Numeric double format (a Double (.d) in
 PureBasic)
#PB_Database_Quad   : Numeric quad format (a Quad (.q) in PureBasic)
#PB_Database_Blob   : Blob format
```

73.6 DatabaseDriverDescription

Syntax

```
Text\$ = DatabaseDriverDescription()
```

Description

Returns the description of the current database driver. Drivers are listed using the ExamineDatabaseDrivers() and NextDatabaseDriver() functions.

This is an ODBC database specific command.

73.7 DatabaseDriverName

Syntax

```
Text\$ = DatabaseDriverName()
```

Description

Return the name of the current database driver. Drivers are listed using the ExamineDatabaseDrivers() and NextDatabaseDriver() functions.

This is an ODBC database specific command.

73.8 DatabaseError

Syntax

```
Error\$ = DatabaseError()
```

Description

Returns a description of the last database error in text format. This is especially useful with the following functions: OpenDatabase() , DatabaseQuery() and DatabaseUpdate() .

Example

```
; First, connect to a database with an employee table
;
If DatabaseQuery(#Database, "SELECT * FROM employee") ; Get all the records in the 'employee' table
  ; ...
  FinishDatabaseQuery(#Database)
Else
  MessageRequester("Error", "Can't execute the query: "+DatabaseError())
EndIf
```

73.9 DatabaseID

Syntax

```
DatabaseID = DatabaseID(#Database)
```

Description

Returns the unique ID which identifies the given '#Database' in the operating system. This function is useful when another library needs a database reference.

73.10 DatabaseQuery

Syntax

```
Result = DatabaseQuery(#Database, Request$)
```

Description

Executes the SQL Request$ on the #Database. If Result is 0, then the query can't be executed correctly (due to a SQL error or a badly-formatted query), otherwise the query has succeeded and NextDatabaseRow() can be used to list returned records (see the example below). In the event of an error, the error text can be retrieved with DatabaseError()). It is safe to use NextDatabaseRow() even if the request doesn't return any records. To execute a request that doesn't return any records, use DatabaseUpdate() instead of DatabaseQuery(). Once the query results aren't needed anymore, FinishDatabaseQuery() has to be called to release all the query resources.

Example

```
; First, connect to a database with an employee table
;
If DatabaseQuery(#Database, "SELECT * FROM employee") ; Get all the records in the 'employee' table

  While NextDatabaseRow(#Database) ; Loop for each records
    Debug GetDatabaseString(#Database, 0) ; Display the content of the first field
  Wend

  FinishDatabaseQuery(#Database)
EndIf
```

73.11 DatabaseUpdate

Syntax

```
Result = DatabaseUpdate(#Database, Request$)
```

Description

Executes the SQL 'Request$' on the #Database. This function is similar to DatabaseQuery() but is independent from the NextDatabaseRow() function. Therefore it's not possible to do a 'SELECT' like

request with this function. This function is useful for updating records in the database. If Result is 0, then the query could not be executed correctly (due to a SQL fault or a badly-formatted query). The error text can be retrieved using DatabaseError()).

Example

```
; First, connect to a database with an employee table
;
If DatabaseQuery(#Database, "SELECT * FROM employee") ; Get all the records in the 'employee' table

  While NextDatabaseRow(#Database) ; Loop for each records

    ; Update the 'checked' field for each records, assuming the 'id' field is
    ; the first one in the 'employee' table
    ;
    DatabaseUpdate(#Database, "UPDATE employee SET checked=1 WHERE id="+GetDatabaseString(#Database, 0))
  Wend

  FinishDatabaseQuery(#Database)
EndIf
```

73.12 ExamineDatabaseDrivers

Syntax

```
Result = ExamineDatabaseDrivers()
```

Description

Examines the database drivers available on the system. If ODBC isn't installed or no drivers are available, it returns 0, otherwise NextDatabaseDriver() can be used to list all the drivers.

This is an ODBC database specific command.

73.13 FinishDatabaseQuery

Syntax

```
FinishDatabaseQuery(#Database)
```

Description

Finish the current #Database SQL query and release its associated resources. Query related functions like FirstDatabaseRow() or NextDatabaseRow() can't be used anymore.

Example

```
; First, connect to a database with an employee table
;
If DatabaseQuery(#Database, "SELECT * FROM employee") ; Get all the records in the 'employee' table
```

```
    While NextDatabaseRow(#Database) ; Loop for each records
      Debug GetDatabaseString(#Database, 0) ; Display the content of the first field
    Wend

    FinishDatabaseQuery(#Database)
  EndIf
```

73.14 FirstDatabaseRow

Syntax

```
Result = FirstDatabaseRow(#Database)
```

Description

Retrieves information about the first #Database row. If Result is 0, then no row is available To access fields within a row, GetDatabaseLong() , GetDatabaseFloat() , GetDatabaseString() can be used.

73.15 GetDatabaseBlob

Syntax

```
Result = GetDatabaseBlob(#Database, Column)
```

Description

Returns the content of the specified #Database column as a pointer to the blob memory. The Column parameter is a number starting from 0 which represents the first field. To determine the type of a column, DatabaseColumnType() can be used. To determine the size of the blob, DatabaseColumnSize() can be used. If 'Result' is 0, then the blob can't be retrieved or its content is empty.

Note: This function can be called only once for each column. Therefore if this value needs to be used more than once, the data has to be stored in a variable, since all subsequent calls will returns the wrong value. This is an ODBC limitation.

73.16 GetDatabaseDouble

Syntax

```
Result.d = GetDatabaseDouble(#Database, Column)
```

Description

Returns the content of the specified #Database column as a double precision floating point number. The Column parameter is a number starting from 0 which represents the first field. To determine the type of a column, DatabaseColumnType() can be used.

Note: This function can be called only once for each column. Therefore if this value needs to be used more than once, the data has to be stored in a variable, since all subsequent calls will returns the wrong value. This is an ODBC limitation.

73.17 GetDatabaseFloat

Syntax

```
Result.f = GetDatabaseFloat(#Database, Column)
```

Description

Returns the content of the specified #Database column as a floating point number. The Column parameter is a number starting from 0 which represents the first field. To determine the type of a column, DatabaseColumnType() can be used.

Note: This function can be called only once for each column. Therefore if this value needs to be used more than once, the data has to be stored in a variable, since all subsequent calls will returns the wrong value. This is an ODBC limitation.

73.18 GetDatabaseLong

Syntax

```
Result.l = GetDatabaseLong(#Database, Column)
```

Description

Returns the content of the specified #Database column as a long number. The Column parameter is a number starting from 0 which represents the first field. To determine the type of a column, DatabaseColumnType() can be used.

Note: This function can be called only once for each column. Therefore if this value needs to be used more than once, the data has to be stored in a variable, since all subsequent calls will returns the wrong value. This is an ODBC limitation.

73.19 GetDatabaseQuad

Syntax

```
Result.q = GetDatabaseQuad(#Database, Column)
```

Description

Returns the content of the specified #Database column as a quad number. The Column parameter is a number starting from 0 which represents the first field. To determine the type of a column, DatabaseColumnType() can be used.

Note: This function can be called only once for each column. Therefore if this value needs to be used more than once, the data has to be stored in a variable, since all subsequent calls will returns the wrong value. This is an ODBC limitation.

73.20 GetDatabaseString

Syntax

```
Text\$ = GetDatabaseString(#Database, Column)
```

Description

Returns the content of the specified #Database column as a string. The Column parameter is a number starting from 0 which represents the first field. To determine the type of a column, DatabaseColumnType() can be used.

Note: This function can be called only once for each column. Therefore if this value needs to be used more than once, the data has to be stored in a variable, since all subsequent calls will returns the wrong value. This is an ODBC limitation.

73.21 IsDatabase

Syntax

```
Result = IsDatabase(#DataBase)
```

Description

This function evaluates if the given #Database number is a valid and correctly-initialized database. This function is bullet-proof and can be used with any value. If Result is not zero then the object is valid and initialized, otherwise it returns zero. This is a good way to check that a database is ready to use.

73.22 NextDatabaseDriver

Syntax

```
Result = NextDatabaseDriver()
```

Description

Retrieves information about the next available database driver. If Result is 0, no more drivers are available. This function must be called after ExamineDatabaseDrivers() . To get information about the current driver, DatabaseDriverName() and DatabaseDriverDescription() can be used.

This is an ODBC database specific command.

73.23 NextDatabaseRow

Syntax

```
Result = NextDatabaseRow(#Database)
```

Description

Retrieves information about the next database row in the #Database. If Result is 0, then no more rows are available (i.e. reached the end of the table). To access fields within a row, GetDatabaseLong() , GetDatabaseFloat() , GetDatabaseString() can be used.

73.24 OpenDatabase

Syntax

```
Result = OpenDatabase(#Database, DatabaseName$, User$, Password$ [,
    Plugin])
```

Description

Open the specified database. If #PB_Any is used as '#Database' parameter, the new database number will be returned to 'Result'. If 'Result' is 0, the database can't be found or the user/password is incorrect (the error text can be retrieved with DatabaseError()). Password$ can be a null string if no password is required. 'Plugin' is an optional parameter and can be one of the following value:

```
  #PB_Database_ODBC       : The database will use ODBC backend
   (UseODBCDatabase()
 has to be called).
  #PB_Database_SQLite     : The database will use SQLite backend
   (UseSQLiteDatabase()
 has to be called).
  #PB_Database_PostgreSQL: The database will use PostgreSQL backend
   (UsePostgreSQLDatabase()
 has to be called).
```

If 'Plugin' isn't specified, then the first registered database plugin will be used.

73.25 OpenDatabaseRequester

Syntax

```
Result = OpenDatabaseRequester(#Database)
```

Description

Open the standard ODBC requester to choose which database to open. If #PB_Any is used as '#Database' parameter, the new database number will be returned to Result. If Result is 0, the database can't be opened (the error text can be retrieved with DatabaseError()).

This is an ODBC database specific command.

Note: This command is not supported on Linux and MacOS X and will return 0.

73.26 PreviousDatabaseRow

Syntax

```
Result = PreviousDatabaseRow(#Database)
```

Description

Retrieves information about the previous database row in the #Database. If Result is 0, then no more rows are available (i.e. reached the start of the table). To access to fields inside a row, GetDatabaseLong() , GetDatabaseFloat() , GetDatabaseString() can be used.

Note: If this function returns 0 despite additional rows being available before the current one, then the ODBC driver does not support data retrieval in a backwards direction. It is not mandatory for an ODBC driver to support this function (unlike NextDatabaseRow()). Of course, if this function works, it will work on every computer using the same driver.

SQLite databases don't support this command.

73.27 SetDatabaseBlob

Syntax

```
Result = SetDatabaseBlob(#Database, StatementIndex, *Buffer,
    BufferLength)
```

Description

Set the blob for future use with DatabaseUpdate() . 'StatementIndex' starts from 0, and indicates on which undefined query parameter the blob should be inserted. The SQL syntax to specify undefined parameter is database manager dependent. See the following examples to see how to proceed.

Example: SQLite and ODBC

```
1  ; SQLite and ODBC shares the same syntax to insert blob. It is
     indicated by the '?' character
2  ;
3  ; The database should be opened and a table PHOTOS with 3 column
     (BLOB, VARCHAR(255), BLOB)
4  ;
5  SetDatabaseBlob(0, 0, ?Picture, PictureLength)
6  SetDatabaseBlob(0, 1, ?SmallPicture, SmallPictureLength)
7  DatabaseUpdate(0, "INSERT INTO PHOTOS (picture, name, small_picture)
     values (?, 'my description', ?);")
```

Example: PostgreSQL

```
1  ; PostgreSQL uses another syntax: $1, $2.. into the statement to
     indicate the undefined parameter
2  ;
3  ; The database should be opened and a table PHOTOS with 3 column
     (BYTEA, VARCHAR(255), BYTEA)
4  ;
5  SetDatabaseBlob(0, 0, ?Picture, PictureLength)
6  SetDatabaseBlob(0, 1, ?SmallPicture, SmallPictureLength)
7  DatabaseUpdate(0, "INSERT INTO PHOTOS (picture, name, small_picture)
     values ($1, 'my description', $2);")
```

Note: PostgreSQL uses BYTEA to store large objects. The escaping needed to store binary data into such a column make it often bigger than expected. A good way to store binary data is to encode it with Base64Encoder() before submitting to the database manager.

73.28 UsePostgreSQLDatabase

Syntax

```
Result = UsePostgreSQLDatabase()
```

Description

Initialize the PostgreSQL database environment for future use. PostgreSQL is a powerful, server based database manager which support very large database and high concurrency. It is really free to use in commercial project, unlike MySQL which requiers a licence to use it in a non-GPL program. There is no additional driver to install, all is ready to connect a PostgreSQL server. For more information about PostgreSQL: http://www.postgresql.org.

A PostgreSQL database has to be connected using OpenDatabase() before using any other database functions. PostgresSQL specific parameters can be passed in the 'DatabaseName$' parameter of OpenDatabase() :

```
- host: Name of host to connect to.
- hostaddr: Numeric IP address of host to connect to.
- port: Port number to connect to at the server host.
- dbname: The database name. Defaults to be the same as the user name.
- connect_timeout: Maximum wait for connection, in seconds (write as
 a decimal integer string).
                   Zero or not specified means wait indefinitely.
                   It is not recommended to use a timeout of less
 than 2 seconds.
```

Example:

```
UsePostgreSQLDatabase()

; You should have a server running on localhost
;
If OpenDatabase(0, "host=localhost port=5432", "postgres", "postgres")
  Debug "Connected to PostgreSQL"
Else
  Debug "Connection failed: "+DatabaseError()
EndIf
```

73.29 UseSQLiteDatabase

Syntax

```
Result = UseSQLiteDatabase()
```

Description

Initialize the SQLite database environment for future use. SQLite is a file based, serverless database manager. There is no driver or additional files to install, all is ready to use. SQLite is widely spread

accross the industry and is considered to be one of the best embedded database manager available. For more information about SQLite: http://www.sqlite.org.

To create a new empty database, create a new file with CreateFile() . Database commands can now be used to create tables and add records.

A SQLite database has to be opened using OpenDatabase() before using any other database functions.

Example:

```
UseSQLiteDatabase()

Filename$ = OpenFileRequester("Choose a file name",
 "PureBasic.sqlite", "*.sqlite|*.sqlite", 0)

If CreateFile(0, Filename$)
  Debug "Database file created"
  CloseFile(0)
EndIf

If OpenDatabase(0, Filename$, "", "")
  Debug "Connected to PureBasic.sqlite"
  If DatabaseUpdate(0, "CREATE TABLE info (test VARCHAR(255));")
    Debug "Table created"
  EndIf
EndIf
```

73.30 UseODBCDatabase

Syntax

```
Result = UseODBCDatabase()
```

Description

Initialize the ODBC database environment for future use. It attempts to load the ODBC driver and allocate the required resources. If Result is 0, then the ODBC driver is not available or is too old (ODBC 3.0 or higher is needed) and the Database functions should not be used.

After calling UseODBCDatabase(), a database has to be opened using OpenDatabase() or OpenDatabaseRequester() before using any other Database functions.

It is possible to obtain a list of available drivers by calling the function ExamineDatabaseDrivers() . ; ; Date library documentation ; ; (c) 2010 - Fantaisie Software ;

Chapter 74

Date

The Date library allows for the manipulation of Date and Time from 1970 up to 2038 using the unix method (i.e. the number of seconds elapsed since the 1st of January 1970).

74.1 AddDate

Syntax

```
Date = AddDate(Date, Field, Value)
```

Description

Add a value to a date. The 'Field' can be one of the following constants:

```
#PB_Date_Year   : Will add 'Value' Year to the date
#PB_Date_Month  : Will add 'Value' Month to the date
#PB_Date_Week   : Will add 'Value' Week to the date
#PB_Date_Day    : Will add 'Value' Day to the date
#PB_Date_Hour   : Will add 'Value' Hour to the date
#PB_Date_Minute : Will add 'Value' Minute to the date
#PB_Date_Second : Will add 'Value' Second to the date
```

Note: when #PB_Date_Month is used, it will automatically account for the fact that the numbers of days per month varies, for example: if a month is added to '31 march 2008' the result will be '30 april 2008', since april does not have 31 days.

Example:

```

Debug FormatDate("%yyyy/%mm/%dd", AddDate(Date(), #PB_Date_Year, 2))
 ; Returns the current date + 2 years

Debug FormatDate("%mm/%dd/%yyyy", AddDate(Date(), #PB_Date_Year, 2))
 ; Returns the current date + 2 years
```

74.2 Date

Syntax

```
Date = Date([Year, Month, Day, Hour, Minute, Second])
```

Description

Returns the current system date value (number of seconds elapsed since 01/01/1970 0:00:00 up to the current time). If Year, Month, Day, Hour, Minute and Second parameters are specified, it returns the date representation in seconds. FormatDate() can be used to transform the date value into a user-readable format.

Example:

```
Debug Date() / (3600*24*365)  ; will print the number of years since 01/01/1970 and now
Debug Date(1999, 12, 31, 23, 59, 59)  ; will print '946684799' (number of seconds between 01/01/1970 0:00:00 and 12/31/1999 23:59:59)
```

Note: supported date/time values are 1970-01-01, 00:00:00 for the minimum and 2038-01-19, 03:14:07 for the maximum.

74.3 Day

Syntax

```
Result = Day(Date)
```

Description

Returns the day value of the specified date. The result is always between 1 and 31.

```
Debug Day(Date(2002, 10, 3, 0, 0, 0))  ; Outputs '3'.
```

74.4 DayOfWeek

Syntax

```
Result = DayOfWeek(Date)
```

Description

Returns the day value in the week of the specified date. The result is always between 0 and 6. Here is the day table for each value:

```
0 : Sunday
1 : Monday
```

```
2 : Tuesday
3 : Wednesday
4 : Thursday
5 : Friday
6 : Saturday
```

```
Debug DayOfWeek(Date(2006, 10, 30, 0, 0, 0))  ; Outputs '1' for Monday.
```

74.5 DayOfYear

Syntax

```
Result = DayOfYear(Date)
```

Description

Returns the number of days elapsed since the beginning of the year of the specified date. The result is always between 1 and 366.

```
Debug DayOfYear(Date(2002, 2, 1, 0, 0, 0))  ; Outputs '32'. (31 days for January + 1)
```

74.6 Month

Syntax

```
Result = Month(Date)
```

Description

Returns the month value of the specified date. The result is always between 1 and 12.

```
Debug Month(Date(2002, 10, 3, 0, 0, 0))  ; Outputs '10'.
```

74.7 Year

Syntax

```
Result = Year(Date)
```

Description

Returns the year value of the specified date. The result is always between 1970 and 2034.

```
Debug Year(Date(2002, 10, 3, 0, 0, 0))  ; Outputs '2002'.
```

74.8 Hour

Syntax

```
Result = Hour(Date)
```

Description

Returns the hour value of the specified date. The result is always between 0 and 23.

```
Debug Hour(Date(1970, 1, 1, 11, 3, 45))  ; Outputs '11'.
```

74.9 Minute

Syntax

```
Result = Minute(Date)
```

Description

Returns the minute value of the specified date. The result is always between 0 and 59.

```
Debug Minute(Date(1970, 1, 1, 11, 3, 45))  ; Outputs '3'.
```

74.10 Second

Syntax

```
Result = Second(Date)
```

Description

Returns the second value of the specified date. The result is always between 0 and 59.

```
Debug Second(Date(1970, 1, 1, 11, 3, 45))  ; Outputs '45'.
```

74.11 FormatDate

Syntax

```
Text\$ = FormatDate(Mask$, Date)
```

Description

Returns a string representation of the Date, according to the specified Mask$. The Mask$ parameter is a string which specifies how the date should be formatted. The Mask$ parameter accepts several reserved tokens:

```
%yyyy: Will be replaced by the year value, on 4 digits.
%yy: Will be replaced by the year value, on 2 digits.
%mm: Will be replaced by the month value, on 2 digits.
%dd: Will be replaced by the day value, on 2 digits.
%hh: Will be replaced by the hour value, on 2 digits.
%ii: Will be replaced by the minute value, on 2 digits.
%ss: Will be replaced by the second value, on 2 digits.
```

Example:

```
Debug FormatDate("Y=%yyyy, M= %mm, D=%dd", Date()) ; Will display the actual date with
                                                   ; the form "Y=2010, M=01, D=07"

Debug FormatDate("%mm/%dd/%yyyy", Date()) ; Will display the actual date with
                                          ; the form "01/07/2010"

Debug FormatDate("%hh:%ii:%ss", Date()) ; Will display the time using the 00:00:00 format
```

Note: supported date/time values are 1970-01-01, 00:00:00 for minimum and 2038-01-19, 03:14:07 for maximum.

74.12 ParseDate

Syntax

```
Date = ParseDate(Mask$, String$)
```

Description

Transforms a string date into a regular date value which then can be used by other date functions. If the date is incorrect, then -1 is returned. This function is very useful to retrieve dates already saved (in a log file for example) and do some operations on them.
The Mask$ is a string which specifies how the date is formatted. The Mask$ accepts several reserved tokens:

```
%yyyy: Will be replaced by the year value, on 4 digits.
%yy: Will be replaced by the year value, on 2 digits.
%mm: Will be replaced by the month value, on 2 digits.
%dd: Will be replaced by the day value, on 2 digits.
%hh: Will be replaced by the hour value, on 2 digits.
%ii: Will be replaced by the minute value, on 2 digits.
%ss: Will be replaced by the second value, on 2 digits.
```

Example:

```
Debug ParseDate("%yy/%mm/%dd", "10/01/07")      ; Returns the date value of "10/01/07"
```

```
3
4   Debug ParseDate("%mm/%dd/%yyyy", "01/07/2010") ; Returns the date
     value of "01/07/2010"
```

Note: The 'String$' to be converted into a date value must contain at least one separator between each part of the date (e.g. a slash character).

Chapter 75

Debugger

The Debugger library provides functions for controlling the debugger , for example to empty the debug output window or to open the memory viewer with a specific memory area to display. All these debugger tools are described in the debugger tools chapter.

The functions in this library are only compiled into the executable if the debugger is enabled on compilation. If the debugger is disabled then the entire call to these functions will be ignored.

There are also a number of special keywords to control the debugger from code.

75.1 ShowDebugOutput

Syntax

```
ShowDebugOutput()
```

Description

Open the debug output window or bring it to the front if it is already open.

75.2 ClearDebugOutput

Syntax

```
ClearDebugOutput()
```

Description

Clear the content of the debug output window.

Example:

```
; Show 10 debug values only, not a continuous list
Repeat
  ClearDebugOutput() ;
  For i = 1 To 10
    Debug x
    x + 1
  Next i

  Delay(500)
ForEver
```

75.3 SaveDebugOutput

Syntax

```
SaveDebugOutput(Filename$)
```

Description

save the content of the debug output window to the given filename. There is no error if the file cannot be saved.

Example:

```
For i = 1 To 100
  Debug Random(i)
Next i
SaveDebugOutput("C:\log.txt")
```

75.4 ShowProfiler

Syntax

```
ShowProfiler()
```

Description

Open the profiler window or bring it to the front if it is already open.

75.5 ResetProfiler

Syntax

```
ResetProfiler()
```

Description

Reset the line counters for the profiler.

75.6 StartProfiler

Syntax

```
StartProfiler()
```

Description

Start the counting of executed lines by the profiler.

75.7 StopProfiler

Syntax

```
StopProfiler()
```

Description

Stop the counting of executed lines by the profiler.

75.8 ShowMemoryViewer

Syntax

```
ShowMemoryViewer([*Buffer, Length])
```

Description

Open the memory viewer window or bring it to the front if it is already open. If a *Buffer and Length are specified then this memory area will be displayed in the memory viewer.

Example:

```
*Memory = AllocateMemory(1000)
If *Memory
  RandomData(*Memory, 1000)        ; Fill memory with some data

  ShowMemoryViewer(*Memory, 1000) ; Open memory viewer
  CallDebugger                    ; Halt the program, so it does not end right away
EndIf
```

75.9 ShowLibraryViewer

Syntax

```
ShowLibraryViewer([Library$ [, #Object]])
```

Description

Open the library viewer window or bring it to the front if it is already open. If Library$ is specified then the viewer will show the objects of that library. If an #Object number is specified, then the viewer will display the specified object of that library.

Example:

```
If CreateImage(0, 200, 200) And StartDrawing(ImageOutput(0))
  DrawingMode(#PB_2DDrawing_Transparent)
  Box(0, 0, 200, 200, RGB(255, 255, 255))
  For i = 1 To 30
    DrawText(Random(200), Random(200), "Hello World!",
 RGB(Random(255), Random(255), Random(255)))
  Next i
  StopDrawing()

  ShowLibraryViewer("Image", 0)  ; Show the image
  CallDebugger                   ; Halt the program so it does not end
 right away
EndIf
```

75.10 ShowWatchlist

Syntax

```
ShowWatchlist()
```

Description

Open the watchlist window or bring it to the front if it is already open.

75.11 ShowVariableViewer

Syntax

```
ShowVariableViewer()
```

Description

Open the variable viewer window or bring it to the front if it is already open.

75.12 ShowCallstack

Syntax

```
ShowCallstack()
```

Description

Open the callstack window or bring it to the front if it is already open.

75.13 ShowAssemblyViewer

Syntax

```
ShowAssemblyViewer()
```

Description

Open the assembly viewer window or bring it to the front if it is already open.

75.14 PurifierGranularity

Syntax

```
PurifierGranularity(GlobalGranularity, LocalGranularity,
    StringGranularity, DynamicGranularity)
```

Description

Change the intervall in which the purifier checks the different areas for memory corruption. Each value specifies the number of source lines that are executed between each check. Using #PB_Ignore will keep the existing intervall value. A value of 0 disables the check.

Example:

```
; Disable check for string memory and check allocated memory every 10
 lines
PurifierGranularity(#PB_Ignore, #PB_Ignore, 0, 10)
```

Chapter 76

Desktop

The desktop library allows access to information about the user's desktop environment, such as screen width, height, depth, mouse position etc.

76.1 ExamineDesktops

Syntax

```
Result = ExamineDesktops()
```

Description

Retrieves information about all the desktops connected to the local computer. This function must be called before using the following functions: DesktopDepth() , DesktopFrequency() , DesktopHeight() , DesktopName() and DesktopWidth() . If successful, this function returns the number of desktops, otherwise it returns 0. Typically, a standard user has only one desktop, but multi-screen users may have several desktops, each with their own resolution.

```
1  MessageRequester("Desktop Information", "You have
    "+Str(ExamineDesktops())+" desktops")
```

76.2 DesktopDepth

Syntax

```
Result = DesktopDepth(#Desktop)
```

Description

Returns the depth (in bits-per-pixel: 1, 2, 4, 8, 15, 16, 24 or 32 bits) of the specified desktop. The primary desktop is always specified using 0 as the parameter. ExamineDesktops() must be called before using this function to retrieve information about any of the available desktops.

```
ExamineDesktops()
MessageRequester("Display Information", "Current resolution =
 "+Str(DesktopWidth(0))+"x"+Str(DesktopHeight(0))+"x"+Str(DesktopDepth(0)))
```

76.3 DesktopFrequency

Syntax

```
Result = DesktopFrequency(#Desktop)
```

Description

Returns the frequency (in Hertz) of the specified desktop. The primary desktop is always specified using 0 as the parameter. ExamineDesktops() must be called before using this function to retrieve information about any of the available desktops. If 'Result' returns 0, then the default hardware frequency is being used (i.e. the user did not change it from the default setting).

```
ExamineDesktops()
f = DesktopFrequency(0)
If f = 0
  MessageRequester("Display Information", "There isn't set any
 desktop frequency, the standard hardware frequency is used.")
Else
  MessageRequester("Display Information", "Frequency of desktop:
 "+Str(f)+" Hz.")
EndIf
```

76.4 DesktopHeight

Syntax

```
Result = DesktopHeight(#Desktop)
```

Description

Returns the height (in pixels) of the specified desktop. The primary desktop is always specified using 0 as the parameter. ExamineDesktops() must be called before using this function to retrieve information about any of the available desktops.

```
ExamineDesktops()
MessageRequester("Display Information", "Current resolution =
 "+Str(DesktopWidth(0))+"x"+Str(DesktopHeight(0))+"x"+Str(DesktopDepth(0)))
```

76.5 DesktopX

Syntax

```
Result = DesktopX(#Desktop)
```

Description

Returns the x coordinate (in pixels) of the top left corner of the specified desktop. The primary desktop is always specified using 0 as the parameter. ExamineDesktops() must be called before using this function to retrieve information about any of the available desktops.

76.6 DesktopY

Syntax

```
Result = DesktopY(#Desktop)
```

Description

Returns the y coordinate (in pixels) of the top left corner of the specified desktop. The primary desktop is always specified using 0 as the parameter. ExamineDesktops() must be called before using this function to retrieve information about any of the available desktops.

76.7 DesktopMouseX

Syntax

```
Result = DesktopMouseX()
```

Description

Returns the absolute x position of the mouse on the desktop.

```
If OpenWindow(0, 0, 0, 300, 30, "Desktop mouse monitor", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
 TextGadget(0, 10, 6, 200, 20, "")

 Repeat
   Event = WindowEvent()

   If Event = 0 ; No events are in queue anymore, so halt the process for a few milliseconds for multitasking
     SetGadgetText(0, "Desktop mouse position: "+Str(DesktopMouseX())+","+Str(DesktopMouseY()))
     Delay(20)
   EndIf

 Until Event = #PB_Event_CloseWindow
EndIf
```

76.8 DesktopMouseY

Syntax

```
Result = DesktopMouseY()
```

Description

Returns the absolute y position of the mouse on the desktop.

```
If OpenWindow(0, 0, 0, 300, 30, "Desktop mouse monitor", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  TextGadget(0, 10, 6, 200, 20, "")

  Repeat
    Event = WindowEvent()

    If Event = 0 ; No events are in queue anymore, so halt the process for a few milliseconds for multitasking
      SetGadgetText(0, "Desktop mouse position: "+Str(DesktopMouseX())+","+Str(DesktopMouseY()))
      Delay(20)
    EndIf

  Until Event = #PB_Event_CloseWindow
EndIf
```

76.9 DesktopName

Syntax

```
Result\$ = DesktopName(#Desktop)
```

Description

Returns the name (if any) for the specified desktop. The primary desktop is always specified using 0 as the parameter. ExamineDesktops() must be called before using this function to retrieve information about any of the available desktops.

```
ExamineDesktops()
MessageRequester("Display Information", "Primary desktop name = "+DesktopName(0))
```

76.10 DesktopWidth

Syntax

```
Result = DesktopWidth(#Desktop)
```

Description

Returns the width (in pixels) for the specified desktop. The primary desktop is always specified using 0 as the parameter. ExamineDesktops() must be called before using this function to retrieve information about any of the available desktops.

```
ExamineDesktops()
MessageRequester("Display Information", "Current resolution = "+Str(DesktopWidth(0))+"x"+Str(DesktopHeight(0))+"x"+Str(DesktopDepth(0)))
```

Chapter 77

DragDrop

Drag & Drop is a widely used technology today as it provides an easy and very intuitive way to move data around between windows and applications. This library provides crossplatform functions to add this functionality to gadgets and windows with just a few extra lines of code.

Furthermore this library provides some functions to extend its default possibilities through platform specific APIs, giving the experienced programmer all the freedom to work with custom formats while still taking advantage of the framework provided by this library.

77.1 DragText

Syntax

```
Result = DragText(Text$ [, Actions])
```

Description

Starts a Drag & Drop operation with text data. Drag & Drop can basically be started any time, but the left mouse button should be currently pressed as otherwise the operation will end immediately without success. The usual time to start a Drag & Drop operation is when a Gadget reported an event with EventType() of #PB_EventType_DragStart.

Text$ specifies the text to drag.

Actions can be a combination of the Drag & Drop actions that should be allowed for the data. If the parameter is not specified, #PB_Drag_Copy will be the only allowed action. Possible actions are: (they can be combined with '||')

```
#PB_Drag_Copy: The text can be copied
#PB_Drag_Move: The text can be moved
#PB_Drag_Link: The text can be linked
```

The user can decide which of these actions to take by pressing modifier keys like Ctrl or Shift. The actions that can really be taken also depend on the actions allowed by the drop target. (On MacOSX, the actions are only treated as a suggestion. The drop target can still choose another action.)

Returns one of the above Drag & Drop action values to indicate what action the user took, or #PB_Drag_None if the user aborted the Drag & Drop operation.

Note that if #PB_Drag_Move is returned, it is your responsibility to remove the dragged text data from your application.

77.2 DragImage

Syntax

```
Result = DragImage(ImageID [, Actions])
```

Description

Starts a Drag & Drop operation with image data. Drag & Drop can basically be started any time, but the left mouse button should be currently pressed as otherwise the operation will end immediately without success. The usual time to start a Drag & Drop operation is when a Gadget reported an event with EventType() of #PB_EventType_DragStart.

ImageID specifies the image to drag. ImageID() can be used to get this ID for an image.

Actions can be a combination of the Drag & Drop actions that should be allowed for the image. If the parameter is not specified, #PB_Drag_Copy will be the only allowed action. Possible actions are: (they can be combined with '||')

```
#PB_Drag_Copy: The image can be copied
#PB_Drag_Move: The image can be moved
#PB_Drag_Link: The image can be linked
```

The user can decide which of these actions to take by pressing modifier keys like Ctrl or Shift. The actions that can really be taken also depend on the actions allowed by the drop target. (On MacOSX, the actions are only treated as a suggestion. The drop target can still choose another action.)

Returns one of the above Drag & Drop action values to indicate what action the user took, or #PB_Drag_None if the user aborted the Drag & Drop operation.

Note that if #PB_Drag_Move is returned, it is your responsibility to remove the dragged image from your application.

Example:

```
; Drag the image to an application that can accept images like an office or graphic program.
;
If LoadImage(1, #PB_Compiler_Home + "examples/sources/data/PureBasicLogo.bmp")
```

```
    If OpenWindow(1, 200, 200, 400, 90, "Drag & Drop", #PB_Window_SystemMenu)
      ImageGadget(1, 10, 10, 380, 70, ImageID(1))

      Repeat
        Event = WaitWindowEvent()
        If Event = #PB_Event_Gadget And EventGadget() = 1 And EventType() = #PB_EventType_DragStart
          DragImage(ImageID(1))
        EndIf
      Until Event = #PB_Event_CloseWindow
    EndIf
  EndIf
```

77.3 DragFiles

Syntax

```
Result = DragFiles(Files$ [, Actions])
```

Description

Starts a Drag & Drop operation with a list of filenames. Drag & Drop can basically be started any time, but the left mouse button should be currently pressed as otherwise the operation will end immediately without success. The usual time to start a Drag & Drop operation is when a Gadget reported an event with EventType() of #PB_EventType_DragStart.

Files$ is the list of filenames or directories to drag. Multiple filenames should be separated with a Chr(10) (linefeed) character. Each filename must include its full path, as the target application will not know how to resolve relative names. The filenames must refer to existing files, so the target application can access them.

Actions can be a combination of the Drag & Drop actions that should be allowed for the files. If the parameter is not specified, #PB_Drag_Copy will be the only allowed action. Possible actions are: (they can be combined with '||')

```
#PB_Drag_Copy: The files can be copied
#PB_Drag_Move: The files can be moved
#PB_Drag_Link: The files can be linked
```

The user can decide which of these actions to take by pressing modifier keys like Ctrl or Shift. The actions that can really be taken also depend on the actions allowed by the drop target. (On MacOSX, the actions are only treated as a suggestion. The drop target can still choose another action.)

Returns one of the above Drag & Drop action values to indicate what action the user took, or #PB_Drag_None if the user aborted the Drag & Drop operation.

Note that unlike with the other functions that start Drag & Drop, nothing should be done when #PB_Drag_Move is returned. As the dragged data is only the filename and not the file itself, any

action that is taken on the file must be done by the drag target.

Example:

```
; Select some files or folders and drag them to another application
;
If OpenWindow(1, 200, 200, 400, 400, "Drag & Drop", #PB_Window_SystemMenu)
 ExplorerListGadget(1, 10, 10, 380, 380, "*", #PB_Explorer_MultiSelect)

 Repeat
   Event = WaitWindowEvent()

   If Event = #PB_Event_Gadget And EventGadget() = 1 And EventType() = #PB_EventType_DragStart
     Files$ = ""
     For i = 0 To CountGadgetItems(1)-1
       If GetGadgetItemState(1, i) & #PB_Explorer_Selected
         Files$ + GetGadgetText(1) + GetGadgetItemText(1, i) + Chr(10)
       EndIf
     Next i

     DragFiles(Files$)
   EndIf

 Until Event = #PB_Event_CloseWindow
EndIf
```

77.4 DragPrivate

Syntax

```
Result = DragPrivate(Type [, Actions])
```

Description

Starts a "private" Drag & Drop operation. Unlike the other functions that start Drag & Drop, this data can only be dropped inside the application (Data dragged with functions like DragText() or DragImage() can be accepted by other applications as well). This function should be used to add Drag & Drop functionality between Gadgets or Windows with data that would not be understood by other applications.

Drag & Drop can basically be started any time, but the left mouse button should be currently pressed as otherwise the operation will end immediately without success. The usual time to start a Drag & Drop operation is when a Gadget reported an event with EventType() of #PB_EventType_DragStart.

Type can be any long value that identifies the data to be dragged in the application. The same value must be specified for EnableGadgetDrop() or EnableWindowDrop() for those Gadget /Windows that should accept this data.

This way it can be exactly defined which private drag operation will be accepted by which Gadget/Window, which allows complex Drag & Drop schemes to be realized.

Actions can be a combination of the Drag & Drop actions that should be allowed for the data. If the parameter is not specified, #PB_Drag_Copy will be the only allowed action. Possible actions are: (they can be combined with '||')

```
#PB_Drag_Copy: The data can be copied
#PB_Drag_Move: The data can be moved
#PB_Drag_Link: The data can be linked
```

The user can decide which of these actions to take by pressing modifier keys like Ctrl or Shift. The actions that can really be taken also depend on the actions allowed by the drop target.

Returns one of the above Drag & Drop action values to indicate what action the user took, or #PB_Drag_None if the user aborted the Drag & Drop operation.

If the operation was not aborted, the event loop will receive a #PB_Event_WindowDrop or #PB_Event_GadgetDrop event of type #PB_Drop_Private.

77.5 DragOSFormats

Syntax

```
Result = DragOSFormats(Formats(), Count [, Actions])
```

Description

Starts a Drag & Drop operation with a list of custom data formats. The types of formats available and the way in which they are represented depends on the Operating system. This function offers the possibility to work with formats not natively supported by PureBasic, while still using the simple mechanisms provided by this library to carry out the Drag & Drop operation.

Drag & Drop can basically be started any time, but the left mouse button should be currently pressed as otherwise the operation will end immediately without success. The usual time to start a Drag & Drop operation is when a Gadget reported an event with EventType() of #PB_EventType_DragStart.

Formats() should be an array of DragDataFormat structures containing one or more formats to drag. 'Count' specifies the number of formats in the array.
The structure has the following form:

```
Structure DragDataFormat
  Format.l     ; The OS specific ID for the format to drag (see below for more information)
 *Buffer       ; The memory buffer containing the data in this format
  Size.l       ; The size of the data in the buffer
EndStructure
```

Windows:

On Windows, the 'Format' field specifies a CLIPBOARDFORMAT value. It can be any standard clipboard format (found in the Windows SDK), or a format registered with the RegisterClipboardFormat_() api.

Linux:

On Linux, the 'Format' field specifies a GdkAtom value. It can be created with the gdk function gdk_atom_intern_(). Widely understood atoms are the common mime types (ie "text/html" for html data). The atom can also be created with any string that is understood by the target application.

MacOSX:

On MacOSX, the 'Format' field specifies a clipboard scrap type. These are 4 letter numeric character constants, for example 'TEXT'. There are a number of predefined scrap types, but also custom values can be used if the target program understands them.

If multiple formats are dragged, the target will accept the first one it recognizes. So the format that provides the most information (i.e. represents the data most accurately) should be first in the array, with less descriptive but more common formats later. This way each application will get the best representation of the data that it understands.

Actions can be a combination of the Drag & Drop actions that should be allowed for the data. If the parameter is not specified, #PB_Drag_Copy will be the only allowed action. Possible actions are: (they can be combined with '||')

```
#PB_Drag_Copy: The data can be copied
#PB_Drag_Move: The data can be moved
#PB_Drag_Link: The data can be linked
```

The user can decide which of these actions to take by pressing modifier keys like Ctrl or Shift. The actions that can really be taken also depend on the actions allowed by the drop target. (On MacOSX, the actions are only treated as a suggestion. The drop target can still choose another action.)

Returns one of the above Drag & Drop action values to indicate what action the user took, or #PB_Drag_None if the user aborted the Drag & Drop operation.

Note that if #PB_Drag_Move is returned, it is your responsibility to remove the dragged data from your application.

77.6 EnableGadgetDrop

Syntax

```
EnableGadgetDrop(#Gadget, Format, Actions [, PrivateType])
```

Description

Enables a gadget to be a target for Drag & Drop operations of a specific format. When the user drags data of this format over the gadget, the cursor will indicate that the data can be dropped there.

If data was dropped on the gadget, the program will receive a #PB_Event_GadgetDrop event. EventGadget() will indicate the target gadget and the Event functions of this library can be used to get the dropped data.

#Gadget is the PureBasic gadget number for the gadget in question.

Format specifies the data format and can be one of the following values, or an OS specific ID for a custom format (see DragOSFormats() for more information).

```
#PB_Drop_Text   : Accept text on this gadget
#PB_Drop_Image  : Accept images on this gadget
#PB_Drop_Files  : Accept filenames on this gadget
#PB_Drop_Private: Accept a "private" Drag & Drop on this gadget
```

Actions can be a combination of the Drag & Drop actions that should be allowed for the data. The user can decide which of these actions to take by pressing modifier keys like Ctrl or Shift. The actions that can really be taken also depend on the actions allowed by the drag source. Possible actions are: (they can be combined with '||')

```
#PB_Drag_None: The data format will not be accepted on the gadget
#PB_Drag_Copy: The data can be copied
#PB_Drag_Move: The data can be moved
#PB_Drag_Link: The data can be linked
```

PrivateType is only needed when #PB_Drop_Private is specified as format. It specifies the type of private Drag & Drop to accept. See DragPrivate() for more information.

Note: Multiple formats can be allowed on the same gadget. If the drag source provides multiple formats that match the list of accepted formats, the one that was added last will be accepted. So the preferred format in which to receive data should be enabled last.

77.7 EnableWindowDrop

Syntax

```
EnableWindowDrop(#Window, Format, Actions [, PrivateType])
```

Description

Enables a window to be a target for Drag & Drop operations of a specific format. Only the area not covered by any gadgets will be the target area. When the user drags data of this format over the window, the cursor will indicate that the data can be dropped there.

If data was dropped on the window, the program will receive a #PB_Event_WindowDrop event. EventWindow() will indicate the target window and the Event functions of this library can be used to get the dropped data.

#Window is the PureBasic window number for the window in question.

Format specifies the data format and can be one of the following values, or an OS specific ID for a custom format (see DragOSFormats() for more information).

```
#PB_Drop_Text    : Accept text on this window
#PB_Drop_Image   : Accept images on this window
#PB_Drop_Files   : Accept filenames on this window
#PB_Drop_Private: Accept a "private" Drag & Drop on this window
```

Actions can be a combination of the Drag & Drop actions that should be allowed for the data. The user can decide which of these actions to take by pressing modifier keys like Ctrl or Shift. The actions that can really be taken also depend on the actions allowed by the drag source. Possible actions are: (they can be combined with '||')

```
#PB_Drag_None: The data format will not be accepted on the window
#PB_Drag_Copy: The data can be copied
#PB_Drag_Move: The data can be moved
#PB_Drag_Link: The data can be linked
```

PrivateType is only needed when #PB_Drop_Private is specified as format. It specifies the type of private Drag & Drop to accept. See DragPrivate() for more information.

Note: Multiple formats can be allowed on the same window. If the drag source provides multiple formats that match the list of accepted formats, the one that was added last will be accepted. So the preferred format in which to receive data should be enabled last.

77.8 EventDropAction

Syntax

```
Result = EventDropAction()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() , this function returns the action that should be taken with the data.

Returns one of the following values:

```
#PB_Drag_Copy: The data should be copied
#PB_Drag_Move: The data should be moved (The drag source is
 responsible for removing the original data)
#PB_Drag_Link: The data should be linked
```

77.9 EventDropType

Syntax

```
Result = EventDropType()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() , this function returns the format of the dropped data.

Returns one of the following values, or an OS specific ID for a custom format (see DragOSFormats() for more information)

```
  #PB_Drop_Text    : Text was dropped.        (use EventDropText()
  to retrieve it)
  #PB_Drop_Image   : An image was dropped.    (use EventDropImage()
 to retrieve it)
  #PB_Drop_Files   : Filenames were dropped. (use EventDropFiles()
 to retrieve them)
  #PB_Drop_Private: A "private" operation finished. (use
   EventDropPrivate()
 to know its type)
```

To handle OS specific formats, EventDropBuffer() and EventDropSize() can be used.

77.10 EventDropText

Syntax

```
Result\$ = EventDropText()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() with a format (can be get with EventDropType()) of #PB_Drop_Text, this function returns the text that was dropped.

77.11 EventDropImage

Syntax

```
Result = EventDropImage(#Image [, Depth])
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() with a format (can be get with EventDropType()) of #PB_Drop_Image, this function can be used to retrieve the dropped image.

#Image is the number for the new image to create. If #PB_Any is used, the new image number will be returned as 'Result'.

Depth specifies the color depth for the new image. If it is omitted or set to #PB_Image_DisplayFormat, the image format is taken from the desktop format. Valid 'Depth' values can be: 1, 2, 4, 8, 16, 24 and 32 bits. When manipulating true colors formats, and want save back the modifications without color information loss, it's advised to use the 24 or 32 bits values. If the images are only used for displaying on the screen, #PB_Image_DisplayFormat will provide faster performances.

77.12 EventDropFiles

Syntax

```
Result\$ = EventDropFiles()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() with a format (can be get with EventDropType()) of #PB_Drop_Files, this function returns the filenames that were dropped.

Returns a list of filenames or directories, all with full path. The individual names are separated by a Chr(10) (linefeed) character.

77.13 EventDropPrivate

Syntax

```
Result = EventDropPrivate()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() with a format (can be get with EventDropType()) of #PB_Drop_Private, this function returns the 'PrivateType' that was dropped. (See DragPrivate() for more information.)

77.14 EventDropBuffer

Syntax

```
*Result = EventDropBuffer()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() with an OS specific format, this function can be used to access the data.

Returns the memory address where the dropped data is located.

Note: This memory buffer is internal to the library and should not be freed. It is also only valid until the next call to WaitWindowEvent() or WindowEvent() , so the data should be copied if it is required longer than that.

77.15 EventDropSize

Syntax

```
Result = EventDropSize()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() with an OS specific format, this function returns the size of the dropped data.

Returns the size in bytes of the memory buffer returned by EventDropBuffer() .

77.16 EventDropX

Syntax

```
Result = EventDropX()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() , this function returns the X position within the target gadget or window at which the data was dropped.

77.17 EventDropY

Syntax

```
Result = EventDropY()
```

Description

After a #PB_Event_GadgetDrop or #PB_Event_WindowDrop is received by WaitWindowEvent() or WindowEvent() , this function returns the Y position within the target gadget or window at which the data was dropped.

77.18 SetDragCallback

Syntax

```
SetDragCallback(@DragCallback())
```

Description

Specifies a callback function to be called during a Drag & Drop operation initiated from this application. The callback allows to modify the Drag & Drop process provided by PureBasic, for example by providing a custom cursor through the API of the Operating system. The form and function of the callback is dependant on the OS and described below:

Windows:

```
Procedure DragCallback(Action)

  ProcedureReturn #True
EndProcedure
```

The callback is called during the Drag & Drop operation. Action specifies the action that would be taken if the user released the mouse at this point. It can be one of these values:

```
#PB_Drag_None: The data will not be accepted if dropped here
#PB_Drag_Copy: The data will be copied
#PB_Drag_Move: The data will be moved
#PB_Drag_Link: The data will be linked
```

The callback can provide a custom cursor or drag image. If it does so, it should return #False. Returning #True will cause the default cursor to be used.

Linux:

```
Procedure DragCallback(*Context.GdkDragContext, isStart)

EndProcedure
```

The callback is only called at the start and the end of a Drag & Drop operation. The '*Context' parameter specifies the gdk drag context of this operation, 'isStart' specifies wether this is the start or the end of the operation. The returnvalue of the callback is ignored.

Gtk functions like gtk_drag_set_icon_pixbuf_() can be used in the callback to set a different drag image for the operation.

MacOSX:

```
Procedure DragCallback(DragReference, isStart)

EndProcedure
```

The callback is only called at the start and the end of a Drag & Drop operation. The 'DragReference' parameter specifies the Carbon drag context of this operation in form of a 'DragRef' value, 'isStart' specifies wether this is the start or the end of the operation. The returnvalue of the callback is ignored.

All functions of the Carbon Drag Manager can be used to influence the drag operation before it starts (adding more flavors/items for example) and to get more information about the operation after it completed.

77.19 SetDropCallback

Syntax

```
SetDropCallback(@DropCallback())
```

Description

Specifies a callback function to be called when data is dragged over a gadget or window that allows data to be dropped (see EnableGadgetDrop() / EnableWindowDrop()). The callback allows to modify the Drag & Drop process provided by PureBasic, for example by providing extra visual notification on the target gadget or window.

The callback is called as the mouse enters, moves and leaves the target gadget or window and allows to provide additional feedback to the user, for example by highlighting the target item or area. Furthermore the callback can deny the currently intended action and this way specify more detailed where within a gadget or window the data can be dropped. The cursor should not be modified here, as the drag source is responsible for this.

The form of the callback is described below:

```
Procedure DropCallback(TargetHandle, State, Format, Action,
 x, y)

  ProcedureReturn #True
EndProcedure
```

The first parameter specifies the OS specific handle for the target gadget or window. On Windows this is a HWND value, on Linux a GtkWidget pointer and on MacOSX it is a ControlRef or WindowRef value. These are the same values as returned by GadgetID() or WindowID() for the target gadget or window.

'State' specifies the current state of the Drag & Drop operation and is one of the following values:

```
#PB_Drag_Enter : The mouse entered the gadget or window
#PB_Drag_Update: The mouse was moved inside the gadget or
 window, or the intended action changed
#PB_Drag_Leave : The mouse left the gadget or window (Format,
 Action, x, y are 0 here)
#PB_Drag_Finish: The Drag & Drop finished
```

'Format' specifies the data format and can be one of the following values, or an OS specific ID for a custom format (see DragOSFormats() for more information).

```
#PB_Drop_Text    : Accept text on this gadget or window
#PB_Drop_Image   : Accept images on this gadget or window
#PB_Drop_Files   : Accept filenames on this gadget or window
#PB_Drop_Private: Accept a "private" Drag & Drop on this
 gadget or window
```

'Action' specifies the action that would be taken if the user released the mouse at this point. It can be one of these values:

```
#PB_Drag_None: The data will not be accepted if dropped here
#PB_Drag_Copy: The data will be copied
#PB_Drag_Move: The data will be moved
#PB_Drag_Link: The data will be linked
```

By returning #True, the callback allows the action to take place at this point. By returning #False, the callback denies the action (the cursor will be changed to a "forbidden" cursor by the drag source). Especially if 'State' is #PB_Drag_Finish, returning #False will cause the whole Drag & Drop operation to fail.

Chapter 78

Engine3D

PureBasic provides easy access to a very powerful OpenSource 3D Engine called OGRE. This choice has been made because we don't like to reinvent the wheel every time and when we saw the OGRE code quality, we felt that it would make a considerable contribution to the PureBasic 3D library.

OGRE is still under heavy development and will probably become a very advanced engine soon. Its community is growing more and more and many tools are available to make the most of this engine (e.g. Blender, Lightwave and 3DS Max exporters, Terrain manager etc). More information about OGRE is available at OGRE web site. The OGRE license file can be consulted here and PureBasic customized OGRE source files can be downloaded from here.

Note: If you use the 3D engine of PureBasic in your projects and you intent to distribute your executable, you will have to copy the Engine3D.dll from the PureBasic/Compilers directory to your main project directory.

78.1 Add3DArchive

Syntax

```
Add3DArchive(Path$, Type)
```

Description

Add a new absolute or relative path to the current 3D path list. All the 3D functions which need to load data (e.g. texture , mesh , sky , world) will use this path.

You may wonder why there is a special way to handle the files for 3D functions? Because it makes the data access more flexible and now, you can access the data using the standard file system or directly via the archive file like a .zip file. It's a very cool feature because you can pack all your data in one big compressed file, and then access the assets as if these were contained a standard directory.

'Type' can be one of the following value:

```
#PB_3DArchive_FileSystem : Standard directory
#PB_3DArchive_Zip        : Compressed zip file
```

Example:

```
Add3DArchive("MyData.zip", #PB_3DArchive_Zip) ; Add the .zip as a directory in the path
LoadTexture(0, "MyTexture.jpg")               ; load the MyTexture.jpg from the zip
LoadTexture(1, "World/Grass.jpg")             ; load the Grass.jpg from the zip, in the World\ path
```

78.2 AmbientColor

Syntax

```
AmbientColor(Color)
```

Description

Changes the ambient color of the world. RGB() can be used to get a valid Color value.

78.3 AntialiasingMode

Syntax

```
AntialiasingMode(Mode)
```

Description

Changes the fullscreen antialising mode. This function has to be called before OpenScreen() to have any effect. 'Mode' can be one of the following constants:

```
- #PB_AntialiasingMode_None: No antialiasing (default).
- #PB_AntialiasingMode_x2  : x2 fullscreen antialiasing
  (FSAA).
- #PB_AntialiasingMode_x4  : x4 fullscreen antialiasing
  (FSAA).
- #PB_AntialiasingMode_x6  : x6 fullscreen antialiasing
  (FSAA).
```

Note: depending of the graphic card, this command can have a severe impact on rendering performance.

78.4 CheckEntityCollision

Syntax

```
Result = CheckEntityCollision(#Entity, #Entity2)
```

Description

Checks if the #Entity has collided with #Entity2. If the 'Result' is 0, the entities have not collided, else they have collided. This function must be called after ExamineWorldCollisions() .

78.5 CountRenderedTriangles

Syntax

```
Result = CountRenderedTriangles()
```

Description

Returns the number of triangles rendered during the last frame.

78.6 Engine3DFrameRate

Syntax

```
Result = Engine3DFrameRate(Type)
```

Description

Returns the 3D engine frame rate (i.e. number of frames per second) depending of the specified 'Type'. 'Type' can be one of the following values:

```
#PB_Engine3D_Current : Current frame rate
#PB_Engine3D_Average : Average frame rate achieved since the
 engine started running
#PB_Engine3D_Maximum : Best frame rate achieved since the
 engine started running
#PB_Engine3D_Minimum : Worst frame rate achieved since the
 engine started running
#PB_Engine3D_Reset   : Reset the frame rate statistics
 (minimum, maximum and average)
```

78.7 EnableWorldCollisions

Syntax

```
EnableWorldCollisions(State)
```

Description

If 'State' is 0, the collisions are disabled in the world (this is the default behaviour), else the collisions are monitored and can examined later with ExamineWorldCollisions() . Collisions only work if the physics engine is enabled with EnableWorldPhysics() .

78.8 EnableWorldPhysics

Syntax

```
EnableWorldPhysics(State)
```

Description

Enable or disable the physics engine in the world. If 'State' is 0, the physics engine will be disabled (default), else the physics engine is applied to all the entities which have a body set with EntityPhysicBody() . Collisions can be enabled with ExamineWorldCollisions() .

78.9 ExamineWorldCollisions

Syntax

```
ExamineWorldCollisions()
```

Description

Examines which entities have collided since the last call using this function. To enumerate the collisions, use the NextWorldCollision() . Collisions are only available if the physics engine is enabled with EnableWorldPhysics() and collisions are activated with EnableWorldCollisions() . For a collision to occur, an entity must have a body associated with it using EntityPhysicBody() .

78.10 FirstWorldCollisionEntity

Syntax

```
Entity = FirstWorldCollisionEntity()
```

Description

Return the first #Entity number that has collided. This function must be called after NextWorldCollision()

78.11 Fog

Syntax

```
Fog(Color, Intensity, StartDistance, EndDistance)
```

Description

Creates a fog at the specified distance of the camera. If 'Intensity' is 0, the fog effect is removed. RGB() can be used to get a valid Color value. 'StartDistance' is the distance from the Camera (in World units) where the fog should start. 'EndDistance' is the distance from the Camera where the fog is fully opaque.

78.12 InitEngine3D

Syntax

```
Result = InitEngine3D()
```

Description

Initializes the 3D environment for later use. You must put this function at the top of your source code if you want to use any of the 3D functions. You should test the result to see if the 3D environment has been correctly initialized. If initialization fails, the program should quit or all 3D function calls should be disabled. This function tries to load the 'Engine3D.dll' and DirectX 7, so if it fails, it's probably because the DLL or DirectX 7+ is not found.

78.13 InputEvent3D

Syntax

```
InputEvent3D(MouseX, MouseY, LeftMouseButton, Text$)
```

Description

Injects events in the 3D GUI system. To be verstatile enough, the events aren't automatically gets from the main mouse and keyboard, but injected on demand. 'MouseX' and 'MouseY' specify the actual 3D GUI mouse position, in pixels. 'LeftMouseButton' is the left mouse button state (either 0 if unpressed or 1 if pressed). 'Text$' sets the text entered in the GUI system, for example to feed a StringGadget3D() when it has the focus.

Once these events are entered, all the 3D GUI state changes are handled automatically.

Example:

```
; Here is the classic case to get the events from the mouse and keyboard
;
InputEvent3D(MouseX(), MouseY(), MouseButton(#PB_MouseButton_Left), KeyboardInkey())
```

78.14 LoadWorld

Syntax

```
Result = LoadWorld(Filename$)
```

Description

This function loads an entire world. Currently, the Quake3 BSP format is the only one supported but more formats will follow. The Filename$ must be accessible in the 3D path, so the Add3DArchive() function should be used before calling this function. A world can be easily created using third party tools like 'Quark'. A world can contain a sky, building, lights and more. Actually, all Quake 3 maps can be loaded out of the box with full detail. If the 'Result' is 0, the world could not be loaded.

The BSP format is owned by iD Software and can only be used in freeware programs. Commercial software must acquire a license from iD to use this format. The license scheme is beyond PureBasic and Fantaisie Software can't be responsible for the incorrect use of this function.

78.15 MousePick

Syntax

```
Result = MousePick(#Camera, x, y)
```

Description

Simulates a mouse click and returns which object is under the specified 2D point (x,y - in pixels) on the specified #Camera. The returned value can be:

```
-1 : Nothing has been detected
#PB_World_WaterPick: the click occured on the water.
#PB_World_TerrainPick: the click occured on a terrain.
0 and above: the click occured on the #Entity. The entity pick is based on its bounding box, not on its mesh.
```

To get more information about the picked object position, use PickX() , PickY() and PickZ() .

78.16 PointPick

Syntax

```
Result = PointPick(#Camera, x, y)
```

Description

Allows to get the direction the specified 2D point (x,y - in pixels) on the specified #Camera. If the pick was not successful, it will returns 0. To get the point direction, use PickX() , PickY() and PickZ() .

78.17 PickX

Syntax

```
Result.f = PickX()
```

Description

After MousePick() , it returns the 'x' position of the picked object in world coordinates.
After PointPick() , it returns the 'x' direction of the picked point, between -1 and 1.

78.18 PickY

Syntax

```
Result.f = PickY()
```

Description

After MousePick() , it returns the 'y' position of the picked object in world coordinates.
After PointPick() , it returns the 'y' direction of the picked point, between -1 and 1.

78.19 PickZ

Syntax

```
Result.f = PickZ()
```

Description

After MousePick() , it returns the 'z' position of the picked object in world coordinates.
After PointPick() , it returns the 'z' direction of the picked point, between -1 and 1.

78.20 ShowGUI

Syntax

```
ShowGUI(Transparency, ShowMouse)
```

Description

Shows or hides the whole GUI elements, which are composed of 3d windows and 3d gadgets . A transparency level can be specified to fade the GUI at will (valid values are between 0 (hidden) and 255 (shown)). The mouse cursor can also be hidden/shown on demand.

78.21 SetGUITheme3D

Syntax

```
SetGUITheme3D(ThemeName$, FontName$)
```

Description

As CEGUI support skining, this command allow to select which theme and which font to use for the GUI. The 'ThemeName$' has to be specified without the '.scheme' extension and the 'FontName$' without the '.font' extension. This command has to be called before any other GUI commands to have an effect.

For more informations about skins, visit the CEGUI web site.

78.22 NextWorldCollision

Syntax

```
Result = NextWorldCollision()
```

Description

Checks if another collision is available. If the 'Result' is 0, no more collisions are available. This function should be called after ExamineWorldCollisions() . The following functions can be used to gather information about the collision: CheckEntityCollision() , FirstWorldCollisionEntity() and SecondWorldCollisionEntity() .

78.23 Parse3DScripts

Syntax

```
Parse3DScripts()
```

Description

Parses all the .materials OGRE scripts found in the paths set with Add3DArchive() . This allows the use of meshes with complex materials scripts directly in PureBasic. When creating the entity the constant #PB_Material_None has to be specified, so all the material information will be taken from the script, if it was correctly loaded.

More information about OGRE material scripts can be found at OGRE online manual.

78.24 RenderWorld

Syntax

```
RenderWorld()
```

Description

Renders the whole world on the screen. This function should be called once all 3D operations are finished and only one time per frame.

Once the RenderWorld() function has been performed, it's possible to use regular 2D functions like DisplaySprite() to display 2D sprites over the 3D world (note: Sprite3D is not supported though).

78.25 SecondWorldCollisionEntity

Syntax

```
Entity = SecondWorldCollisionEntity()
```

Description

Returns the second #Entity number which had collided. This function must be called after NextWorldCollision() .

78.26 SkyBox

Syntax

```
Result = SkyBox(TextureName$)
```

Description

Loads a 6 face cube and creates an artificial box which is far away from the camera but close to the world. This is a very useful function to easily make a closed world. If 'Result' is 0, the SkyBox could not be created. The textures must be in a path previously declared with the Add3DArchive() function. The TextureName$ is the base of the texture's names, which should be named using the following rule:

```
TextureName_BK ; BacK face
TextureName_FR ; FRont face
TextureName_DN ; DowN face
TextureName_UP ; UP face
TextureName_LF ; LeFt face
TextureName_RT ; RighT face
```

See also the SkyDome() function.

78.27 SkyDome

Syntax

```
Result = SkyDome(TextureName$, Curve)
```

Description

Creates a new SkyDome which is a curved moving sky displayed using the specified texture. The texture 'TextureName$' must be in a path previously declared with the Add3DArchive() function. The curve value indicates how much the sky should be curved (can be either negative or positive). If 'Result' is 0, the SkyDome could not be created.

See also the SkyBox() function.

78.28 CreateWater

Syntax

```
CreateWater(#Camera, x, y, z, Transparency, Flags)
```

Description

Creates a water plane at the given position. 'Transparency' adjust the whole water transparency, from 0 (fully opaque) to 255 (transparent). 'Flags' can be a combination of the following constants:

```
  - #PB_World_WaterMediumQuality: medium quality (default).
  - #PB_World_WaterLowQuality   : low quality, which means less
   polygons (but faster rendering).
  - #PB_World_WaterHighQuality  : high quality, which means
   more polygons (but slower rendering).
  - #PB_World_WaterCaustics     : Enable the water caustics
   effects, which are small patterns created by the light on
   the water surface.
  - #PB_World_WaterSmooth       : Enable smooth wave transition.
  - #PB_World_WaterFoam         : Enable foam effect, which
   affect underwater objects when viewed out of the water plane.
  - #PB_World_WaterSun          : Enable sun reflection on
   water. To control the sun position and color, use Sun()
.
```

78.29 Sun

Syntax

```
Sun(x, y, z, Color)
```

Description

Changes the sun position and color. This affects water created with the #PB_World_WaterSun support. RGB() can be used to get a valid color value.

78.30 WorldShadows

Syntax

```
Result = WorldShadows(Type)
```

Description

Sets how shadows will be rendered in the world. The 'Type' parameter can be one of the following values:

```
  #PB_Shadow_None       : No shadows will be displayed in the
   world. This can save a lot of CPU
                         power if shadows aren't needed
   (default)
  #PB_Shadow_Modulative: Modulative shadows will be displayed
   for entities
 that have the cast
                         shadow set with EntityRenderMode()
 and #PB_Entity_CastShadow.
                         This shadow mode is the fastest
   available, but are not very good-looking as the shadows are
   not translucent.
  #PB_Shadow_Additive  : Additive shadows will be displayed for
   the entities
```

```
which have the cast
                        shadow set with EntityRenderMode()
and #PB_Entity_CastShadow.
                        This mode is slower than the
  modulative mode, but it looks much better as the shadows are
  translucent.
                        Also if some shadows overlap, the
  shadows will be added resulting in a more realistic darker
  result.
 #PB_Shadow_TextureAdditive: Texture additive shadows will be
  displayed for the entities
which have the cast
                        shadow set with EntityRenderMode()
and #PB_Entity_CastShadow. This mode
                        is average in speed and quality
  between modulative and additive shadow, and will work even
  with water.
```

78.31 WorldGravity

Syntax

```
Result = WorldGravity(Gravity)
```

Description

Changes the gravity of the world when the physics engine is enabled with EnableWorldPhysics() . The default gravity is set to -9.806 (which is Earth's gravitational value).

78.32 WorldDebug

Syntax

```
Result = WorldDebug(Mode)
```

Description

Changes the world debug mode. 'Mode' can be a combination of the following constants:

```
  - #PB_World_DebugNone   : no debug information (default).
  - #PB_World_DebugEntity: shows the entities bounding boxes.
  - #PB_World_DebugBody   : shows the physics bodies, both
    static and dynamic.
```

This can be very useful to help finding issues with collisions or picking for example.

Chapter 79

Entity

Entities are objects composed of one mesh object and one material which can be freely moved and transformed in realtime.

It's possible to share a mesh or a material between several entities, reducing memory consumption and saving CPU clocks.

InitEngine3D() should be called successfully before using the entity functions.

79.1 AnimateEntity

Syntax

```
AnimateEntity(#Entity, Name$)
```

Description

Changes the current animation with the specified one, and start it. Entity uses the mesh animation (skeletons based, using bones) format.

Note: the "Name$" parameter is case-sensitive, which means than "Walk" will be different than "walk" for example.

79.2 CopyEntity

Syntax

```
Result = CopyEntity(#Entity, #NewEntity)
```

Description

Creates a #NewEntity which is the exact copy of the specified #Entity. If #PB_Any is used as '#NewEntity' parameter, the new entity number will be returned as 'Result'.

If the 'Result' is 0, the entity copy has failed. If #NewEntity was already created, it will be automatically freed and replaced by the new one.

79.3 CreateEntity

Syntax

```
Result = CreateEntity(#Entity, MeshID, MaterialID, [x, y, z])
```

Description

Creates a new #Entity using the specified Mesh and Material. If #PB_Any is used as '#Entity' parameter, the new entity number will be returned as 'Result'.

Valid 'MeshID' and 'MaterialID' can be easily obtained with the MeshID() and MaterialID() functions. #PB_Material_None can be used as 'MaterialID' value, to use the .material script associated to the mesh if the function Parse3DScripts() has been used.

The optionals x,y,z parameters can be specified to set the position of the newly created entity in the world.

If the 'Result' is 0, the #Entity creation has failed. If the #Entity was already created, it is automatically freed and replaced by the new one.

79.4 EntityID

Syntax

```
EntityID = EntityID(#Entity)
```

Description

Returns the unique ID which identifies the given '#Entity' in the operating system. This function is very useful when another library needs a entity reference.

79.5 EntityLocate

Syntax

```
EntityLocate(#Entity, x, y, z)
```

Description

Changes the #Entity absolute position in the world.

79.6 EntityLookAt

Syntax

```
EntityLookAt(#Entity, x, y, z)
```

Description

Changes the #Entity orientation in the world, to point towards the specified x,y,z point.

79.7 EntityMaterial

Syntax

```
EntityMaterial(#Entity, MaterialID)
```

Description

Assign a material to the specified #Entity. An entity can only have one material assigned at once.

'MaterialID' can be easily retrieved with the MaterialID() function.

79.8 EntityPhysicBody

Syntax

```
EntityPhysicBody(#Entity, Type [, Flags])
```

Description

Changes the type of the body associted with the #Entity.

To have its collisions managed by the physic engine, an entity has to set a body. In fact, only the body is known by the physic engine, which will do all the calculation about the entity, check the mass, friction and if it collides will move back the real entity.

To have any effect, the physic engine needs to be activated with the EnableWorldPhysics() . 'Type' can be one of the following constants:

```
  #PB_Entity_None        : No body is associated to the entity
   (default)
  #PB_Entity_StaticBody : The body is a static only, which
   means than the mesh can't be animated.
                          This mode allows very precise
   collisions, as it's done against triangles
                          (also known as tri-mesh collision).
   It's also fast when colliding with
                          a box or sphere entity body type.
   It's perfect when using a mesh for
                          a ground or static world.
  #PB_Entity_BoxBody     : A 'virtual' box is set around the
   entity (with the same dimensions)
                          and is used to manage the collision
   against the other entities.
  #PB_Entity_SphereBody : A 'virtual' sphere is set around the
   entity and is used to manage the collision
                          against the other entities.
```

'Flags' can be a combination of the following constants:

```
  - #PB_Entity_AbsoluteBodyMove: The body can be freely moved
   by MoveEntity()
 and EntityLocate()
.
```

79.9 EntityRenderMode

Syntax

```
EntityRenderMode(#Entity, Mode)
```

Description

Changes the render mode of the specified #Entity.

The 'Mode' parameter can be a combination (using the '||' operator) of the following values:

```
  #PB_Entity_CastShadow : The entity casts shadow if the
   WorldShadows()
 has been activated (default)
```

79.10 EntityX

Syntax

```
Result = EntityX(#Entity)
```

Description

Returns the 'x' position of the #Entity in the world.

79.11 EntityY

Syntax

```
Result = EntityY(#Entity)
```

Description

Returns the 'y' position of the #Entity in the world.

79.12 EntityZ

Syntax

```
Result = EntityZ(#Entity)
```

Description

Returns the 'z' position of the #Entity in the world.

79.13 FreeEntity

Syntax

```
FreeEntity(#Entity)
```

Description

Free the specified #Entity created with CreateEntity() before. All its associated memory is released and this object can't be used anymore.

Note: All remaining entities are automatically freed when the program ends.

79.14 GetEntityAnimationTime

Syntax

```
Time.f = GetEntityAnimationTime(#Entity)
```

Description

Returns the time position in the specified #Entity animation.

The time can be changed manually by using SetEntityAnimationTime() .

79.15 GetEntityFriction

Syntax

```
Friction.f = GetEntityFriction(#Entity)
```

Description

Returns the friction value of the specified #Entity.

To have any effect, the physic engine needs to be activated with the EnableWorldPhysics() .

The function SetEntityMass() will also affect the entity friction.

79.16 GetEntityMass

Syntax

```
Mass.f = GetEntityMass(#Entity)
```

Description

Returns the mass of the specified #Entity. To have any effect, the physic engine needs to be activated with the EnableWorldPhysics() . The function WorldGravity() can be used to affect the mass behaviour in the world.

79.17 EntityAnimationLength

Syntax

```
Length.f = EntityAnimationLength(#Entity)
```

Description

Returns the length of the current entity animation.

79.18 HideEntity

Syntax

```
HideEntity(#Entity, State)
```

Description

Hides or shows the specified #Entity.

'State' can take the following values:

```
1: the #Entity is hidden
0: the #Entity is shown
```

79.19 IsEntity

Syntax

```
Result = IsEntity(#Entity)
```

Description

Tests if the given #Entity is a valid and correctly initialized entity.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that an entity is ready to use.

79.20 MoveEntity

Syntax

```
MoveEntity(#Entity, x, y, z)
```

Description

Moves the #Entity in the world according to the specified x,y,z values.

This is a relative move based on the previous entity location. To perform an absolute move in the world, use EntityLocate() .

79.21 ResizeEntity

Syntax

```
ResizeEntity(#Entity, x, y, z)
```

Description

Resizes the #Entity using the specified x,y,z dimensions.

79.22 RotateEntity

Syntax

```
RotateEntity(#Entity, x, y, z [, Mode])
```

Description

Rotates the #Entity according to the specified x,y,z angle values. All angles are in degree, with values ranging from 0 to 360. The optional 'Mode' can be one of the following value:

```
PB_Absolute: absolute rotation (default).
PB_Relative: relative rotation based on the previous entity
 rotation.
```

79.23 ScaleEntity

Syntax

```
ScaleEntity(#Entity, x, y, z)
```

Description

Scales the #Entity according to the specified x,y,z values.

This is a factor based scale which means than the Entity size will be multiply with the given value to obtain the new size. To perform an absolute size of the entity, use ResizeEntity() .

Example:

```
1  ScaleEntity(0, 2, 2, 2) ; Double the size of the entity
2  ScaleEntity(0, 1, 1, 1) ; Don't change the size of the entity
     (multiply by 1 don't change anything
3  ScaleEntity(0, 3, 1, 1) ; Make the width of the entity 3
     times larger
```

79.24 SetEntityAnimationTime

Syntax

```
SetEntityAnimationTime(#Entity, Time)
```

Description

Changes the time position in the specified #Entity animation.

The time position can be retrieved by using GetEntityAnimationTime() .

79.25 SetEntityMass

Syntax

```
SetEntityMass(#Entity, Mass)
```

Description

Changes the mass of the specified #Entity.

To have any effect, the physic engine needs to be activated with the EnableWorldPhysics() .

The function WorldGravity() can be used to affect the mass behaviour in the world.

79.26 SetEntityFriction

Syntax

```
SetEntityFriction(#Entity, Friction)
```

Description

Changes the friction of the specified #Entity. It affect how the entity will react when colliding with other entities. If the friction is set to an high value, it willn't slide a lot.

To have any effect, the physic engine needs to be activated with the EnableWorldPhysics() .

The function SetEntityMass() will also affect the entity friction.

Chapter 80

File

Files are the main method for storage on computers. PureBasic allows the programmer to create applications in such a manner that the methods used to manage these files, are simple to use, and yet still optimized. Any number of files may be handled at the same time. This library uses buffered functions to increase the reading/writing speed. All the file functions can handle huge files, all the way up to: 2 ^64 bytes, (i.e. if the filesystem supports it).

For large amounts of data it may be useful to load the data into an array or a linked list , using a memory block may also be a good idea.

To get valid file paths for reading/saving data, take a look at the FileSystem and the Requester libraries.

80.1 CloseFile

Syntax

```
CloseFile(#File)
```

Description

Close the specified #File. Once the file is closed, it may not be used anymore. Closing a file ensures the buffer will effectively be put to the disk.

Note: all remaining opened files are automatically closed when the program ends.

For an example see the ReadFile() or the CreateFile() functions.

80.2 CreateFile

Syntax

```
Result = CreateFile(#File, Filename$)
```

Description

Create an empty file. If the file already exists, it will be opened and replaced by blank content! Be careful.

If #PB_Any is used as '#File' parameter, the new file number will be returned as 'Result'.

If 'Result' is not 0, the file has been effectively created, else the creation has failed. The result must always be tested, since performing operations on a non-created file will cause a severe crash.

If the #File was already used, the previous file will be closed automatically.

Example:

```

If CreateFile(0, "Text.txt")          ; we create a new text file...
  For a=1 To 10
    WriteStringN(0, "Line "+Str(a))   ; we write 10 lines (each with 'end of line' character)
  Next
  For a=1 To 10
    WriteString(0, "String"+Str(a))   ; and now we add 10 more strings on the same line (because there is no 'end of line' character)
  Next
  CloseFile(0)                        ; close the previously opened file and store the written data this way
Else
  MessageRequester("Information","may not create the file!")
EndIf
```

80.3 Eof

Syntax

```
Result = Eof(#File)
```

Description

Eof stands for 'End Of File'. It will return a non-null value if the end of the specified file has been reached, else it returns 0.

For an example see the ReadFile() function.

80.4 FileBuffersSize

Syntax

```
FileBuffersSize(#File, Size)
```

Description

Change the size of the buffers for the specified file. This method is available for advanced users to allow the tuning of the file access performance. The default buffer size is 4096 bytes per file. If the 'Size' is specified as 0, then all caching is disabled and all write operations are immediately written to the disk.

If '#File' is set to #PB_Default, then the default buffer size value will be changed to the specified 'Size': all further calls to OpenFile() , CreateFile() and ReadFile() will be use this new value.

Note: For performance reasons, the buffer size should be kept large enough (1028 seems to be ok as a minimum). When buffers are used, the information is really written to the disk once the cache buffer is full or when the file is closed. The FlushFileBuffers() function forces the cache buffer to be written at the time the function is called.

80.5 FileID

Syntax

```
Result = FileID(#File)
```

Description

Returns the FileID of the #File.

80.6 FileSeek

Syntax

```
FileSeek(#File, NewPosition.q)
```

Description

Change the read/write pointer position in the #File. The parameter 'NewPosition' must be given in Bytes.

Example:

```

file$ = OpenFileRequester("Select a file","","Text (.txt)|*.txt|All files (*.*)|*.*",0)
If file$
  If ReadFile(0, file$)
    length = Lof(0)                        ; read length of file
    FileSeek(0, length - 10)               ; set the file pointer 10 chars from end of file
    Debug "Position: " + Str(Loc(0))       ; show actual file pointer position
    *MemoryID = AllocateMemory(10)         ; allocate the needed memory for 10 bytes
    If *MemoryID
      bytes = ReadData(0, *MemoryID, 10)   ; read this last 10 chars in the file
      Debug PeekS(*MemoryID)
    EndIf
    CloseFile(0)
  EndIf
EndIf
```

80.7 FlushFileBuffers

Syntax

```
FlushFileBuffers(#File)
```

Description

Ensures than all buffered operations are really written to disk. See FileBuffersSize() for more information about file buffer management.

80.8 IsFile

Syntax

```
Result = IsFile(#File)
```

Description

Tests if the given #File number is a valid and correctly initialized file. This function is bullet-proof and may be used with any value.

If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a file is ready to use.

80.9 Loc

Syntax

```
Position.q = Loc(#File)
```

Description

Returns the actual pointer position in the file.

For an example look at the FileSeek() function.

80.10 Lof

Syntax

```
Length.q = Lof(#File)
```

Description

Lof stands for 'Length Of File'. It will return the length of the #File.

Example:

```

file$ = OpenFileRequester("Select a file","","Text (.txt)|*.txt|All files (*.*)|*.*",0)
If file$
  If ReadFile(0, file$)
    length = Lof(0)                          ; get the length of opened file
    *MemoryID = AllocateMemory(length)       ; allocate the needed memory
    If *MemoryID
      bytes = ReadData(0, *MemoryID, length)   ; read all data into the memory block
      Debug "Number of bytes read: " + Str(bytes)
    EndIf
    CloseFile(0)
  EndIf
EndIf
```

80.11 OpenFile

Syntax

```
Result = OpenFile(#File, Filename$)
```

Description

It will open the designated file or create it if it does not exists. Read and write may be performed on this file. OpenFile() will be used in the same manner as the "read-only" ReadFile() and the "write-only" CreateFile() functions. When an existing file is opened using OpenFile(), the file pointer is positioned at the beginning of the file.

If #PB_Any is used as '#File' parameter, the new file number will be returned as 'Result'.

If 'Result' is not 0, the file has been effectively opened, else the file may not be opened. The result must be always tested, as performing operations on a non-opened file will cause a severe crash.

If the #File was already used, the previous file will be closed automatically.

If data should be added to an existing file, the file pointer must be set to the end of the file first. For this the commands FileSeek() and Lof() are used after OpenFile():

Example:

```

If OpenFile(0, "Test.txt")     ; opens an existing file or creates one, if it does not exist yet
  FileSeek(0, Lof(0))          ; jump to the end of the file (result of Lof() is used)
  WriteStringN(0, "... another line at the end.")
  CloseFile(0)
EndIf
```

Note: when using OpenFile() the file must have the rights for reading AND writing. For example, this may be a problem when attempting to manipulate a file on a CD-ROM. To read from a file, the preferred method is to use the ReadFile() function instead.

80.12 TruncateFile

Syntax

```
TruncateFile(#File)
```

Description

Cuts the file at the current file position and discards all data that follows. This function may be used to shorten a file without the necessity of recreating it entirely. To make a file longer, simply append more data with the write commands of this library.

80.13 ReadAsciiCharacter

Syntax

```
Number.a = ReadAsciiCharacter(#File)
```

Description

Read an ascii character (1 byte) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadAsciiCharacter() only a ascii character is read, instead of a complete line (string).

80.14 ReadByte

Syntax

```
Number.b = ReadByte(#File)
```

Description

Read a byte (1 byte) from the #File.

For an example of how read from a file see the ReadFile() function - with ReadByte() only a byte value is read, instead of a complete line (string).

80.15 ReadCharacter

Syntax

```
Number.c = ReadCharacter(#File)
```

Description

Read a character (1 byte in ASCII, 2 bytes in unicode) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadCharacter() only a character value is read, instead of a complete line (string).

80.16 ReadDouble

Syntax

```
Number.d = ReadDouble(#File)
```

Description

Read a double (8 bytes) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadDouble() only a double value is read, instead of a complete line (string).

80.17 ReadFile

Syntax

```
Result = ReadFile(#File, Filename$)
```

Description

Open an existing file for read-only operations.

If #PB_Any is used as '#File' parameter, the new file number will be returned as 'Result'.

If 'Result' is not 0, the file has been effectively opened, else the file may not be opened. The result must be always tested, since performing operations on a non-opened file will cause a severe crash.

If the #File was already used, the previous file will be closed automatically.

Example:

```
1
2   If ReadFile(0, "Text.txt")    ; if the file could be read, we
     continue...
3    While Eof(0) = 0             ; loop as long the 'end of file'
     isn't reached
4       Debug ReadString(0)       ; display line by line in the
     debug window
5    Wend
6    CloseFile(0)                 ; close the previously opened
     file
7   Else
8     MessageRequester("Information","Couldn't open the file!")
9   EndIf
```

80.18 ReadFloat

Syntax

```
Number.f = ReadFloat(#File)
```

Description

Read a float (4 bytes) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadFloat() only a float value is read, instead of a complete line (string).

80.19 ReadInteger

Syntax

```
Number.i = ReadInteger(#File)
```

Description

Read an integer (4 bytes on 32 bits processors, or 8 bytes on 64 bits processors) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadInteger() only a integer value is read, instead of a complete line (string).

80.20 ReadLong

Syntax

```
Number.l = ReadLong(#File)
```

Description

Read a long (4 bytes) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadLong() only a long value is read, instead of a complete line (string).

80.21 ReadQuad

Syntax

```
Number.q = ReadQuad(#File)
```

Description

Read a quad (8 bytes) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadQuad() only a quad value is read, instead of a complete line (string).

80.22 ReadData

Syntax

```
Length = ReadData(#File, *MemoryBuffer, LengthToRead)
```

Description

Read the content from the #File to the specified memory buffer. 'LengthToRead' specifies the number of bytes to read.

After executing the function, the 'Length' variable will hold the number of bytes that were really read. This may be useful for validation purposes.

For a code example look at the Lof() function.

80.23 ReadString

Syntax

```
Text\$ = ReadString(#File [, Flags])
```

Description

Read a string from the #File until an 'End Of Line' character is found (Unix and DOS file formats are supported). Without any 'Flags', the string is reads as UTF8 if the program is compiled in unicode mode, else it's reads as ASCII.

The optional 'Flags' parameter may be one of the following values:

```
#PB_Ascii   : Reads the string as ASCII, even when the program
 is compiled in unicode mode
#PB_UTF8    : Reads the string as UTF8, even when the program
 is compiled in ASCII mode
#PB_Unicode: Reads the string as UTF16 (the program may be
 either in unicode or ASCII mode).
```

For an example see the ReadFile() function.

For detecting the string encoding format (byte order mark) used in a file there is the ReadStringFormat() function available.

80.24 ReadStringFormat

Syntax

```
Result = ReadStringFormat(#File)
```

Description

Checks if the current file position contains a BOM (Byte Order Mark) and tries to identify the String encoding used in the file. If a BOM is detected, the file pointer will be placed at the end of the BOM.

The Result may be one of the following:

```
#PB_Ascii   : No BOM detected. This usually means a plain text
 file.
#PB_UTF8    : UTF-8 BOM detected.
#PB_Unicode: UTF-16 (little endian) BOM detected.

#PB_UTF16BE: UTF-16 (big endian) BOM detected.
#PB_UTF32   : UTF-32 (little endian) BOM detected.
#PB_UTF32BE: UTF-32 (big endian) BOM detected.
```

The #PB_Ascii, #PB_UTF8 and #PB_Unicode results may be used directly in further calls to ReadString() to read the file. The other results represent string formats that cannot be directly read with PB string functions. They are included for completeness so that an application can display a proper error message.

The Byte Order Mark is a commonly used way to indicate the encoding for a textfile. It is usually placed at the beginning of the file. It is however not a standard, just a commonly used practice. So if no BOM is detected at the start of a file, it does not necessarily mean that it is a plain text file. It could also just mean that the program that created the file did not use this practice. WriteStringFormat() may be used to place such a BOM in a file.

For more information, see this Wikipedia Article.
More information about using unicode in a PureBasic program can also be found here .

80.25 ReadUnicodeCharacter

Syntax

```
Number.u = ReadUnicodeCharacter(#File)
```

Description

Read a unicode character (2 bytes) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadUnicodeCharacter() only a unicode character is read, instead of a complete line (string).

80.26 ReadWord

Syntax

```
Number.w = ReadWord(#File)
```

Description

Read a word (2 bytes) from the #File.

For an example of how to read from a file, see the ReadFile() function - with ReadWord() only a word value is read, instead of a complete line (string).

80.27 WriteAsciiCharacter

Syntax

```
WriteAsciiCharacter(#File, Number)
```

Description

Write an ascii character (1 byte) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteAsciiCharacter() only an ascii character is written, instead of a string.

80.28 WriteByte

Syntax

```
WriteByte(#File, Number)
```

Description

Write a byte number (1 byte) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteByte() only a byte number is written, instead of a string.

80.29 WriteCharacter

Syntax

```
WriteCharacter(#File, Number)
```

Description

Write a character number (1 bytes in ASCII, 2 bytes in unicode) in the #File. The file must be opened a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteCharacter() only a character number is written, instead of a string.

80.30 WriteDouble

Syntax

```
WriteDouble(#File, Number)
```

Description

Write a double number (8 bytes) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteDouble() only a double number is written, instead of a string.

80.31 WriteFloat

Syntax

```
WriteFloat(#File, Number)
```

Description

Write a float number (4 bytes) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteFloat() only a float number is written, instead of a string.

80.32 WriteInteger

Syntax

```
WriteInteger(#File, Number)
```

Description

Write an integer number (4 bytes on 32 bits processors, or 8 bytes on 64 bits processors) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteInteger() only a integer variable is written, instead of a string.

80.33 WriteLong

Syntax

```
WriteLong(#File, Number)
```

Description

Write a long number (4 bytes) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteLong() only a long variable is written, instead of a string.

80.34 WriteData

Syntax

```
WriteData(#File, *MemoryBuffer, Length)
```

Description

Write the content of the specified memory buffer to the #File. 'Length' is the memory size (in bytes) to be written.

Example:

```

*MemoryID = AllocateMemory(1000)        ; allocating a memory block
If *MemoryID
  PokeS(*MemoryID, "Store this string in the memory area") ; we write a string into the memory block
EndIf
If CreateFile(0, "Text.txt")            ; we create a new text file...
  WriteData(0, *MemoryID, 10)           ; write the first 10 chars from the memory block into the file
  CloseFile(0)                          ; close the previously opened file and so store the written data
Else
  Debug "may not create the file!"
EndIf
```

80.35 WriteQuad

Syntax

```
WriteQuad(#File, Number)
```

Description

Write a quad number (8 bytes) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteQuad() only a quad number is written, instead of a string.

80.36 WriteString

Syntax

```
WriteString(#File, Text$ [, Flags])
```

Description

Write a string in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()). Without any 'Flags', the string is written in UTF8 format if the program is compiled in unicode mode, else it's written in ASCII format.

The optional 'Flags' parameter may be one of the following values:

```
#PB_Ascii   : Writes the string in ASCII format, even when the program is compiled in unicode mode
#PB_UTF8    : Writes the string in UTF8 format, even when the program is compiled in ASCII mode
#PB_Unicode: Writes the string in UTF16 format (the program may be either in unicode or ASCII mode).
```

For an example see the CreateFile() function.

For placing a BOM (byte order mark) to later identify the string encoding format of a file there is the WriteStringFormat() function available.

80.37 WriteStringFormat

Syntax

```
WriteStringFormat(#File, Format)
```

Description

Writes a BOM (Byte Order Mark) at the current position in the file. It is a commonly used practice to identify the string encoding in a file.

'Format' may be one of the following:

```
#PB_Ascii   : Writes no BOM at all (this is usually
 interpreted as an plain Ascii file.)
#PB_UTF8    : UTF-8 BOM
#PB_Unicode: UTF-16 (little endian) BOM

#PB_UTF16BE: UTF-16 (big endian) BOM
#PB_UTF32   : UTF-32 (little endian) BOM
#PB_UTF32BE: UTF-32 (big endian) BOM
```

The #PB_Ascii, #PB_UTF8 and #PB_Unicode correspond to the flags supported by WriteString() and WriteStringN() . After placing such a BOM, the strings which follow should all be written with this flag. The other formats represent string formats that may not be written directly with the PB string functions. They are included only for completeness,

The Byte Order Mark is a commonly used method with which to indicate the encoding of a textfile. It is usually placed at the beginning of the file. It is however not a standard, just a commonly used practice. So if no BOM is detected at the start of a file, it does not necessarily mean that it is a plain text file. It could also just mean that the program that created the file did not use this practice. ReadStringFormat() may be used detect a BOM within a file.

For more information, see this Wikipedia Article.
More information about using unicode in PureBasic program can also be found here .

80.38 WriteStringN

Syntax

```
WriteStringN(#File, Text$ [, Flags])
```

Description

Write a string in the #File and add an 'end of line' character. The file must be opened using a write-capable function (i.e. not with ReadFile()). Without any 'Flags', the string is written in UTF8 format, if the program has been compiled using the unicode mode, else it is written in ASCII format.

The optional 'Flags' parameter may be one of the following values:

```
#PB_Ascii   : Writes the string in ASCII format, even when the
 program is compiled in unicode mode
#PB_UTF8    : Writes the string in UTF8 format, even when the
 program is compiled in ASCII mode
#PB_Unicode: Writes the string in UTF16 format (the program
 may be either in unicode or ASCII mode).
```

For an example see the CreateFile() function.

For placing a BOM (byte order mark) in order to later identify the string encoding format of a file there is the WriteStringFormat() function available.

80.39 WriteUnicodeCharacter

Syntax

```
WriteUnicodeCharacter(#File, Number)
```

Description

Write a unicode character (2 bytes) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteUnicodeCharacter() only a unicode character is written, instead of a string.

80.40 WriteWord

Syntax

```
WriteWord(#File, Number)
```

Description

Write a word number (2 bytes) in the #File. The file must be opened using a write-capable function (i.e. not with ReadFile()).

For an example see the CreateFile() function - with WriteWord() only a word variable is written, instead of a string.

Chapter 81

FileSystem

FileSystem is a generic term which deal with all advanced file related manipulations. For example, it's possible to read a directory content, create a new directory and more...

If you want to examine the contents of a directory start with the ExamineDirectory() function.

In addition to the functions in this library you will also functions for manipulating file contents in the File library. To get the name of a running program use the function ProgramFilename() from the Process library.

81.1 CopyDirectory

Syntax

```
Result = CopyDirectory(SourceDirectory$, DestinationDirectory$,
    Pattern$ [, Mode])
```

Description

Copy the contents of the source directory to the destination. Warning, if the directory already exists, it will be overwritten automatically. If the Result is 0, the directory can't be copied.

The 'Pattern$' should be a valid OS pattern. For example: "*.*" will copy any files in the directory. "*.exe" will copy only the .exe files. By default, a null Pattern$ ("") will copy all the files.

'Mode' is optional and can be a combination of the following values:

```
#PB_FileSystem_Recursive: Copy the directory with all
 subdirectories.
#PB_FileSystem_Force    : Overwrites the files which are
 protected (ReadOnly).
```

Example:

```
Debug CopyDirectory("D:\Games\MyGame\", "D:\Games\Backup\",
 "", #PB_FileSystem_Recursive)
```

81.2 CopyFile

Syntax

```
Result = CopyFile(SourceFilename$, DestinationFilename$)
```

Description

Copy the source file to the destination. Warning, if the file already exists, it will be erased automatically. If the Result is 0, the file can't be copied.

81.3 CreateDirectory

Syntax

```
Result = CreateDirectory(DirectoryName$)
```

Description

Creates a new directory. If Result is 0, the directory can't be created.

81.4 DeleteDirectory

Syntax

```
Result = DeleteDirectory(Directory$, Pattern$ [, Mode])
```

Description

Deletes the specified Directory$. Warning, it's a very dangerous function so use it with care. If the Result is 0, the directory hasn't be fully deleted.

The 'Pattern$' should be a valid OS pattern. For example: "*.*" will delete any files in the directory. "*.exe" will delete only the .exe files. By default, a null Pattern$ ("") will delete all the files.

'Mode' is optional and can be a combination of the following values:

```
#PB_FileSystem_Recursive: Deletes the directory with all
 subdirectories.
#PB_FileSystem_Force     : Deletes even the files which are
 protected (ReadOnly).
```

81.5 DeleteFile

Syntax

```
Result = DeleteFile(Filename$)
```

Description

Deletes the specified file. If the Result is 0, the file can't be deleted.

81.6 DirectoryEntryAttributes

Syntax

```
Attributes = DirectoryEntryAttributes(#Directory)
```

Description

Returns the attributes of the current entry in the #Directory being listed with ExamineDirectory() and NextDirectoryEntry() functions.

On Windows, Attributes is a combination of the following values:

```
#PB_FileSystem_Hidden     : File is hidden
#PB_FileSystem_Archive    : File has been archived and not
 changed since the last time
#PB_FileSystem_Compressed: File is compressed
#PB_FileSystem_Normal     : Normal attributes
#PB_FileSystem_ReadOnly   : File is in readonly mode
#PB_FileSystem_System     : File is a system file
```

On Linux or MacOSX, Attributes is a combination of these values:

```
#PB_FileSystem_Link       : The file is a symbolic link
#PB_FileSystem_ReadUser   : Access flags for the owning user
#PB_FileSystem_WriteUser
#PB_FileSystem_ExecUser
#PB_FileSystem_ReadGroup : Access flags for the owning user's
 group
#PB_FileSystem_WriteGroup
#PB_FileSystem_ExecGroup
#PB_FileSystem_ReadAll    : Access flags for all other users
#PB_FileSystem_WriteAll
#PB_FileSystem_ExecAll
```

To check if one attribute is actually set, just use the '&' (binary AND) and the attribute constant value:

```
[...]

FileAttributes = DirectoryEntryAttributes(#Directory)
If FileAttributes & #PB_FileSystem_Hidden
  Debug "This file is hidden !"
EndIf
```

81.7 DirectoryEntryDate

Syntax

```
Result = DirectoryEntryDate(#Directory, DateType)
```

Description

Returns the date of the current entry in the #Directory being listed with ExamineDirectory() and NextDirectoryEntry() functions.

The date is in the same format than the one of the Date library, so all the date related functions like FormatDate() can be used.

The 'DateType' parameter can be one of the following constants:

```
#PB_Date_Created : returns the file creation date.
#PB_Date_Accessed: returns the last file access date.
#PB_Date_Modified: returns the last file modification date.
```

81.8 DirectoryEntryName

Syntax

```
Filename\$ = DirectoryEntryName(#Directory)
```

Description

Returns the name of the current entry in the #Directory being listed with ExamineDirectory() and NextDirectoryEntry() functions.

81.9 DirectoryEntryType

Syntax

```
Result = DirectoryEntryType(#Directory)
```

Description

Returns the type of the current entry in the #Directory being listed with ExamineDirectory() and NextDirectoryEntry() functions.

The 'Result' can be one of the following constants:

```
#PB_DirectoryEntry_File      : This entry is a file.
#PB_DirectoryEntry_Directory: This entry is a directory.
```

81.10 DirectoryEntrySize

Syntax

```
Size.q = DirectoryEntrySize(#Directory)
```

Description

Returns the size of the current entry in the #Directory being listed with ExamineDirectory() and NextDirectoryEntry() functions.

81.11 ExamineDirectory

Syntax

```
Result = ExamineDirectory(#Directory, DirectoryName$, Pattern$)
```

Description

Start to examine the specified #Directory for future listing with functions NextDirectoryEntry() , DirectoryEntryName() and DirectoryEntryType() .

If #PB_Any is used as '#Directory' parameter, the new directory number will be returned as 'Result'. If the result is 0, the directory can't be examined.

Once the enumeration is done, FinishDirectory() must be called to free the resources associated to the listing.

The Pattern$ specify which files must be listed. For example: A 'Pattern$' like "*.*" or "" will list all the files (and sub-directories) in the directory. A 'Pattern$' like "*.exe" will list only .exe files (and sub-directories ending with .exe if any).

Note: when the program ends, all remaining ExamineDirectory() are finished automatically.

Example:

```
Directory$ = "C:\"   ; Lists all entries in the Windows directory (without the contents of sub-directories)
If ExamineDirectory(0, Directory$, "*.*")
  While NextDirectoryEntry(0)
    If DirectoryEntryType(0) = #PB_DirectoryEntry_File
      Type$ = " [File] "
    Else
      Type$ = " [Sub-Dir] "
    EndIf

    Debug DirectoryEntryName(0) + Type$ + "- Size in byte: " + Str(DirectoryEntrySize(0))
  Wend
  FinishDirectory(0)
EndIf
```

81.12 FinishDirectory

Syntax

```
FinishDirectory(#Directory)
```

Description

Finish the enumeration started with ExamineDirectory() . This allows to free the resources associated with the #Directory listing.

Note: when the program ends, all remaining ExamineDirectory() are automatically finished.

81.13 GetExtensionPart

Syntax

```
Path\$ = GetExtensionPart(FullPathName$)
```

Description

Retrieves the extension part of a full path. For example, if the full path is "C:\PureBasic\PB.exe", the result will be "exe".

To retrieve the file or the path part from a full path, look at the GetFilePart() and GetPathPart() functions.

More useful file-related functions can be found in the File and FileSystem libs.

81.14 GetFilePart

Syntax

```
Filename\$ = GetFilePart(FullPathName$)
```

Description

Retrieves the file part of a full path. For example, if the full path is "C:\PureBasic\PB.exe", the result will be "PB.exe".

To retrieve the extension or the path part from a full path, look at the GetExtensionPart() and GetPathPart() functions.

More useful file-related functions can be found in the File and FileSystem libs.

81.15 GetPathPart

Syntax

```
Path\$ = GetPathPart(FullPathName$)
```

Description

Retrieves the path part of a full path. For example, if the full path is "C:\PureBasic\PB.exe", the result will be "C:\PureBasic\".

To retrieve the extension or the file part from a full path, look at the GetExtensionPart() and GetFilePart() functions.

More useful file-related functions can be found in the File and FileSystem libs.

81.16 IsDirectory

Syntax

```
Result = IsDirectory(#Directory)
```

Description

Tests if the given #Directory number is a valid and correctly initialized directory. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a directory is ready to use.

81.17 CheckFilename

Syntax

```
Result = CheckFilename(Filename$)
```

Description

Checks if the specified Filename$ doesn't contain invalid characters for the filesystem. For example, on Windows '/' and '\' characters are not allowed in the filename. If the 'Filename$' is correct, it will returns 1, else 0. The 'Filename$' must not include its path.

81.18 FileSize

Syntax

```
Result.q = FileSize(Filename$)
```

Description

Returns the size of the specified file.

Special 'Result' values:

```
-1: File not found.
-2: File is a directory.
```

81.19 GetCurrentDirectory

Syntax

```
Result\$ = GetCurrentDirectory()
```

Description

Returns the current directory name. All files accesses are relative to this directory, when no absolute path is specified. To change the current directory, use SetCurrentDirectory() .

It's very unlikely, but if this function fails, it will return an empty string.

81.20 GetHomeDirectory

Syntax

```
Result\$ = GetHomeDirectory()
```

Description

Returns the home directory name of the currently logged user. It means than it's possible to read/write in this directory. This directory is user specific and can be useful to store per user data (preferences, plugins etc.).

It's very unlikely, but if this function fails, it will return an empty string.

81.21 GetTemporaryDirectory

Syntax

```
Result\$ = GetTemporaryDirectory()
```

Description

Returns the temporary directory name. All temporary files should be created in this directory, as it is granted to have full read/write access.

It's very unlikely, but if this function fails, it will return an empty string.

81.22 GetFileDate

Syntax

```
Result = GetFileDate(Filename$, DateType)
```

Description

Returns the date of the specified file. The returned date is in the same format than the one of the Date library, so all the date related functions like FormatDate() can be used.

The 'DateType' parameter can be one of the following constants:

```
#PB_Date_Created : returns the file creation date.
#PB_Date_Accessed: returns the last file access date.
#PB_Date_Modified: returns the last file modification date.
```

81.23 GetFileAttributes

Syntax

```
Attributes = GetFileAttributes(Filename$)
```

Description

Returns the attributes of the given Filename$.

On Windows, Attributes is a combination of the following values:

```
#PB_FileSystem_Hidden     : File is hidden
#PB_FileSystem_Archive    : File has been archived and not
 changed since the last time
#PB_FileSystem_Compressed: File is compressed
#PB_FileSystem_Normal     : Normal attributes
#PB_FileSystem_ReadOnly   : File is in readonly mode
#PB_FileSystem_System     : File is a system file
```

On Linux or MacOSX, Attributes is a combination of these values:

```
#PB_FileSystem_Link       : The file is a symbolic link
#PB_FileSystem_ReadUser   : Access flags for the owning user
#PB_FileSystem_WriteUser
#PB_FileSystem_ExecUser
#PB_FileSystem_ReadGroup : Access flags for the owning user's
 group
#PB_FileSystem_WriteGroup
#PB_FileSystem_ExecGroup
#PB_FileSystem_ReadAll    : Access flags for all other users
#PB_FileSystem_WriteAll
#PB_FileSystem_ExecAll
```

To check if one attribute is actually set, just use the '&' (binary AND) and the attribute constant value:

```
1  FileAttributes = GetFileAttributes("C:\Text.txt")
2  If FileAttributes & #PB_FileSystem_Hidden
3    Debug "This file is hidden !"
4  EndIf
```

If the file attributes can't be read, the command returns -1.

Example:

```
1  Value.l = GetFileAttributes("c:\autoexec.bat")
2
3  If Value = -1
4    Debug "Error reading file attributes!"
5  Else
6    If Value & #PB_FileSystem_Hidden     : txt$ + "H" : Else :
     txt$+"-" : EndIf
7    If Value & #PB_FileSystem_Archive    : txt$ + "A" : Else :
     txt$+"-" : EndIf
8    If Value & #PB_FileSystem_Compressed : txt$ + "C" : Else :
     txt$+"-" : EndIf
9    If Value & #PB_FileSystem_Normal     : txt$ + "N" : Else :
     txt$+"-" : EndIf
10   If Value & #PB_FileSystem_ReadOnly   : txt$ + "R" : Else :
     txt$+"-" : EndIf
11   If Value & #PB_FileSystem_System     : txt$ + "S" : Else :
     txt$+"-" : EndIf
```

```
    Debug txt$
  EndIf
```

81.24 NextDirectoryEntry

Syntax

```
Result = NextDirectoryEntry(#Directory)
```

Description

This function must be called after an ExamineDirectory() . It will go step by step into the #Directory and list its contents.

The entry name can be get with the DirectoryEntryName() function. If you want to know whether an entry is a sub-directory or a file, just use the DirectoryEntryType() function.

If no more entries are available in the directory, 'Result' will be 0, else it will be non null.

81.25 RenameFile

Syntax

```
Result = RenameFile(OldFilename$, NewFilename$)
```

Description

Rename the old file to the new file. If the Result is 0, the rename has failed.

Its even possible to rename directory names, just use the related path with directory name.

Example:

```
  RenameFile("C:\Temp\Test.txt", "C:\Temp\TestNew.txt")     ; renaming the Test file in Temp directory
  RenameFile("C:\Temp\", "C:\TempNew\")                     ; renaming the Temp directory itself
```

By using two different directories (and even drives) as parameter it's possible to even move a file.

Example:

```
  If RenameFile("C:\test.txt", "D:\temp\test_backup.txt")
    Debug "Moving and renaming successful."     ; Moving and renaming of the file was successful
  Else
    Debug "Moving and renaming failed."         ; Moving and renaming failed, e.g. because of a not found source file
  EndIf
```

81.26 SetFileDate

Syntax

```
SetFileDate(Filename$, DateType, Date)
```

Description

Changes the date of the specified file.

The date is in the same format than the one of the Date library, so all the date related functions like ParseDate() can be used.

The 'DateType' parameter can be one of the following constants:

```
#PB_Date_Created : change the file creation date.
#PB_Date_Accessed: change the last file access date.
#PB_Date_Modified: change the last file modification date.
```

81.27 SetFileAttributes

Syntax

```
Result = SetFileAttributes(Filename$, Attributes)
```

Description

Set the attributes of the given Filename$. If 'Result' is 0, then the attributes of the file can not be changed. This command also works for directories, if the directory name is given as 'FileName$'.

On Windows, 'Attributes' is a combination of the following values:

```
#PB_FileSystem_Hidden     : File is hidden
#PB_FileSystem_Archive    : File has been archived and not
 changed since the last time
#PB_FileSystem_Compressed: File is compressed
#PB_FileSystem_Normal     : Normal attributes
#PB_FileSystem_ReadOnly   : File is in readonly mode
#PB_FileSystem_System     : File is a system file
```

On Linux and MacOSX, the following values can be used:

```
#PB_FileSystem_ReadUser   : Access flags for the owning user
#PB_FileSystem_WriteUser
#PB_FileSystem_ExecUser
#PB_FileSystem_ReadGroup : Access flags for the owning user's
 group
#PB_FileSystem_WriteGroup
#PB_FileSystem_ExecGroup
#PB_FileSystem_ReadAll    : Access flags for all other users
#PB_FileSystem_WriteAll
#PB_FileSystem_ExecAll
```

To combine several attributes, just use the '||' (binary OR) operand:

```
SetFileAttributes("C:\Text.txt", #PB_FileSystem_Hidden |
 #PB_FileSystem_ReadOnly)
```

81.28 SetCurrentDirectory

Syntax

```
SetCurrentDirectory(Directory$)
```

Description

Changes the current directory. All files accesses are relative to this directory, when no absolute path is specified. To get the current directory, use GetCurrentDirectory() .

Chapter 82

Font

Fonts are widely used on computers since it is the only way to render text in differents sizes and forms.

Note: With Purebasic using a colored font is as yet not possible. This may be achieved by using the StartDrawing() function and and then drawing directly on the visual area.

82.1 FreeFont

Syntax

```
FreeFont(#Font)
```

Description

Free the given Font, previously initialized by LoadFont() .

Note: all remaining fonts are automatically freed when the program ends.

82.2 FontID

Syntax

```
FontID = FontID(#Font)
```

Description

Returns the unique system identifier of the #Font.

This result is sometimes also refered to as a 'Handle'. Take a look at the extra chapter Handles and Numbers for more information.

It is also possible to use the result of FontID() to choose from several loaded fonts.
An example of using FontID() in combination with SetGadgetFont() :

```
1  If OpenWindow(0, 0, 0, 222, 130, "FontID()",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2   ButtonGadget(0, 10, 10, 200, 30, "Click to change the
    font...")
3   Font1 = LoadFont(#PB_Any, "Arial"  ,  8, #PB_Font_Bold)
4   Font2 = LoadFont(#PB_Any, "Verdana", 12, #PB_Font_StrikeOut)
5   UsedFont = 1
6  EndIf
```

```
7
8   Repeat
9     Event = WaitWindowEvent()
10
11  If Event = #PB_Event_Gadget
12    If EventGadget() = 0
13      If UsedFont = 1
14        SetGadgetFont(0, FontID(Font2))
15        UsedFont = 2
16      Else
17        SetGadgetFont(0, FontID(Font1))
18        UsedFont = 1
19      EndIf
20    EndIf
21  EndIf
22
23  Until Event = #PB_Event_CloseWindow
```

82.3 IsFont

Syntax

```
Result = IsFont(#Font)
```

Description

Tests if the given #Font number is a valid and correctly initialized font. This function is bullet-proof and may be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a font is ready to use.

82.4 LoadFont

Syntax

```
Result = LoadFont(#Font, Name$, YSize [, Flags])
```

Description

Tries to open the specified font.

If #PB_Any is used as '#Font' parameter, the new font number will be returned as 'Result'. If 'Result' is 0, the font may not be loaded.

If another font was previously loaded with the same #Font number, then the previous font is automatically freed.

The 'Flags' parameter is optional and may be a combination of the following constants:

```
#PB_Font_Bold        : The font will be bold
#PB_Font_Italic      : The font will be italic
#PB_Font_Underline   : The font will be underlined
#PB_Font_StrikeOut   : The font will be strikeout
#PB_Font_HighQuality: The font will be in high-quality mode
 (slower)
```

Note: Only #PB_Font_Bold and #PB_Font_Italic are supported on all OS. The rest are supported on Windows only.

Example:

```
If OpenWindow(0, 0, 0, 270, 160, "Loading font...",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If LoadFont(1, "Arial", 24)
    SetGadgetFont(#PB_Default, FontID(1))
    TextGadget(0, 10, 10, 250, 40, "Arial 24")
  EndIf
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

Note: Under Windows the font mapper will always attempt to locate a font. If a font name which does not exist such as: "Tim Now Ronin", is used, then the font mapper will attempt to find the closest match. This will be based upon such criteria as: the font name, the font height, the style and so forth. Therefore, the assumption can not be made that a font will not be loaded, due only to an incorrect font name, size, ect...

Chapter 83

Ftp

Ftp (File Transfer Protocol) is a way to share files based amonst users on a client-server model. This library implement the client side of Ftp, and allows to connect to a remote server and manipulate files (like download, upload, list available files, navigate trough the directories and more).

All the Ftp commands are based on the network library , so InitNetwork() has to be called before using them.

83.1 AbortFTPFile

Syntax

```
AbortFTPFile(#Ftp)
```

Description

Abort the current background file transfer previously started with SendFTPFile() or ReceiveFTPFile() . If no file transfer is in progress, this function has no effect.

83.2 CheckFTPConnection

Syntax

```
Result = CheckFTPConnection(#Ftp)
```

Description

Check if the specified #Ftp connection is still connected to the server. If 'Result' is 0, the connection has been closed by the server.

83.3 CloseFTP

Syntax

```
CloseFTP(#Ftp)
```

Description

Close the specified #Ftp client connection previously opened with OpenFTP() .

Note: all remaining ftp connection are automatically closed when the program ends.

83.4 CreateFTPDirectory

Syntax

```
Result = CreateFTPDirectory(#Ftp, Directory$)
```

Description

Create a new directory on the #Ftp server. If 'Result' is 0, the directory has not be created. The new directory will be created in the current directory (see GetFTPDirectory() and SetFTPDirectory()). This is not possible to specify a subpath in the Directory$.

83.5 DeleteFTPDirectory

Syntax

```
Result = DeleteFTPDirectory(#Ftp, Directory$)
```

Description

Delete a directory on the #Ftp server. The directory has to be empty, or the delete with fail. If 'Result' is 0, the directory has not be deleted. The deleted directory has to be in the current directory (see GetFTPDirectory() and SetFTPDirectory()). This is not possible to specify a subpath in the Directory$.

83.6 DeleteFTPFile

Syntax

```
Result = DeleteFTPFile(#Ftp, Filename$)
```

Description

Delete a file on the #Ftp server. The deleted file has to be in the current directory (see GetFTPDirectory() and SetFTPDirectory()). This is not possible to specify a subpath in the Filename$. If 'Result' is 0, the file has not be deleted.

83.7 ExamineFTPDirectory

Syntax

```
Result = ExamineFTPDirectory(#Ftp)
```

Description

Start to examine the current #Ftp directory for future listing with functions NextFTPDirectoryEntry() , FTPDirectoryEntryName() , FTPDirectoryEntryType() , FTPDirectoryEntryAttributes() , FTPDirectoryEntryDate() and FTPDirectoryEntrySize() . To change the current directory, use SetFTPDirectory() . If 'Result' is 0, the directory can not be examine.

Note: for now, only unix-like servers are supported for directory listing. The FTP protocol doesn't specify how a directory listing has to be returned, so every server has its own way to returns the directory informations. Fortunately, most of the world servers are running under unix/linux and uses the same way to return the data. That said, there is no warranty than the listing works on every servers. This command will be updated on demand to automatically handle more servers type. If the server isn't supported, please use FTPDirectoryEntryRaw() to get back the raw information about each entry.

Example:

```
InitNetwork()

If OpenFTP(0, "ftp.free.fr", "anonymous", "")
  If ExamineFTPDirectory(0)
    While NextFTPDirectoryEntry(0)
      Debug FTPDirectoryEntryName(0)
    Wend
  EndIf
Else
  Debug "Can't connect to ftp.free.fr"
EndIf
```

83.8 GetFTPDirectory

Syntax

```
Result\$ = GetFTPDirectory(#Ftp)
```

Description

Return the current #Ftp directory, relative to the FTP root. To change the current directory, use SetFTPDirectory()

83.9 FinishFTPDirectory

Syntax

```
FinishFTPDirectory(#Ftp)
```

Description

Finish the enumeration started with ExamineFTPDirectory() . This allows to free the resources associated with the #Ftp listing.

Note: when the program ends, all remaining ExamineFTPDirectory() are automatically finished.

83.10 FTPDirectoryEntryAttributes

Syntax

```
Attributes = FTPDirectoryEntryAttributes(#Ftp)
```

Description

Returns the attributes of the current entry in the #Ftp being listed with ExamineFTPDirectory() and NextFTPDirectoryEntry() functions.

Attributes is a combination of the following values:

```
#PB_FTP_ReadUser  : Access flags for the owning user
#PB_FTP_WriteUser
#PB_FTP_ExecuteUser
#PB_FTP_ReadGroup : Access flags for the owning user's group
```

```
#PB_FTP_WriteGroup
#PB_FTP_ExecuteGroup
#PB_FTP_ReadAll    : Access flags for all other users
#PB_FTP_WriteAll
#PB_FTP_ExecuteAll
```

To check if one attribute is actually set, just use the '&' (binary AND) and the attribute constant value:

```
[...]

FileAttributes = FTPDirectoryEntryAttributes(#Ftp)
If FileAttributes & #PB_FTP_ReadUser
  Debug "This file has the ReadUser flag"
EndIf
```

83.11 FTPDirectoryEntryDate

Syntax

```
Result = FTPDirectoryEntryDate(#Ftp)
```

Description

Returns the date of the current entry in the #Ftp being listed with ExamineFTPDirectory() and NextFTPDirectoryEntry() functions.

The date is in the same format than the one of the Date library, so all the date related functions like FormatDate() can be used.

83.12 FTPDirectoryEntryName

Syntax

```
Filename\$ = FTPDirectoryEntryName(#Ftp)
```

Description

Returns the name of the current entry in the #Ftp being listed with ExamineFTPDirectory() and NextFTPDirectoryEntry() functions.

83.13 FTPDirectoryEntryType

Syntax

```
Result = FTPDirectoryEntryType(#Ftp)
```

Description

Returns the type of the current entry in the #Ftp being listed with ExamineFTPDirectory() and NextFTPDirectoryEntry() functions.

'Result' can be one of the following constants:

```
#PB_FTP_File      : This entry is a file.
#PB_FTP_Directory: This entry is a directory.
```

83.14 FTPDirectoryEntryRaw

Syntax

```
Entry\$ = FTPDirectoryEntryRaw(#Ftp)
```

Description

Returns the raw line of the current entry in the #Ftp being listed with ExamineFTPDirectory() and NextFTPDirectoryEntry() functions, as it has been sent by the FTP server. It can be useful when the server isn't supported by ExamineFTPDirectory() , so the informations about the directory listing can still be retrieved and parsed accordingly.

Example:

```
InitNetwork()

If OpenFTP(0, "ftp.free.fr", "anonymous", "")
  If ExamineFTPDirectory(0)
    While NextFTPDirectoryEntry(0)
      Debug FTPDirectoryEntryRaw(0)
    Wend
  EndIf
Else
  Debug "Can't connect to ftp.free.fr"
EndIf
```

83.15 FTPDirectoryEntrySize

Syntax

```
Size = FTPDirectoryEntrySize(#Ftp)
```

Description

Returns the size of the current entry in the #Ftp being listed with ExamineFTPDirectory() and NextFTPDirectoryEntry() functions.

83.16 FTPProgress

Syntax

```
Result = FTPProgress(#Ftp)
```

Description

Return the progress of the current file transfer, started either with ReceiveFTPFile() or SendFTPFile() .

'Result' is the actual number of bytes which has been recieved/sent or can be one of the following value:

```
#PB_FTP_Started : The file transfer is in the initialization
 phase.
#PB_FTP_Finished: The file transfer is finished correctly.
#PB_FTP_Error   : The file transfer is finished but an error
 occured.
```

83.17 IsFtp

Syntax

```
Result = IsFtp(#Ftp)
```

Description

Tests if the given #Ftp number is a valid and correctly initialized ftp client. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a ftp is ready to use.

83.18 NextFTPDirectoryEntry

Syntax

```
Result = NextFTPDirectoryEntry(#Ftp)
```

Description

This function must be called after an ExamineFTPDirectory() . It will go step by step into the #Ftp and list its contents.

The entry name can be get with FTPDirectoryEntryName() . To know whether an entry is a file or directory, use FTPDirectoryEntryType() .

If no more entries are available in the directory, 'Result' will be 0, else it will be non null.

83.19 OpenFTP

Syntax

```
Result = OpenFTP(#Ftp, ServerName$, User$, Password$ [, Passive
    [, Port]])
```

Description

Try to open a connection on the specified FTP server. If #PB_Any is used as '#Ftp' parameter, the new ftp number will be returned as 'Result'. 'ServerName$' has to be a valid ftp server adress. 'User$' and 'Password$' will be used to authenticate on the server. For public server, 'User$' can be set to "anonymous" and 'Password$' be let empty. By default the connection is opened in passive mode, as it's the most compliant mode (most PC have a firewall which will complain if a ftp connection is initialized in active mode). The default port is 21 but can be overriden with the 'Port' parameter.

If the connection to the server can't be successly completed, 'Result' will be 0.

InitNetwork() has to be called before using this command, as all the FTP commands are based on the network library.

Example:

```
InitNetwork()

If OpenFTP(0, "ftp.free.fr", "anonymous", "")
  Debug "Successfully connected"
Else
  Debug "Can't connect to ftp.free.fr"
EndIf
```

83.20 ReceiveFTPFile

Syntax

```
Result = ReceiveFTPFile(#Ftp, RemoteFilename$, Filename$ [,
    Asynchronous])
```

Description

Receive a file from a #Ftp server. 'RemoteFilename$' has to be in the current ftp directory (see GetFTPDirectory() and SetFTPDirectory()). 'Filename$' is the local file where the download file will be written. By default, the download is synchronous and blocks the program until it is finished. The 'Asynchronous' mode (optional parameter = #True) allows to do the download in the backgroud, and FTPProgress() can be used to get the current state of the download. An asynchronous transfer can be aborted with AbortFTPFile() . Only one file can be downloaded or uploaded (see SendFTPFile()) in the background at once.

If the file has not be correctly recieved, 'Result' will be 0.

83.21 RenameFTPFile

Syntax

```
Result = RenameFTPFile(#Ftp, Filename$, NewFilename$)
```

Description

Rename a file on the #Ftp server. The renamed file has to be in the current directory (see GetFTPDirectory() and SetFTPDirectory()). This is not possible to specify a subpath in the Filename$. If 'Result' is 0, the file has not be renamed.

83.22 SendFTPFile

Syntax

```
Result = SendFTPFile(#Ftp, Filename$, RemoteFilename$ [,
    Asynchronous])
```

Description

Send a file to a #Ftp server. 'RemoteFilename$' has to be in the current ftp directory (see GetFTPDirectory() and SetFTPDirectory()). 'Filename$' is the local file to be sent. By default, the upload is synchronous and blocks the program until it is finished. The 'Asynchronous' mode (optional parameter = #True) allows to do the upload in the backgroud, and FTPProgress() can be used to get the current state of the upload. An asynchronous transfer can be aborted with AbortFTPFile() . Only one file can be uploaded or downloaded (see ReceiveFTPFile()) in the background at once.

If the file has not be correctly sent, 'Result' will be 0.

83.23 SetFTPDirectory

Syntax

```
Result = SetFTPDirectory(#Ftp, Directory$)
```

Description

Change the current #Ftp directory, relative to the current directory. If the directory couldn't be changed, 'Result' will be 0.

To get the current directory, use GetFTPDirectory() . Subpath are not supported, only immediate directory change are allowed: to change to several level of directory, this command will have to be called several time.

Example:

```
InitNetwork()

If OpenFTP(0, "ftp.free.fr", "anonymous", "")

  Debug "Successfully connected"

  If SetFTPDirectory(0, "pub")
    If SetFTPDirectory(0, "linux")
      Debug "Cool, changed to '/pub/linux'"
      Debug GetFTPDirectory(0)
    Else
      Debug "Can't change to '/pub/linux'"
    EndIf
  Else
    Debug "Can't change to 'pub'"
  EndIf

Else
  Debug "Can't connect to ftp.free.fr"
EndIf
```

Chapter 84

Gadget

The Gadgets in PureBasic (in other languages also called "controls" or "widgets") are a generic name for all the interface components: button, combobox, listview, panels, ... This library is OS independent and uses the real OS Graphical User Interface (GUI) components.

Before using gadgets there will be normally opened a window first, furthermore there will be often used menus , toolbars and statusbars when creating graphical user interfaces.

The functions that create a new gadget return the new gadget number (called #Gadget in this library) if #PB_Any was used to create the gadget. If a static number was given to identify the gadget instead of #PB_Any, then the functions return the OS identifier for the created Gadget. These OS identifiers (or also called handles) can be used for other PureBasic functions, as well for WinAPI functions like SendMessage_() etc. Look at the chapter Handles and Numbers in the reference manual for more information.

84.1 AddGadgetColumn

Syntax

```
AddGadgetColumn(#Gadget, Position, Title$, Width)
```

Description

Adds a column to the specified #Gadget. The position specifies the column index where the new item should be inserted. Column indexes start from 0, which is the leftmost column, and increase by 1 for each column to the right. When you add a column, all the old columns which are on the right of the new column will have a position which is one more than they previously had. The 'Width' represents the initial Width of the new column.

The gadget type can be one of the following:

- ListIconGadget()
- ExplorerListGadget()

For ExplorerListGadget() you can use this function to completly customize the information that the gadget displays, by first removing the standard columns with RemoveGadgetColumn() and adding new ones of your choise. Note that the 'Name' column doesn't need to be the first one in the gadget.
Note: To update the gadget contents after you have added new columns, use SetGadgetText() . To fill in a custom column, use SetGadgetItemText() for each item, after you have received a #PB_EventType_Change event for the gadget.
When adding a column, use one of the following constants in the Title$ field to create a column whose contents will be automatically updated by the gadget, or use any title string you wish to create an empty column which you can then fill with SetGadgetItemText() .

```
#PB_Explorer_Name       : displays the name of the
 file/directory
#PB_Explorer_Size       : displays the filesize in Kb
#PB_Explorer_Type       : displays a string describing the
 filetype
#PB_Explorer_Attributes : displays the attributes of the
 file/directory
#PB_Explorer_Created    : displays the time the
 file/directory was created
#PB_Explorer_Modified   : displays the time the
 file/directory was last modified
#PB_Explorer_Accessed   : displays the time the
 file/directory was last accessed
```

Example:

```
If OpenWindow(0, 0, 0, 400, 150, "ListIcon - Add Columns",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ListIconGadget(0, 10, 10, 380, 100, "Standard Column", 150,
 #PB_ListIcon_GridLines)
  ButtonGadget(1, 10, 120, 150, 20, "Add new column")
  index = 1     ; "Standard column" has already index 0
  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Gadget
      If EventGadget() = 1
        AddGadgetColumn(0, index, "Column "+Str(index), 80)
        index + 1
      EndIf
    EndIf
  Until Event = #PB_Event_CloseWindow
EndIf
```

84.2 AddGadgetItem

Syntax

```
Result = AddGadgetItem(#Gadget, Position, Text$ [, ImageID [,
   Flags]])
```

Description

Add an item to the specified #Gadget. An optional ImageID() can be specified for any gadgets which support it (like ListIconGadget()). The used image should be in the standard 16x16 icon size. 'Result' is only supported in combination with MDIGadget() (see below).

The following gadgets are supported:

- ComboBoxGadget() : supports the ImageID if the #PB_ComboBox_Image is set.
- EditorGadget()
- ListIconGadget() : supports the ImageID.
- ListViewGadget()
- MDIGadget() : ImageID can contain an icon for the childwindow titlebar. Flags can specify the new window flags.
- PanelGadget() : supports the ImageID.
- TreeGadget() : supports the ImageID. 'Flags' is required and specifies the new sublevel.

The position specifies the item index where the new item should be inserted. To add the item at the start, use an index of 0. To add this item to the end of the current item list, use a value of -1. Remember that when you add an item that all current items which come after the new one will have their positions increased by 1.

For the MDIGadget() the 'Position' parameter specifies the #Window number for the new MDI childwindow. #PB_Any can be used, in which case the returnvalue will be the new number assigned by PB. The 'Flags' parameter can be used to specifiy the flags for the new window (see OpenWindow()). The #PB_Window_Borderless, #PB_Window_Screencentered and #PB_Window_WindowCentered flags are not supported for MDI windows.

For the TreeGadget() the 'Flags' parameter is always required. It specifies the new sublevel for the item. If the sublevel is greater than that of the previous item, the new one will become this item's child. If it is lower, it will be added after the parent of the previous item.

84.3 ButtonImageGadget

Syntax

```
Result = ButtonImageGadget(#Gadget, x, y, Width, Height,
    ImageID [, Flags])
```

Description

Create a button gadget in the current GadgetList.

If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function.

The 'Flags' parameter can have be #PB_Button_Toggle to create a toggle button.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

- GetGadgetState() can be used to get the toggle state of the gadget.
- SetGadgetState() can be used to set the toggle state of the gadget.
- GetGadgetAttribute() with the following values:

```
#PB_Button_Image        : Get the displayed image.
#PB_Button_PressedImage: Get the image displayed when the
 button is pressed.
```

- SetGadgetAttribute() with the following values:

```
#PB_Button_Image        : Set the displayed image.
#PB_Button_PressedImage: Set the image displayed when the
 button is pressed.
```

To get a valid ImageID to display in this gadget, ImageID() from the Image library can be used. If ImageID is set to 0, then no image will be displayed.

Example:

```
1  If OpenWindow(0, 0, 0, 120, 100, "ButtonImage",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2   If LoadImage(0, "map.bmp")    ; change 2nd parameter to the
    path/filename of your image
3      ButtonImageGadget(0, 10, 10, 100, 83, ImageID(0))
4   EndIf
5   Repeat
6   Until WaitWindowEvent() = #PB_Event_CloseWindow
7  EndIf
```

84.4 ButtonGadget

Syntax

```
Result = ButtonGadget(#Gadget, x, y, Width, Height, Text$ [,
    Flags])
```

Description

Create a button gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function.

'Flags' are always optional and can be composed of one or several (using the bitwise OR operator '||') of the following constants:

```
#PB_Button_Right      : Aligns the button text at the right.
 (not supported on Mac OSX)
#PB_Button_Left       : Aligns the button text at the left.
 (not supported on Mac OSX)
#PB_Button_Default    : Makes the button look as if it is the
 default button in the window.
#PB_Button_MultiLine : If the text is too long, it will be
 displayed on several lines. (not supported on Mac OSX)
#PB_Button_Toggle     : Creates a toggle button: one click
 pushes it, another will release it.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to control the gadget:

- SetGadgetText() : Changes the text of the ButtonGadget.
- GetGadgetText() : Returns the text of the ButtonGadget.
- SetGadgetState() : Used with #PB_Button_Toggle buttons to set the actual state (1 = toggled, 0 = normal).
- GetGadgetState() : Used with #PB_Button_Toggle buttons to get the actual state of the button (1 = toggled, 0 = normal).

Special Windows feature:
In the button 'Text$' you can use the special '&' character to underline a specific letter: "&Button" will give in reality : Button
(Note: under Windows XP the underlined letter will only be displayed if this is not deactivated in the system settings.)

In combination with the AddKeyboardShortcut() function a keyboard shortcut for the button can be realized this way.

Example:

```
; Shows possible flags of ButtonGadget in action...
If OpenWindow(0, 0, 0, 222, 200, "ButtonGadgets",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ButtonGadget(0, 10, 10, 200, 20, "Standard Button")
  ButtonGadget(1, 10, 40, 200, 20, "Left Button",
 #PB_Button_Left)
```

```
    ButtonGadget(2, 10, 70, 200, 20, "Right Button", #PB_Button_Right)
    ButtonGadget(3, 10,100, 200, 60, "Multiline Button  (longer text gets automatically wrapped)", #PB_Button_MultiLine)
    ButtonGadget(4, 10,170, 200, 20, "Toggle Button", #PB_Button_Toggle)
    Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```

84.5 CalendarGadget

Syntax

```
Result = CalendarGadget(#Gadget, x, y, Width, Height [, Date [, Flags]])
```

Description

Create a calendar gadget in the current GadgetList. This gadget displays a month calendar and lets the user select a date. The dates used by this gadget and the functions for it use the same date format as the PB date library.

If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function.

The Date parameter can optionally be used to select a certain date, if not specified, the current date is selected.

You can specify #PB_Calendar_Borderless in the Flags parameter if you want the gadget to be created without a border.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used for this gadget:

- SetGadgetState() : Set the currently displayed date.
- GetGadgetState() : Get the currently displayed date.
- SetGadgetItemState() : Make a specific date appear bold (Windows only).
- GetGadgetItemState() : Get the bold state of a specific date (Windows only).
- SetGadgetAttribute() : With the following attributes:

```
#PB_Calendar_Minimum: Set the minimum selectable date
#PB_Calendar_Maximum: Set the maximum selectable date this gadget.
```

- GetGadgetAttribute() : With the following attributes:

```
#PB_Calendar_Minimum: Get the minimum selectable date
#PB_Calendar_Maximum: Get the maximum selectable date this gadget.
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

```
#PB_Gadget_BackColor       : backgroundcolor
#PB_Gadget_FrontColor      : textcolor for displayed days
#PB_Gadget_TitleBackColor  : backgroundcolor of the month title
#PB_Gadget_TitleFrontColor: textcolor of the month title
#PB_Gadget_GrayTextColor   : textcolor for days not of the
 current month
```

Example:

```
If OpenWindow(0, 0, 0, 220, 200, "CalendarGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  CalendarGadget(0, 10, 10, 200, 180)
  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

84.6 CheckBoxGadget

Syntax

```
Result = CheckBoxGadget(#Gadget, x, y, Width, Height, Text$ [,
   Flags])
```

Description

Create a checkbox gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function. The 'Text$' is a description of the checkbox which will be displayed to the right of it.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

- GetGadgetState() can be used to get the current gadget state.
- SetGadgetState() can be used to change the gadget state.

'Flags' are always optional and can be one of the following constants:

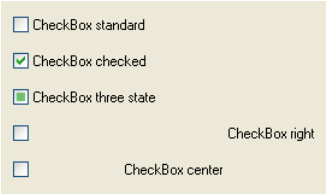
```
#PB_CheckBox_Right      : Aligns the text to right (Windows
 only).
#PB_CheckBox_Center     : Centers the text (Windows only).
#PB_CheckBox_ThreeState: Create a checkbox that can have a
 third "inbetween" state.
```

The #PB_CheckBox_ThreeState flag can be used for a checkbox that represents the state of multiple items. The "inbetween" state can then be used to indicate that the setting is not the same for all items. By clicking on the checkbox, the user can bring it back to either the

"on" or "off" state to apply this to all the items. Therefore the "inbetween" state can only be set by the program via SetGadgetState() and not by the user by clicking on the checkbox.

Example:

```
If OpenWindow(0, 0, 0, 270, 160, "CheckBoxGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  CheckBoxGadget(0, 10,  10, 250, 20, "CheckBox standard")
  CheckBoxGadget(1, 10,  40, 250, 20, "CheckBox checked"):
 SetGadgetState(1, #PB_Checkbox_Checked)
  CheckBoxGadget(2, 10,  70, 250, 20, "CheckBox three state",
 #PB_CheckBox_ThreeState): SetGadgetState(2,
 #PB_Checkbox_Inbetween)
  CheckBoxGadget(3, 10, 100, 250, 20, "CheckBox right",
 #PB_CheckBox_Right)
  CheckBoxGadget(4, 10, 130, 250, 20, "CheckBox center",
 #PB_CheckBox_Center)
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.7 ClearGadgetItems

Syntax

```
ClearGadgetItems(#Gadget)
```

Description

Clears all the items from the specified #Gadget. The Gadget must be one of the following types:

- ComboBoxGadget()
- EditorGadget()
- ListIconGadget()
- ListViewGadget()
- MDIGadget()
- PanelGadget()
- TreeGadget()

Example:

```
If OpenWindow(0, 0, 0, 220, 150, "ClearGadgetItems",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ListViewGadget(0,10,10,200,100)
  For a = 1 To 10 : AddGadgetItem (0, -1, "Item "+Str(a)) :
 Next     ; add 10 items
  ButtonGadget(1, 10, 120, 150, 20, "Clear Listview contents")
  Repeat
    Event = WaitWindowEvent()
```

```
      If Event = #PB_Event_Gadget
        If EventGadget() = 1
          ClearGadgetItems(0)
        EndIf
      EndIf
    Until Event = #PB_Event_CloseWindow
  EndIf
```

84.8 CloseGadgetList

Syntax

```
CloseGadgetList()
```

Description

Terminate the current gadget list creation and go back to the previous GadgetList.

This is especially useful for the following gadgets:

- ContainerGadget()
- PanelGadget()
- ScrollAreaGadget()

84.9 ComboBoxGadget

Syntax

```
Result = ComboBoxGadget(#Gadget, x, y, Width, Height [, Flags])
```

Description

Create a ComboBox gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function. Once a ComboBox is created, its list of items is empty.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on the list contents:

- AddGadgetItem() : Add an item.
- GetGadgetItemText() : Returns the gadget item text.
- CountGadgetItems() : Count the items in the current combobox.
- ClearGadgetItems() : Remove all the items.
- RemoveGadgetItem() : Remove an item.
- SetGadgetItemText() : Changes the gadget item text.

- GetGadgetState() : Get the index (starting from 0) of the current element.
- GetGadgetText() : Get the (text) content of the current element.
- SetGadgetState() : Change the selected element.
- SetGadgetText() : Set the displayed text. If the ComboBoxGadget is not editable, the text must be in the dropdown list.
- GetGadgetItemData() : Returns the value that was stored with item.
- SetGadgetItemData() : Stores a value with the item.

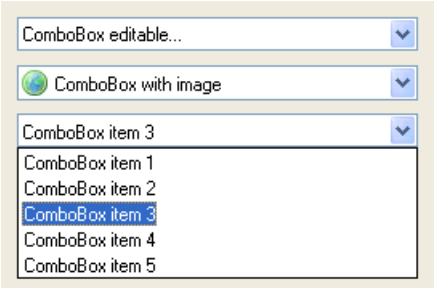
'Flags' are always optional and can be composed (using the '||' operator) of one of the following constants:

```
#PB_ComboBox_Editable  : Makes the ComboBox editable
#PB_ComboBox_LowerCase : All text entered in the ComboBox
 will be converted to lower case.
```

```
#PB_ComboBox_UpperCase : All text entered in the ComboBox
 will be converted to upper case.
#PB_ComboBox_Image     : Enable support for images in items
 (not supported for editable ComboBox on OSX)
```

Example:

```
UsePNGImageDecoder()
LoadImage(0, #PB_Compiler_Home +
 "examples/sources/Data/world.png")

If OpenWindow(0, 0, 0, 270, 180, "ComboBoxGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ComboBoxGadget(0, 10, 10, 250, 21, #PB_ComboBox_Editable)
    AddGadgetItem(0, -1, "ComboBox editable...")

  ComboBoxGadget(1, 10, 40, 250, 21, #PB_ComboBox_Image)
    AddGadgetItem(1, -1, "ComboBox item with image",
 ImageID(0))

  ComboBoxGadget(2, 10, 70, 250, 21)
    For a = 1 To 5
      AddGadgetItem(2, -1,"ComboBox item " + Str(a))
    Next

  SetGadgetState(0, 0)
  SetGadgetState(1, 0)
  SetGadgetState(2, 2)     ; set (beginning with 0) the third
 item as active one

  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.10 ContainerGadget

Syntax

```
Result = ContainerGadget(#Gadget, x, y, Width, Height [, Flags])
```

Description

Creates a container gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'.

It's a simple panel gadget which can contain other gadgets. Once the gadget is created, all future created gadgets will be created inside the container. When all the needed gadgets have

been created, CloseGadgetList() must be called to return to the previous GadgetList. OpenGadgetList() can be used later to add others gadgets on the fly in the container area.

'Flags' are optional and can be composed of one of the following constants:

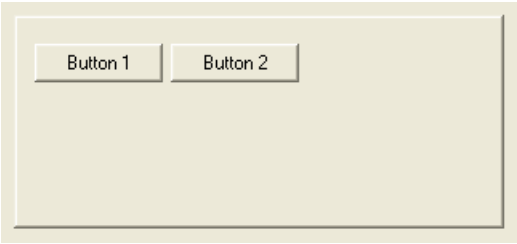
```
#PB_Container_BorderLess : Without any border (Default).
#PB_Container_Flat       : Flat frame.
#PB_Container_Raised     : Raised frame.
#PB_Container_Single     : Single sunken frame.
#PB_Container_Double     : Double sunken frame.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with #PB_Gadget_BackColor as type to change the gadget background.

Example:

```
1  If OpenWindow(0, 0, 0, 322, 150, "ContainerGadget",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2    ContainerGadget(0, 8, 8, 306, 133, #PB_Container_Raised)
3      ButtonGadget(1, 10, 15, 80, 24, "Button 1")
4      ButtonGadget(2, 95, 15, 80, 24, "Button 2")
5    CloseGadgetList()
6    Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
7  EndIf
```



84.11 CountGadgetItems

Syntax

```
Result = CountGadgetItems(#Gadget)
```

Description

Returns the number of items in the specified #Gadget. If there are any items in the gadget then the count starts from 1. If the #Gadget doesn't contain any items, 0 is returned. This is a universal function which works for all gadgets which handle several items:

- ComboBoxGadget()
- EditorGadget()
- ExplorerListGadget()
- ListIconGadget()
- ListViewGadget()
- MDIGadget()
- PanelGadget()
- TreeGadget()

84.12 DateGadget

Syntax

```
Result = DateGadget(#Gadget, x, y, Width, Height [, Mask$ [,
    Date [, Flags]]])
```

Description

Creates a String gadget in the current GadgetList, in which a date and/or time can be entered. This gadget uses the same date format for its functions as used by the Date library. So you can use for example FormatDate() to display the results you get from GetGadgetState() in a proper format. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the numeric number returned by the EventGadget() function.

With the optional Mask$ parameter you can specify the format in which the date can be entered. See FormatDate() for the format of this mask. Important note: The gadget does not support the display of seconds, so if you specify "%ss" in the Mask$ parameter, it will simply be ignored! If you don't specify the mask or specify an empty string, a default mask will be choosen. The mask can be modified with the SetGadgetText() function. With the optional Date parameter you can set the displayed date to any date of your choise. Not specifying it or specifying a 0 value will display the current date.

By default, the gadget has a button to display a calendar in which the user can choose a date (see image below). You can change this by specifying #PB_Date_UpDown in the Flags parameter. This will make the gadget display an up/down button that lets the user change the current selected part of the gadget. This option is only available on Windows.
If you specify #PB_Date_Checkbox in the Flags parameter, the Gadget will have a checkbox with which the user can set the Gadget to 'no date' (if the checkbox is unchecked). While the checkbox is unchecked, GetGadgetState() will return 0. To change the state of the checkbox, use SetGadgetState() with either 0 (=checkbox unchecked) or any valid date (=checkbox checked).

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used for this gadget:

- SetGadgetState() : Set the currently displayed date.
- SetGadgetText() : Change the input mask of the gadget.
- GetGadgetState() : Get the currently displayed date.
- GetGadgetText() : Get the current displayed date as a string, as it is displayed in the gadget.
- GetGadgetAttribute() : With the following attributes:

```
#PB_Date_Minimum: Get the minimum date that can be entered
#PB_Date_Maximum: Get the maximum date that can be entered
```

- SetGadgetAttribute() : With the following attributes:

```
#PB_Date_Minimum: Set the minimum date that can be entered
#PB_Date_Maximum: Set the maximum date that can be entered
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType' to color the dropdown calendar (the edit area cannot be colored):

```
#PB_Gadget_BackColor      : backgroundcolor
#PB_Gadget_FrontColor     : textcolor for displayed days
#PB_Gadget_TitleBackColor : backgroundcolor of the month title
#PB_Gadget_TitleFrontColor: textcolor of the month title
#PB_Gadget_GrayTextColor  : textcolor for days not of the
 current month
```

Example:

```
If OpenWindow(0, 0, 0, 200, 250, "DateGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  DateGadget(0, 10, 10, 180, 25, "Date: %mm/%dd/%yyyy Time:
 %hh:%ii")
  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

84.13 DisableGadget

Syntax

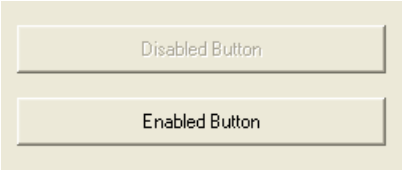
```
DisableGadget(#Gadget, State)
```

Description

Disable or enable the gadget. If State = 1, the gadget will be disabled, if State = 0 it will be enabled.

Example:

```
If OpenWindow(0, 0, 0, 250, 105, "Disable/enable buttons...",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ButtonGadget(0, 10, 15, 230, 30, "Disabled Button") :
 DisableGadget(0, 1)
  ButtonGadget(1, 10, 60, 230, 30, "Enabled Button") :
 DisableGadget(1, 0)
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.14 EditorGadget

Syntax

```
Result = EditorGadget(#Gadget, x, y, Width, Height [, Flags])
```

Description

Creates an Editor gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by the EventGadget() function.

'Flags' can be the following value:

```
#PB_Editor_ReadOnly: The user cannot edit the text in the
 gadget.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on the editor content:

- AddGadgetItem() : Add a text line.
- CountGadgetItems() : Return the number of lines in the editor gadget.
- GetGadgetItemText() : Get the specified line text.
- GetGadgetText() : Get the text content of the editor gadget.
- RemoveGadgetItem() : Remove a line in the editor.
- ClearGadgetItems() : Clear the text content.
- SetGadgetItemText() : Set the specified line text.
- SetGadgetText() : Change the text content of the editor gadget.
- SetGadgetAttribute() : With the following attribute:

```
#PB_Editor_ReadOnly: Set the readonly state. (0 means
 editable, nonzero means readonly)
```

- GetGadgetAttribute() : With the following attribute:

```
#PB_Editor_ReadOnly: Get the readonly state. (0 means
 editable, nonzero means readonly)
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

```
#PB_Gadget_BackColor       : backgroundcolor
#PB_Gadget_FrontColor      : textcolor
```

Example:

```
If OpenWindow(0, 0, 0, 322, 150, "EditorGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  EditorGadget(0, 8, 8, 306, 133)
  For a = 0 To 5
    AddGadgetItem(0, a, "Line "+Str(a))
  Next
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

84.15 ExplorerComboGadget

Syntax

```
Result = ExplorerComboGadget(#Gadget, x, y, Width, Height,
   Directory$, [, Flags])
```

Description

Creates a ComboBox that lets you display a path and all its parent folders, so the user can choose one of them. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. You can find such a ComboBox, for example, in the OpenFileRequester() . This can be a nice addition to the ExplorerListGadget() .

Directory$ is the initial displayed directory (must be set as full path), an empty string specifies the root folder. If the #PB_Explorer_DrivesOnly flag is set, Directory$ may only be a drive letter. Everything that follows the drive letter in this case will be ignored.

'Flags' can be a combination of the following values:

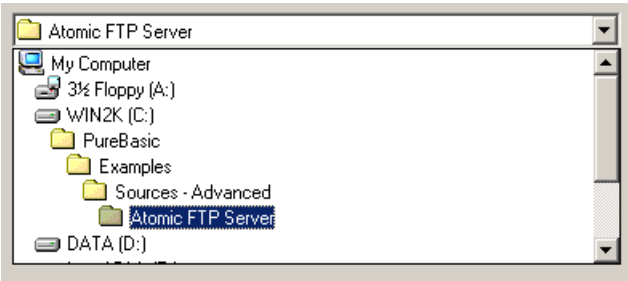
```
#PB_Explorer_DrivesOnly    : The gadget will only display
 drives to choose from.
#PB_Explorer_Editable      : The gadget will be editable with
 an autocomplete feature. With this flag set, it acts exactly
 like the one in Windows Explorer.
#PB_Explorer_NoMyDocuments: The 'My Documents' Folder will
 not be displayed as a separate item.
```

The following functions can be used to control the gadget:

- GetGadgetText() : Get the currently displayed directory. Use this to ckeck what the user has selected after you get an event for this Gadget.
- SetGadgetText() : Changes the currently displayed directory.

Example:

```
1  If OpenWindow(0, 0, 0, 400, 100, "ExplorerComboGadget",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2   ExplorerComboGadget(0, 10, 40, 380, 200, "C:\",
    #PB_Explorer_Editable)
3   Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
4  EndIf
```



84.16 ExplorerListGadget

Syntax

```
Result = ExplorerListGadget(#Gadget, x, y, Width, Height,
   Directory$, [, Flags])
```

Description

Creates a listing of a directory just as Explorer does. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. It lets the user choose a file or a folder and (if you do not prevent it by a flag) navigate through the whole directory tree.

Directory$ is the initial displayed directory, it can include one or multiple patterns, like "C:*.pb;*.pbi". If no pattern is included, the directory must end with a '\'. Including no directory will display the root containing the drives. Including no pattern defaults to '*.*'. So a Directory$ of "" will display the root and set '*.*' as pattern.

'Flags' can be a combination of the following values:

```
#PB_Explorer_BorderLess          : Create Gadget without
 borders.
#PB_Explorer_AlwaysShowSelection : The selection is visible,
 even when the gadget is not activated.
#PB_Explorer_MultiSelect         : Enable multiple selection
 of items in the gadget.
#PB_Explorer_GridLines           : Display separator lines
 between rows and columns.
#PB_Explorer_HeaderDragDrop      : In report view, the
 headers can be changed by Drag'n'Drop.
#PB_Explorer_FullRowSelect       : The selection covers the
 full row instead of the first column.

#PB_Explorer_NoFiles             : No files will be displayed.
#PB_Explorer_NoFolders           : No folders will be
 displayed.
#PB_Explorer_NoParentFolder      : There will be no [..] link
 to the parent folder.
#PB_Explorer_NoDirectoryChange   : The directory cannot be
 changed by the user.
#PB_Explorer_NoDriveRequester    : There will be no 'please
 insert drive X:' displayed.
#PB_Explorer_NoSort              : The user cannot sort the
 content by clicking on a column header.
#PB_Explorer_NoMyDocuments       : The 'My Documents' Folder
 will not be displayed as a separate item.
#PB_Explorer_AutoSort            : The content will be sorted
 automatically by name.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to control the gadget:

- AddGadgetColumn() : Add a new automatically or custom filled column to the gadget. See the AddGadgetColumn() function help for more details.
- RemoveGadgetColumn() : Remove a column from the gadget.
- GetGadgetText() : Get the currently displayed directory.
- SetGadgetText() : Changes the currently displayed directory, or the current pattern for files.
- GetGadgetState() : Get the first selected item (-1 if none selected).
- GetGadgetItemText() : Get the name of an item (or column header, if item = -1).
- SetGadgetItemText() : Alter the contents of any items text (or column header, if item = -1), or fill a custom column with data.
- GetGadgetItemState() : Check if an item is a directory or a file, and if it is currently selected.
- CountGadgetItems() : Count the items in the current directory.
- GetGadgetAttribute() / SetGadgetAttribute() : With the following attribute:

```
#PB_Explorer_DisplayMode : Changes the display of the gadget.
 Can be one of the following constants:
  - #PB_Explorer_LargeIcon: Large icon mode
```

```
    - #PB_Explorer_SmallIcon: Small icon mode
    - #PB_Explorer_List     : List icon mode
    - #PB_Explorer_Report   : Report mode (columns, default
   mode)
```

- GetGadgetItemAttribute() / SetGadgetItemAttribute() : With the following attribute:

```
#PB_Explorer_ColumnWidth : Returns/Changes the width of the
 given 'Column'. The 'Item' parameter is ignored.
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

```
#PB_Gadget_FrontColor: Textcolor
#PB_Gadget_BackColor : Backgroundcolor
#PB_Gadget_LineColor : Color for the gridlines if the
 #PB_Explorer_GridLines flag is used.
```

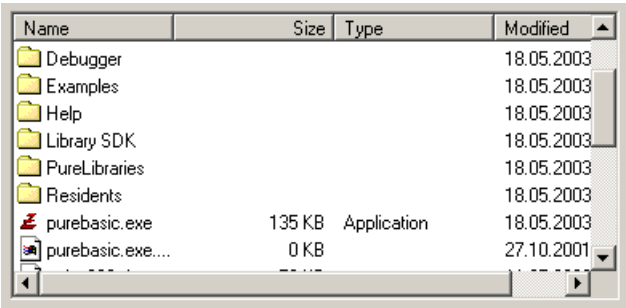
ExplorerListGadget() supports the following events reported by EventType() :

```
#PB_EventType_Change           : The selection or the current
 displayed directory has changed.
#PB_EventType_LeftClick        : User clicked on an item with
 the left mouse button.
#PB_EventType_RightClick       : User clicked on an item with
 the right mouse button.
#PB_EventType_LeftDoubleClick  : User doubleclicked on an
 item with the left mouse button.
#PB_EventType_RightDoubleClick : User doubleclicked on an
 item with the right mouse button.
#PB_EventType_DragStart        : User tried to start a Drag &
 Drop operation.
```

After a #PB_EventType_DragStart event, the Drag & Drop library can be used to start a Drag & Drop operation.

Example:

```
1  If OpenWindow(0, 0, 0, 400, 200, "ExplorerListGadget",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2   ExplorerListGadget(0, 10, 10, 380, 180, "*.*",
    #PB_Explorer_MultiSelect)
3   Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
4  EndIf
```



84.17 ExplorerTreeGadget

Syntax

```
Result = ExplorerTreeGadget(#Gadget, x, y, Width, Height,
   Directory$, [, Flags])
```

Description

Creates a tree listing of the directory tree just as Explorer does. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. It lets the user navigate through his file system, and choose a file or folder.

Directory$ is the directory that will be initially selected. It can include one or multiple patterns, like "C:*.pb;*.pbi". If no pattern is included, the directory must end with a '\'. Including no directory will display the root containing the drives. Including no pattern defaults to '*.*'. So a Directory$ of "" will display the root and set '*.*' as pattern.

'Flags' can be a combination of the following values:

```
#PB_Explorer_BorderLess          : Create Gadget without
 borders.
#PB_Explorer_AlwaysShowSelection : The selection is still
 visible, even when the gadget is not activated.
#PB_Explorer_NoLines             : Hide the little lines
 between each node.
#PB_Explorer_NoButtons           : Hide the '+' node buttons.

#PB_Explorer_NoFiles             : No files will be displayed.
#PB_Explorer_NoDriveRequester    : There will be no 'please
 insert drive X:' displayed.
#PB_Explorer_NoMyDocuments       : The 'My Documents' Folder
 will not be displayed as a separate item.
#PB_Explorer_AutoSort            : The content will be sorted
 automatically by name.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to control the gadget:

- GetGadgetText() : Get the full path of the currently selected directory/file.
- SetGadgetText() : Set the currently selected file/directory.
- GetGadgetState() : Check if the selected item is a file or a directory.

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

```
#PB_Gadget_FrontColor: Textcolor
#PB_Gadget_BackColor : Backgroundcolor
#PB_Gadget_LineColor : Color for the lines and buttons if
 they are displayed
```

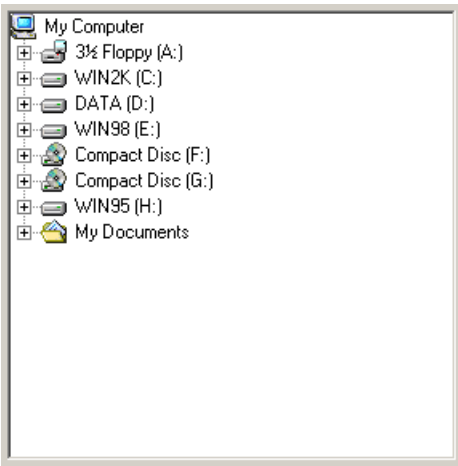
ExplorerTreeGadget() supports the following Events reported by EventType() :

```
#PB_EventType_Change           : The selection has changed.
#PB_EventType_LeftClick        : User clicked on an item with
 the left mouse button.
#PB_EventType_RightClick       : User clicked on an item with
 the right mouse button.
#PB_EventType_LeftDoubleClick  : User doubleclicked on an
 item with the left mouse button.
#PB_EventType_RightDoubleClick : User doubleclicked on an
 item with the right mouse button.
#PB_EventType_DragStart        : User tried to start a Drag &
 Drop operation.
```

After a #PB_EventType_DragStart event, the Drag & Drop library can be used to start a Drag & Drop operation.

Example:

```
If OpenWindow(0, 0, 0, 300, 300, "ExplorerTreeGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ExplorerTreeGadget(0, 10, 10, 280, 280, "*.pb;*.pbi")
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.18 Frame3DGadget

Syntax

```
Result = Frame3DGadget(#Gadget, x, y, Width, Height, Text$ [,
    Flags])
```

Description

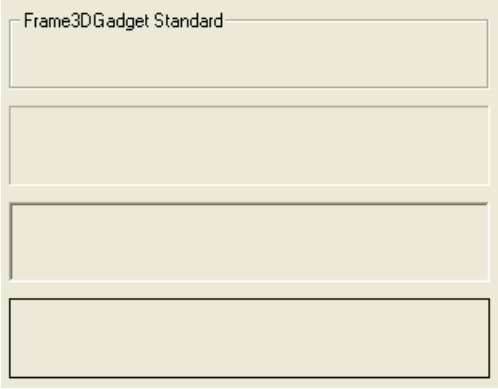
Creates a Frame3D gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. This kind of gadget is decorative only. The Text$ parameter is only valid if no borders are specfied, else it will be ignored. 'Flags' is optional and can be used to change the gadget frame:

```
#PB_Frame3D_Single   : Single sunken frame.
#PB_Frame3D_Double   : Double sunken frame.
#PB_Frame3D_Flat     : Flat frame.
```

Note: As this Gadget is decorative only, GadgetToolTip() cannot be used. This Gadget also receives no events.

Example:

```
If OpenWindow(0, 0, 0, 320, 250, "Frame3DGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  Frame3DGadget(0, 10,  10, 300, 50, "Frame3DGadget Standard")
  Frame3DGadget(1, 10,  70, 300, 50, "", #PB_Frame3D_Single)
  Frame3DGadget(2, 10, 130, 300, 50, "", #PB_Frame3D_Double)
  Frame3DGadget(3, 10, 190, 300, 50, "", #PB_Frame3D_Flat)

  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.19 FreeGadget

Syntax

```
FreeGadget(#Gadget)
```

Description

Free and remove the #Gadget from the display (and free its gadgetlist if the gadget was a container).

Note: all remaining gadgets are automatically freed when the program ends.

84.20 GadgetID

Syntax

```
GadgetID = GadgetID(#Gadget)
```

Description

Returns the unique system identifier of the #Gadget.

This result is sometimes also known as 'Handle'. Look at the extra chapter Handles and Numbers for more information.

84.21 GadgetItemID

Syntax

```
Result = GadgetItemID(#Gadget, Item)
```

Description

Returns the OS handle for the given item in a gadget. This is especially useful for use with the OS API.

This function is currently only supported by TreeGadget() .

Supported OS

Windows

84.22 GadgetToolTip

Syntax

```
GadgetToolTip(#Gadget, Text$)
```

Description

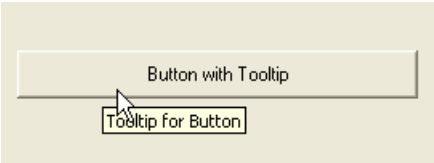
Associate the specified Text$ with the #Gadget. A ToolTip text is text which is displayed when the mouse cursor is over the gadget for a small amount of time (typically a yellow floating box).

The following gadgets are supported:

- ButtonGadget()
- ButtonImageGadget()
- CalendarGadget()
- CheckBoxGadget()
- ComboBoxGadget()
- ContainerGadget()
- DateGadget()
- EditorGadget()
- ExplorerListGadget()
- ExplorerTreeGadget()
- HyperLinkGadget()
- ImageGadget()
- IPAddressGadget()
- ListIconGadget()
- ListViewGadget()
- MDIGadget()
- OptionGadget()
- PanelGadget()
- ProgressBarGadget()
- ScrollBarGadget()
- SpinGadget()
- SplitterGadget()
- StringGadget()
- TrackBarGadget()
- TreeGadget()

Example:

```
1  If OpenWindow(0, 0, 0, 270, 100, "GadgetTooltip",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2    ButtonGadget(0, 10, 30, 250, 30, "Button with Tooltip")
3    GadgetToolTip(0, "Tooltip for Button")
4    Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
5  EndIf
```



84.23 GadgetX

Syntax

```
Result = GadgetX(#Gadget)
```

Description

Returns the X position of the specified #Gadget. The X position is measured in pixels, starting from the inside of the left edge of the gadget list that the gadget is in.

84.24 GadgetY

Syntax

```
Result = GadgetY(#Gadget)
```

Description

Returns the Y position of the specified #Gadget. The Y position is measured in pixels, starting from the inside of the top edge of the gadget list that the gadget is in.

84.25 GadgetHeight

Syntax

```
Result = GadgetHeight(#Gadget)
```

Description

Returns the height of the specified #Gadget, in pixels.

84.26 GadgetType

Syntax

```
Result = GadgetType(#Gadget)
```

Description

Returns the type of gadget that is represented by the specified #Gadget number. It can be useful to write generic functions that work with more than one type of gadget.

It returns one of the following values:

```
#PB_GadgetType_Button        : ButtonGadget()

#PB_GadgetType_ButtonImage   : ButtonImageGadget()

#PB_GadgetType_Calendar      : CalendarGadget()

#PB_GadgetType_CheckBox      : CheckBoxGadget()

#PB_GadgetType_ComboBox      : ComboBoxGadget()

#PB_GadgetType_Container     : ContainerGadget()

#PB_GadgetType_Date          : DateGadget()

#PB_GadgetType_Editor        : EditorGadget()
```

```
#PB_GadgetType_ExplorerCombo : ExplorerComboGadget()

#PB_GadgetType_ExplorerList  : ExplorerListGadget()

#PB_GadgetType_ExplorerTree  : ExplorerTreeGadget()

#PB_GadgetType_Frame3D       : Frame3DGadget()

#PB_GadgetType_HyperLink     : HyperLinkGadget()

#PB_GadgetType_Image         : ImageGadget()

#PB_GadgetType_IPAddress     : IPAddressGadget()

#PB_GadgetType_ListIcon      : ListIconGadget()

#PB_GadgetType_ListView      : ListViewGadget()

#PB_GadgetType_MDI           : MDIGadget()

#PB_GadgetType_Option        : OptionGadget()

#PB_GadgetType_Panel         : PanelGadget()

#PB_GadgetType_ProgressBar   : ProgressBarGadget()

#PB_GadgetType_Scintilla     : ScintillaGadget()

#PB_GadgetType_ScrollArea    : ScrollAreaGadget()

#PB_GadgetType_ScrollBar     : ScrollBarGadget()

#PB_GadgetType_Spin          : SpinGadget()

#PB_GadgetType_Splitter      : SplitterGadget()

#PB_GadgetType_String        : StringGadget()

#PB_GadgetType_Text          : TextGadget()

#PB_GadgetType_TrackBar      : TrackBarGadget()

#PB_GadgetType_Tree          : TreeGadget()

#PB_GadgetType_Web           : WebGadget()

#PB_GadgetType_Unknown       : The type is unknown. Most
 likely it is not a PB gadget at all.
```

84.27 GadgetWidth

Syntax

```
Result = GadgetWidth(#Gadget)
```

Description

Returns the width of the specified #Gadget, in pixels.

84.28 GetActiveGadget

Syntax

```
Result = GetActiveGadget()
```

Description

Returns the #Gadget number of the Gadget that currently has the keyboard focus. If no Gadget has the focus, -1 is returned.

Example:

```
If OpenWindow(0, 0, 0, 270, 70, "GetActiveGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  StringGadget  (0, 10, 10, 250, 20, "Press escape...")
  StringGadget  (1, 10, 40, 250, 20, "Press escape...")
  AddKeyboardShortcut(0, #PB_Shortcut_Escape, 1)
  SetActiveGadget(0)
  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Menu And EventMenu() = 1
      MessageRequester("Test", "Escape pressed in Gadget " +
 Str(GetActiveGadget()))
    EndIf
  Until Event = #PB_Event_CloseWindow
EndIf
```

84.29 GetGadgetAttribute

Syntax

```
Value = GetGadgetAttribute(#Gadget, Attribute)
```

Description

Gets the 'Attribute' value of the specified #Gadget. This function is available for all gadgets which support attributes. See the individual gadget for the supported attributes:

- ButtonImageGadget()
- CalendarGadget()
- DateGadget()
- EditorGadget()
- ExplorerListGadget()
- ListIconGadget()
- PanelGadget()
- ProgressBarGadget()
- ScrollAreaGadget()
- ScrollBarGadget()
- SpinGadget()
- SplitterGadget()
- TrackBarGadget()
- WebGadget()

84.30 GetGadgetColor

Syntax

```
Color = GetGadgetColor(#Gadget, ColorType)
```

Description

Returns the color specified by 'ColorType' from the given #Gadget. This function returns the color that was previously set by SetGadgetColor() . If no custom color is set for the #Gadget and ColorType, the function returns -1.

The following values are possible for the ColorType parameter: (not every gadget supports all of them)

```
  #PB_Gadget_FrontColor      : Gadget text
  #PB_Gadget_BackColor       : Gadget background
  #PB_Gadget_LineColor       : Color for gridlines
  #PB_Gadget_TitleFrontColor: Text color in the title
   (for CalendarGadget()
)
  #PB_Gadget_TitleBackColor : Background color in the title
   (for CalendarGadget()
)
  #PB_Gadget_GrayTextColor  : Color for grayed out text
   (for CalendarGadget()
)
```

This function is supported by the following gadgets: (See each gadget description for the supported ColorType values.)

- CalendarGadget()
- ContainerGadget()
- DateGadget()
- EditorGadget()
- ExplorerListGadget()
- ExplorerTreeGadget()
- HyperLinkGadget()
- ListViewGadget()
- ListIconGadget()
- MDIGadget()
- ProgressBarGadget()
- ScrollAreaGadget()
- SpinGadget()
- StringGadget()
- TextGadget()
- TreeGadget()

84.31 GetGadgetData

Syntax

```
Result = GetGadgetData(#Gadget)
```

Description

Returns the 'Data' value that has been stored for this gadget with the SetGadgetData() function. This allows to associate a custom value with any gadget. If there was never a data value set for this gadget, the returnvalue will be 0.

This function works with all PureBasic gadgets. See SetGadgetData() for an example.

84.32 GetGadgetFont

Syntax

```
FontID = GetGadgetFont(#Gadget)
```

Description

Get the FontID associated with the specified #Gadget. If the #PB_Default constant is used as #Gadget, the default FontID used for new created gadgets is returned. The FontID can be later used with DrawingFont() or similar functions. SetGadgetFont() can be used to change the default gadget font, or affect a font to a specific gadget.

84.33 GetGadgetItemAttribute

Syntax

```
Value = GetGadgetItemAttribute(#Gadget, Item, Attribute [,
    Column])
```

Description

Gets the 'Attribute' value of the specified #Gadget item. This function is available for all gadgets which support item attributes:

- ExplorerListGadget() :

```
#PB_Explorer_ColumnWidth : Returns the width of the given
 'Column'. The 'Item' parameter is ignored.
```

- ListIconGadget() :

```
#PB_ListIcon_ColumnWidth : Returns the width of the given
 'Column'. The 'Item' parameter is ignored.
```

- TreeGadget() :

```
#PB_Tree_SubLevel : Returns the sublevel of the given item
```

The sublevel value of the tree item can be used to determine relations between items in the tree. For example the first item with a sublevel smaller than the current item, if the list is checked backwards, is the parent of the current item.

84.34 GetGadgetItemColor

Syntax

```
Color = GetGadgetItemColor(#Gadget, Item, ColorType [, Column])
```

Description

Returns the color specified by 'ColorType' from the given #Gadget item. This function returns the color that was previously set by SetGadgetItemColor() . If no custom color is set for the #Gadget item and ColorType, the function returns -1.

The following values are possible for the ColorType parameter:

```
#PB_Gadget_FrontColor: Item text
#PB_Gadget_BackColor : Item background
```

This function is supported by the following gadgets:

- ListIconGadget()
- TreeGadget()

84.35 GetGadgetItemData

Syntax

```
Result = GetGadgetItemData(#Gadget, Item)
```

Description

Returns the value that was previously stored with this #Gadget item with the SetGadgetItemData() function. This allows to associate a custom value with the items of a gadget. If no value has been stored with the item yet, the returnvalue will be 0.

This function works with the following gadgets:
- ComboBoxGadget()
- ListIconGadget()
- ListViewGadget()
- TreeGadget()

See SetGadgetItemData() for an example.

84.36 GetGadgetState

Syntax

```
Result = GetGadgetState(#Gadget)
```

Description

Returns the current state of the gadget. This is a universal function which works for almost all gadgets:

- ButtonImageGadget() : returns 1 if a #PB_Button_Toggle button is toggled, else 0.
- ButtonGadget() : returns 1 if a #PB_Button_Toggle button is toggled, else 0.
- CalendarGadget() : returns the currently selected date.
- CheckBoxGadget() : Returns one of the following values:

```
#PB_CheckBox_Checked  : The check mark is set.
#PB_CheckBox_Unchecked: The check mark is not set.
#PB_CheckBox_Inbetween: The "inbetween" state is set. (Only
 for #PB_CheckBox_ThreeState checkboxes)
```

- ComboBoxGadget() : returns the currently selected item index, -1 if none is selected.
- DateGadget() : returns the currently selected date/time. IF #PB_Date_CheckBox was used, and the checkbox is unchecked, 0 is returned.
- ExplorerListGadget() : returns the index of the first selected item in the Gadget, -1 if none is selected.
- ExplorerTreeGadget() : returns the type of the currently selected item (#PB_Explorer_File or #PB_Explorer_Directory).
- ImageGadget() : returns the ImageID of the currently displayed image.
- IPAddressGadget() : returns the current IP address.
- ListIconGadget() : returns the first selected item index, -1 if none is selected.
- ListViewGadget() : returns the currently selected item index, -1 if none is selected.
- MDIGadget() : returns the currently focused child window, -1 if none has the focus.
- OptionGadget() : returns 1 if activated, 0 otherwise.
- PanelGadget() : returns the current panel index, -1 if no panel.
- ProgressBarGadget() : returns the current value of the ProgressBar.
- ScrollBarGadget() : returns the current slider position.
- ShortcutGadget() : returns the currently selected keyboard shortcut.
- SpinGadget() : returns the current value of the SpinGadget.
- SplitterGadget() : returns the current splitter position, in pixels.

- TrackBarGadget() : returns the current position of the TrackBar (value inside the minimum - maximum range).
- TreeGadget() : returns the currently selected item index, -1 if none is selected.

84.37 GetGadgetItemText

Syntax

```
Result\$ = GetGadgetItemText(#Gadget, Item [, Column])
```

Description

Returns the item text of the specified #Gadget. This is a universal function which works for almost all gadgets which handle several items:

- ComboBoxGadget() - 'Item' is the index of the item in the ComboBox list. 'Column' will be ignored.
- EditorGadget() - 'Item' is the text line in the EditorGadget. 'Column' will be ignored.
- ExplorerListGadget() - returns the name of the 'Item', without the path. If 'Item' = -1, the 'Column' header is returned.
- ListIconGadget() - returns the entry of the given 'Item' and 'Column'. If 'Item' = -1, the 'Column' header is returned.
- ListViewGadget() - 'Item' is the index of the entry from which you want to receive the content. 'Column' will be ignored.
- MDIGadget() - 'Item' is the index of the child window of which you want to get the title.
- PanelGadget() - 'Item' is the panel from which you want to receive the header text.
- TreeGadget() - 'Item' is the index of the entry from which you want to receive the content. 'Column' will be ignored.
- WebGadget() - Get the html code, page title, status message or current selection.

Note: All 'Item' index start from 0.

84.38 GetGadgetItemState

Syntax

```
Result = GetGadgetItemState(#Gadget, Item)
```

Description

Returns the item state of the specified #Gadget. This is a universal function that works for almost all gadgets which handle several items:
- CalendarGadget() : returns #PB_Calendar_Bold when the specified date is displayed bold, #PB_Calendar_Normal otherwise. 'Item' must be a PureBasic date value.
- ExplorerListGadget() : returns a combination of the following values

```
#PB_Explorer_File      : The item is a file.
#PB_Explorer_Directory : The item is a Directory (or drive).
#PB_Explorer_Selected  : The item is currently selected.
```

- ListViewGadget() : returns 1 if the item is selected, 0 otherwise.
- ListIconGadget() : returns a combination of the following values:

```
#PB_ListIcon_Selected: The item is selected.
#PB_ListIcon_Checked : The item has its checkbox checked
 (#PB_ListIcon_CheckBoxes flag).
```

- TreeGadget() : returns a combination of the following values:

```
#PB_Tree_Selected  : The item is selected, 0 otherwise.
#PB_Tree_Expanded  : The item is expanded (a tree branch
 opened).
#PB_Tree_Collapsed : The item is collapsed (the tree branch
 closed).
#PB_Tree_Checked   : The Checkbox of the item is checked, 0
 otherwise. (#PB_Tree_CheckBoxes flag)
```

Check for these states like this:

```
If Result & #PB_Tree_Checked
  ; Item is checked
EndIf
```

Below an example with the ListIconGadget() , how you can check for a combination of several results:

```
; ... here a code snippet from a WaitWindowEvent() - event
 loop:
If GetGadgetItemState(#Listicon, n) & #PB_ListIcon_Checked
  ; Item n is checked (no matter if selected)
EndIf

If GetGadgetItemState(#Listicon, n) & #PB_ListIcon_Selected
  ; Item n is selected (no matter if checked)
EndIf

If GetGadgetItemState(#Listicon, n) = #PB_ListIcon_Checked |
 #PB_ListIcon_Selected
  ; Item n is checked AND selected
EndIf

If GetGadgetItemState(#Listicon, n) & #PB_ListIcon_Checked |
 #PB_ListIcon_Selected
  ; Item n is checked OR selected OR both
EndIf
```

84.39 GetGadgetText

Syntax

```
String\$ = GetGadgetText(#Gadget)
```

Description

Returns the gadget text content of the specified #Gadget. This function is especially useful for:

- ButtonGadget() : return the text of the ButtonGadget.
- ComboBoxGadget() - return the content of the current item.
- DateGadget() - return the currently displayed date, in the form it is displayed in the gadget.
- EditorGadget() - return the text content of the editor gadget. Please note, that several lines of text are normaly separated with "Chr(13)+Chr(10)".
- ExplorerComboGadget() - return the currently selected and displayed directory.
- ExplorerListGadget() - return the currently displayed directory.
- ExplorerTreeGadget() - return the full path of the currently selected file/directory.
- HyperLinkGadget() - return the text of the HyperLinkGadget.
- ListIconGadget() - return the content of first column of the currently selected item.
- ListViewGadget() - return the content of the current item.

- StringGadget() - return contents of the StringGadget.
- TextGadget() - return contents of the TextGadget.
- TreeGadget() - return the text of the currently selected Item in the TreeGadget.
- WebGadget() - return the URL of the displayed website.

84.40 HideGadget

Syntax

```
HideGadget(#Gadget, State)
```

Description

Hide or show a gadget. If State = 1, the gadget will be hidden, if State = 0 it will be shown.

Example:

```
If OpenWindow(0, 0, 0, 180, 120, "HideGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ButtonGadget(0, 10, 10, 160, 50, "Button 1")        : button
 = #True    ; Button is displayed
  ButtonGadget(1, 10, 80, 160, 30, "Hide Button 1")
  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Gadget
      If EventGadget() = 1
        If button = #True      ; ButtonGadget is displayed
          HideGadget(0, 1)      ; => hide it
          button = #False
          SetGadgetText(1, "Show Button 1")
        Else                   ; ButtonGadget is hidden
          HideGadget(0, 0)      ; => show it
          button = #True
          SetGadgetText(1, "Hide Button 1")
        EndIf
      EndIf
    EndIf
  Until Event = #PB_Event_CloseWindow
EndIf
```

84.41 HyperLinkGadget

Syntax

```
Result = HyperLinkGadget(#Gadget, x, y, Width, Height, Text$,
    Color [, Flags])
```

Description

Creates an HyperLink gadget in the current GadgetList. A hyperlink gadget is a text area which reacts to the mouse pointer by changing its color and the cursor shape. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'.

The content can be changed with the function SetGadgetText() and can be received with GetGadgetText() .

'Flags' is always optional and can be the following constant:

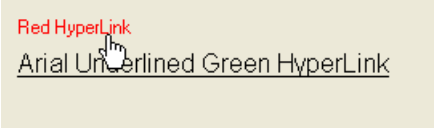
```
#PB_Hyperlink_Underline: Draw a line under the text without
 the need to use an underlined font.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the #PB_Gadget_FrontColor type to change the text color. The backgroundcolor is always the color of the window.

Example:

```
1  If OpenWindow(0, 0, 0, 270, 160, "HyperlinkGadget",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2   HyperLinkGadget(0, 10, 10, 250,20,"Red HyperLink",
    RGB(255,0,0))
3   HyperLinkGadget(1, 10, 30, 250,20,"Arial Underlined Green
    HyperLink", RGB(0,255,0), #PB_HyperLink_Underline)
4   SetGadgetFont(1, LoadFont(0, "Arial", 12))
5   Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
6  EndIf
```



84.42 ImageGadget

Syntax

```
Result = ImageGadget(#Gadget, x, y, Width, Height, ImageID [,
    Flags])
```

Description

Creates an Image gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. The ImageID represents an image and can be obtained with ImageID() from the Image library. If the ImageID is 0, then no image will be displayed. The gadget dimensions autofit to the image size.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

- SetGadgetState() : Change the current Image of the gadget. A valid ImageID can be easily obtained with the ImageID() function. If the ImageID is 0, then the image is removed from the gadget.

The following events are supported through EventType() :

```
#PB_EventType_LeftClick
#PB_EventType_RightClick
#PB_EventType_LeftDoubleClick
#PB_EventType_RightDoubleClick
#PB_EventType_DragStart
```

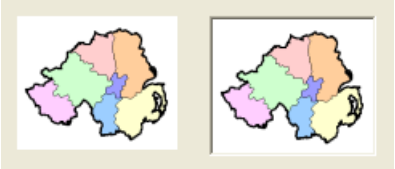
After a #PB_EventType_DragStart event, the Drag & Drop library can be used to start a Drag & Drop operation.

'Flags' are always optional and can be composed of one of the following constants:

```
#PB_Image_Border : display a sunken border around the image.
```

Example:

```
If OpenWindow(0, 0, 0, 245, 105, "ImageGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If LoadImage(0, "map.bmp")     ; change 2nd parameter to the
 path/filename of your image
     ImageGadget(0,  10, 10, 100, 83, ImageID(0))
                          ; imagegadget standard
     ImageGadget(1, 130, 10, 100, 83, ImageID(0),
 #PB_Image_Border)      ; imagegadget with border
  EndIf
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.43 IPAddressGadget

Syntax

```
Result = IPAddressGadget(#Gadget, x, y, Width, Height)
```

Description

Creates an IPAddress gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. It allows you to easily enter a full IPv4 address. #Gadget will be the numeric number returned by the EventGadget() function.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on this gadget:

- GetGadgetState() : Returns the current IP address (Use IPAddressField() to get the value of each field).
- SetGadgetState() : Changes the current IP address (Use MakeIPAddress() to build a valid IP address).
- GetGadgetText() : Returns the current IP address as text, in decimal dotted form ("127.0.0.1", for example).
- SetGadgetText() : Only used to clear the IP address contents, by passing an empty string.

Example:

```
If OpenWindow(0, 0, 0, 180, 50, "IPAddressGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  IPAddressGadget(0, 10, 15, 160, 20)
  SetGadgetState(0, MakeIPAddress(127, 0, 0, 1))    ; set a
 valid ip address
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

84.44 IsGadget

Syntax

```
Result = IsGadget(#Gadget)
```

Description

Tests if the given #Gadget number is a valid and correctly initialized gadget. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a gadget is ready to use.

84.45 ListIconGadget

Syntax

```
Result = ListIconGadget(#Gadget, x, y, Width, Height, Title$,
    TitleWidth [, Flags])
```

Description

Creates a ListIcon gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by the EventGadget() function. 'Title$' is the first column title, and 'TitleWidth' its initial width (in pixels). Once a ListIconGadget is created, its list of items is empty.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on the list content:

- AddGadgetColumn() : Add a column to the gadget.
- RemoveGadgetColumn() : Remove a column from the gadget.
- AddGadgetItem() : Add an item (with an optional image in the standard 16x16 icon size).
- RemoveGadgetItem() : Remove an item.
- ClearGadgetItems() : Remove all the items.
- CountGadgetItems() : Returns the number of items currently in the #Gadget.

- GetGadgetItemColor() : Returns front or backcolor of the item.
- SetGadgetItemColor() : Changes front or backcolor of the item.
- GetGadgetItemData() : Returns the value that was stored with item.
- SetGadgetItemData() : Stores a value with the item.
- GetGadgetItemState() : Returns the current state of the specified item.
- SetGadgetItemState() : Changes the current state of the specified item.
- GetGadgetItemText() : Returns the current text of the specified item. (or column header, if item = -1)
- SetGadgetItemText() : Changes the current text of the specified item. (or column header, if item = -1)
- GetGadgetState() : Returns the first selected item or -1 if there is no item selected.
- SetGadgetState() : Change the selected item (all other selected items will be deselected). If -1 is specified, no more item will be selected.
- GetGadgetAttribute() / SetGadgetAttribute() : With the following attribute:

```
#PB_ListIcon_DisplayMode : Changes the display of the gadget.
 Can be one of the following constants (Windows only):
  - #PB_ListIcon_LargeIcon: Large icon mode
  - #PB_ListIcon_SmallIcon: Small icon mode
  - #PB_ListIcon_List     : List icon mode
  - #PB_ListIcon_Report   : Report mode (columns, default
 mode)
```

- GetGadgetItemAttribute() / SetGadgetItemAttribute() : With the following attribute:

```
#PB_ListIcon_ColumnWidth : Returns/Changes the width of the
 given 'Column'. The 'Item' parameter is ignored.
```

If you want to add content to ListIcons with several columns using AddGadgetItem() , you should use the format "first column"+Chr(10)+"second one" as Text$ parameter. See also the following example.

'Flags' are always optional and can be composed of one of the following constants:

```
#PB_ListIcon_CheckBoxes          : Display checkboxes in the
 first column.
#PB_ListIcon_MultiSelect         : Enable multiple selection.
#PB_ListIcon_GridLines           : Display separator lines
 between rows and columns (not supported on Mac OSX).
#PB_ListIcon_FullRowSelect       : The selection covers the
 full row instead of the first column (Windows only).
#PB_ListIcon_HeaderDragDrop      : The order of columns can be
 changed using drag'n'drop.
#PB_ListIcon_AlwaysShowSelection: The selection is still
 visible, even when the gadget is not activated (Windows
 only).
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

```
#PB_Gadget_FrontColor: Textcolor
#PB_Gadget_BackColor : Backgroundcolor
#PB_Gadget_LineColor : Color for the gridlines if the
 #PB_ListIcon_GridLines flag is used.
```

The following events are supported through EventType() :

```
#PB_EventType_LeftClick: left click on an item, or a checkbox
 was checked/unchecked
#PB_EventType_LeftDoubleClick
#PB_EventType_RightClick
#PB_EventType_RightDoubleClick
#PB_EventType_Change: the current item changed
#PB_EventType_DragStart: the user tried to start a Drag &
 Drop operation.
```

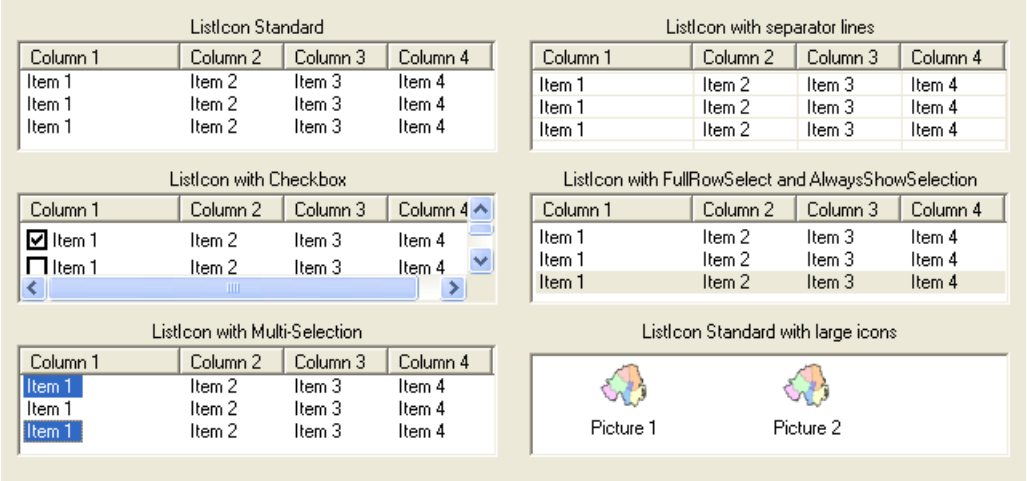
After a #PB_EventType_DragStart event, the Drag & Drop library can be used to start a Drag & Drop operation.

Example 1:

```
If OpenWindow(0, 100, 100, 300, 100, "ListIcon Example",
  #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ListIconGadget(0, 5, 5, 290, 90, "Name", 100,
  #PB_ListIcon_FullRowSelect|#PB_ListIcon_AlwaysShowSelection)
  AddGadgetColumn(0, 1, "Address", 250)
  AddGadgetItem(0, -1, "Harry Rannit"+Chr(10)+"12 Parliament
  Way, Battle Street, By the Bay")
  AddGadgetItem(0, -1, "Ginger Brokeit"+Chr(10)+"130 PureBasic
  Road, BigTown, CodeCity")
  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```

Example 2:

```
; Shows possible flags of ListIconGadget in action...
If OpenWindow(0, 0, 0, 640, 300, "ListIconGadgets", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ; left column
  TextGadget     (6,  10,  10, 300, 20, "ListIcon Standard", #PB_Text_Center)
  ListIconGadget(0,  10,  25, 300, 70, "Column 1", 100)
  TextGadget     (7,  10, 105, 300, 20, "ListIcon with Checkbox", #PB_Text_Center)
  ListIconGadget(1,  10, 120, 300, 70, "Column 1", 100, #PB_ListIcon_CheckBoxes)  ; ListIcon with checkbox
  TextGadget     (8,  10, 200, 300, 20, "ListIcon with Multi-Selection", #PB_Text_Center)
  ListIconGadget(2,  10, 215, 300, 70, "Column 1", 100, #PB_ListIcon_MultiSelect) ; ListIcon with multi-selection
  ; right column
  TextGadget     (9, 330,  10, 300, 20, "ListIcon with separator lines",#PB_Text_Center)
  ListIconGadget(3, 330,  25, 300, 70, "Column 1", 100, #PB_ListIcon_GridLines)
  TextGadget    (10, 330, 105, 300, 20, "ListIcon with FullRowSelect and AlwaysShowSelection",#PB_Text_Center)
  ListIconGadget(4, 330, 120, 300, 70, "Column 1", 100, #PB_ListIcon_FullRowSelect|#PB_ListIcon_AlwaysShowSelection)
  TextGadget    (11, 330, 200, 300, 20, "ListIcon Standard with large icons",#PB_Text_Center)
  ListIconGadget(5, 330, 220, 300, 65, "", 200,#PB_ListIcon_GridLines)
  For a = 0 To 4             ; add columns to each of the first 5 listicons
    For b = 2 To 4           ; add 3 more columns to each listicon
      AddGadgetColumn(a, b, "Column " + Str(b), 65)
    Next
    For b = 0 To 2           ; add 4 items to each line of the listicons
      AddGadgetItem(a, b, "Item 1"+Chr(10)+"Item 2"+Chr(10)+"Item 3"+Chr(10)+"Item 4")
    Next
  Next
  ; Here we change the ListIcon display to large icons and show an image
  If LoadImage(0, "map2.bmp")      ; change path/filename to your own 32x32 pixel image
    SetGadgetAttribute(5, #PB_ListIcon_DisplayMode, #PB_ListIcon_LargeIcon)
    AddGadgetItem(5, 1, "Picture 1", ImageID(0))
    AddGadgetItem(5, 2, "Picture 2", ImageID(0))
  EndIf
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.46 ListViewGadget

Syntax

```
Result = ListViewGadget(#Gadget, x, y, Width, Height [, Flags])
```

Description

Creates a ListView gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by the EventGadget() function. Once a ListView is created, its list of items is empty.

'Flags' are always optional and can be composed of one of the following constants:

```
#PB_ListView_Multiselect: allows multiple items to be selected
#PB_ListView_ClickSelect: allows multiple items to be
 selected. clicking on one item selects/deselects it
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on the list content:

- AddGadgetItem() : Add an item.
- RemoveGadgetItem() : Remove an item.
- ClearGadgetItems() : Remove all the items
- CountGadgetItems() : Returns the number of items currently in the #Gadget.
- GetGadgetItemData() : Get the value that was stored with the gadget item.
- GetGadgetItemState() : Returns nonzero if the item is selected, zero otherwise.
- GetGadgetItemText() : Get the content of the given item.
- GetGadgetState() : Get the index of the selected item or -1 if there is no selected item.
- GetGadgetText() : Get the content of the selected item.
- SetGadgetItemData() : store a value with the given item.
- SetGadgetItemState() : Selects or deselects the given item.
- SetGadgetItemText() : Set the text of the given item.
- SetGadgetState() : Change the selected item. If -1 is specified, the selection will be removed.
- SetGadgetText() : Selects the item with the given text (the text must match exactly).

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

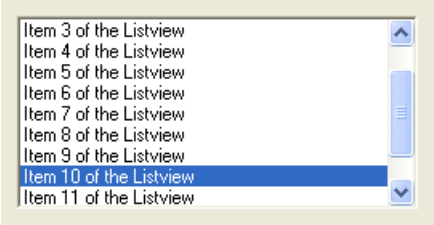
```
#PB_Gadget_FrontColor: Textcolor
#PB_Gadget_BackColor : Backgroundcolor
```

The following events are supported through EventType() :

```
#PB_EventType_LeftClick
#PB_EventType_LeftDoubleClick
```

Example:

```
If OpenWindow(0, 0, 0, 270, 140, "ListViewGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ListViewGadget(0, 10, 10, 250, 120)
  For a = 1 To 12
    AddGadgetItem (0, -1, "Item " + Str(a) + " of the
 Listview") ; define listview content
  Next
  SetGadgetState(0, 9) ; set (beginning with 0) the tenth
 item as the active one
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.47 MDIGadget

Syntax

```
Result = MDIGadget(#Gadget, x, y, Width, Height, SubMenu,
   MenuItem [, Flags])
```

Description

Creates a client area, in which child windows can be displayed. These child windows are fully movable and sizable by the user in this area. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'.

An MDIGadget() is always connected to a window Menu (see CreateMenu()). Therefore a MDIGadget() can only be put on a window that has a menu attached to it. The gadget will give the user the opportunity to select the child windows through one of the windows submenus. In the 'SubMenu' parameter, you have to specify the submenu index (created with MenuTitle()) where these items will be attached to (the first submenu has the index 0). The Gadget will add a seperator at the end of this menu, and then add one item for each currently displayed child window.

The gadget will require a set of Menu item identifiers (see the 'MenuID' parameter of MenuItem()) to add these menuitems. In the 'MenuItem' parameter of MDIGadget() you have to specify the lowest number that the gadget can use for that purpose. It will use numbers above that, when new child windows are added, so you need to reserve at least as much numbers as you plan to add items. It is recommended to use a number above all menu identifiers of your program, to make sure there is never a collision.

Additional notes:
Because of the connection with the window menu, there can only be one MDIGadget() on a window, however you can put another one in a second window if you wish. You can only put

this gadget directly on a window, you can NOT put it inside a ContainerGadget() , SplitterGadget() or PanelGadget() .
As the whole point of this gadget is to dynamically display data, it is recommended to use the #PB_Any feature to populate a child window with gadgets.

When using AddGadgetItem() with this gadget, there is actually a new PB window created. All the functions of the Window library can be used with this new window (except StickyWindow()). Of course the number choosen for the new window may not overlap with another open window, otherwise the other window will be closed. The MDI Gadget does not return any events. Events concerning the childwindows will be received as normal window events (#PB_Event_SizeWindow, #PB_Event_CloseWindow, ...) instead.

The following values can be used in the 'Flags' parameter:

```
#PB_MDI_AutoSize        : The gadget will automatically resize
 itself to fit the parent window.
                          If you have no other gadgets on the
 window, this is a helpfull option.
#PB_MDI_BorderLess      : There will be no border drawn around
 the client area.
#PB_MDI_NoScrollBars    : When the user drags a childwindow
 outside of the displayed area, there will be no scrollbars.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to manage the gadget contents:

- CountGadgetItems() : Return the number of child windows.
- AddGadgetItem() : Add a new child window to the gadget.
- ClearGadgetItems() : Close all child windows.
- GetGadgetState() : Get the currently focused child window.
- SetGadgetState() : Set the currently focused window, or arrange the child windows. (see GetGadgetState() for more info.)
- SetGadgetAttribute() : With the following attributes:

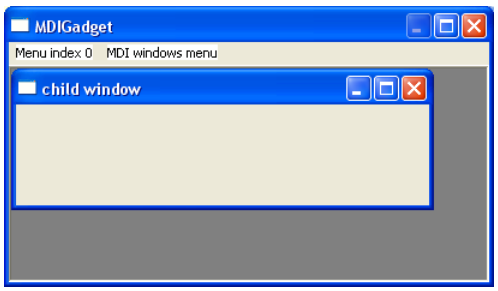
```
#PB_MDI_Image     : Set a background image. An ImageID value
 must be passed as value. (see ImageID()
)
#PB_MDI_TileImage: Set the tile mode. 0 draws the image just
 once in the top/left corner, 1 repeats the image to fill the
 whole area
```

This gadget supports the SetGadgetColor() and GetGadgetColor functions with the #PB_Gadget_BackColor type to change the background color of the MDI area.

Example:

```
#Main = 0
#MDIChild = 1
If OpenWindow(#Main, 0, 0, 400, 300, "MDIGadget",
 #PB_Window_SystemMenu|#PB_Window_ScreenCentered|#PB_Window_SizeGadget|#PB_Win
  If CreateMenu(#Main, WindowID(#Main))
    MenuTitle("Menu index 0")
    MenuTitle("MDI windows menu")
      MenuItem(0, "self created item")
      MenuItem(1, "self created item")

    MDIGadget(0, 0, 0, 0, 0, 1, 2, #PB_MDI_AutoSize)
      AddGadgetItem(0, #MDIChild, "child window")
        ; add gadgets here...
    UseGadgetList(WindowID(#Main)) ; go back to the main
 window gadgetlist
  EndIf
```

```
    Repeat : Until WaitWindowEvent()=#PB_Event_CloseWindow
  EndIf
```



Supported OS

Windows

84.48 OpenGadgetList

Syntax

```
OpenGadgetList(#Gadget [, Item])
```

Description

Use the specified #Gadget as a GadgetList, to dynamically add new gadgets to it. Once the all the needed changes are done, CloseGadgetList() should be called. The following gadgets are supported by OpenGadgetList():

- ContainerGadget()
- PanelGadget() : 'Item' need to be specified when adding a gadget on a specific panel. To add a new panel dynamically to the PanelGadget() the 'Item' parameter must be omitted.
- ScrollAreaGadget()

84.49 OptionGadget

Syntax

```
Result = OptionGadget(#Gadget, x, y, Width, Height, Text$)
```

Description

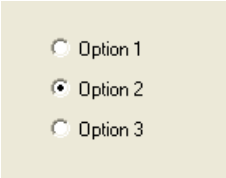
Creates an OptionGadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. The first time this function is called, a group is created and all following calls of OptionGadget() will add a gadget to this group. To finish the group, just create a gadget of another type. These kind of gadgets are very useful as only one gadget from the group can be selected at any time.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

Example:

```
  If OpenWindow(0, 0, 0, 140, 110, "OptionGadget",
   #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
    OptionGadget(0, 30, 20, 60, 20, "Option 1")
    OptionGadget(1, 30, 45, 60, 20, "Option 2")
```

```
    OptionGadget(2, 30, 70, 60, 20, "Option 3")
    SetGadgetState(1, 1)    ; set second option as active one
    Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```



84.50 PanelGadget

Syntax

```
Result = PanelGadget(#Gadget, x, y, Width, Height)
```

Description

Creates a Panel gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. Once a Panel is created, its list of panels is empty.

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on the list content:

- AddGadgetItem() : Add a panel.
- RemoveGadgetItem() : Remove a panel.
- CountGadgetItems() : Count the number of panels.
- ClearGadgetItems() : Remove all panels.
- GetGadgetItemText() : Retrieve the text of the specified item.
- SetGadgetItemText() : Change the text of the specified item.

- SetGadgetState() : Change the active panel.
- GetGadgetState() : Get the index of the current panel.
- GetGadgetAttribute() : With one of the following attributes: (there must be at least 1 item for this to work)

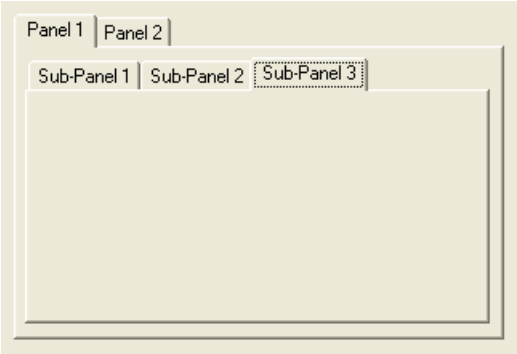
```
#PB_Panel_ItemWidth : Returns the width of the inner area
 where the gadgets are displayed.
#PB_Panel_ItemHeight: Returns the height of the inner area
 where the gadgets are displayed.
#PB_Panel_TabHeight : Returns height of the panel tabs on top
 of the gadget.
```

You must call AddGadgetItem() to add a panel before you can create other gadgets inside the panel gadget. The next gadgets will then be created automatically in the new panel. This is very convenient. When the PanelGadget item has been filled with all the needed gadgets, CloseGadgetList() must be called to return to the previous GadgetList. This means that a PanelGadget can be created inside another PanelGadget...

Example:

```
; Shows using of several panels...
If OpenWindow(0, 0, 0, 322, 220, "PanelGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  PanelGadget     (0, 8, 8, 306, 203)
    AddGadgetItem (0, -1, "Panel 1")
```

```
        PanelGadget (1, 5, 5, 290, 166)
          AddGadgetItem(1, -1, "Sub-Panel 1")
          AddGadgetItem(1, -1, "Sub-Panel 2")
          AddGadgetItem(1, -1, "Sub-Panel 3")
        CloseGadgetList()
      AddGadgetItem (0, -1,"Panel 2")
        ButtonGadget(2, 10, 15, 80, 24,"Button 1")
        ButtonGadget(3, 95, 15, 80, 24,"Button 2")
    CloseGadgetList()
    Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```



84.51 ProgressBarGadget

Syntax

```
Result = ProgressBarGadget(#Gadget, x, y, Width, Height,
    Minimum, Maximum [, Flags])
```

Description

Creates a ProgressBar gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. The possible values that the progress bar can take are fixed between the Minimum and Maximum one.

'Flags' are are always optional and can be composed of one of the following constants:

```
#PB_ProgressBar_Smooth    : The progress bar is smooth instead
  of using blocks
                            (Note: On Windows XP with enabled
  skins, this coloring has no effect).
#PB_ProgressBar_Vertical : The progress bar is in vertical
  mode.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on the gadget:

- SetGadgetState() : Change progress bar value.
- GetGadgetState() : Get the current progress bar value.
- SetGadgetAttribute() : With the following attributes:

```
#PB_ProgressBar_Minimum    : Changes the minimum value.
#PB_ProgressBar_Maximum    : Changes the maximum value.
```

- GetGadgetAttribute() : With the following attributes:

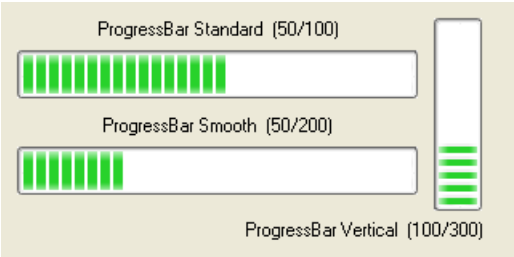
```
#PB_ProgressBar_Minimum    : Returns the minimum value.
#PB_ProgressBar_Maximum    : Returns the maximum value.
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType'. (Note: On Windows XP with enabled skins, this coloring has no effect.)

```
#PB_Gadget_FrontColor: Progressbarcolor
#PB_Gadget_BackColor : Backgroundcolor
```

Example:

```
If OpenWindow(0, 0, 0, 320, 160, "ProgressBarGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  TextGadget        (3,  10, 10, 250,  20, "ProgressBar
 Standard  (50/100)", #PB_Text_Center)
  ProgressBarGadget(0,  10, 30, 250,  30, 0, 100)
  SetGadgetState    (0, 50)   ;  set 1st progressbar (ID = 0)
 to 50 of 100
  TextGadget        (4,  10, 70, 250,  20, "ProgressBar Smooth
  (50/200)", #PB_Text_Center)
  ProgressBarGadget(1,  10, 90, 250,  30, 0, 200,
 #PB_ProgressBar_Smooth)
  SetGadgetState    (1, 50)   ;  set 2nd progressbar (ID = 1)
 to 50 of 200
  TextGadget        (5, 100,135, 200,  20, "ProgressBar
 Vertical  (100/300)", #PB_Text_Right)
  ProgressBarGadget(2, 270, 10,  30, 120, 0, 300,
 #PB_ProgressBar_Vertical)
  SetGadgetState    (2, 100)   ; set 3rd progressbar (ID = 2)
 to 100 of 300
  Repeat : Until WaitWindowEvent()=#PB_Event_CloseWindow
EndIf
```



84.52 RemoveGadgetColumn

Syntax

```
RemoveGadgetColumn(#Gadget, Column)
```

Description

Removes a column of the specified #Gadget. Column index starts at 0. The gadget type can be one of the following:

- ListIconGadget()
- ExplorerListGadget()

Example:

```
If OpenWindow(0, 0, 0, 320, 160, "RemoveGadgetColumn",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)

  ListIconGadget(0, 10, 10, 300, 140, "Hello", 100)
    AddGadgetColumn(0, 1, "Column 2", 70)
    AddGadgetColumn(0, 2, "Column 3", 70)

  RemoveGadgetColumn(0, 1) ; Remove the 'Column 2'

  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

84.53 RemoveGadgetItem

Syntax

```
RemoveGadgetItem(#Gadget, Position)
```

Description

Removes an item of the specified #Gadget at the given Position. Position index starts at 0. The gadget type can be one of the following:

- ComboBoxGadget()
- EditorGadget()
- PanelGadget()
- ListViewGadget()
- ListIconGadget()
- MDIGadget()
- TreeGadget() - If the removed Item is a Node, all Child Items will be removed too.

84.54 ResizeGadget

Syntax

```
ResizeGadget(#Gadget, x, y, Width, Height)
```

Description

Resize the specified #Gadget to the given position and dimensions. To ease the building of realtime resizeable Graphical User Interfaces (GUIs), #PB_Ignore can be passed as any parameter (x, y, Width or Height) and this parameter will not be changed. This is very handy because often the gadget should only change in width or in height.

```
[...]

ResizeGadget(0, #PB_Ignore, #PB_Ignore, 300, #PB_Ignore) ;
 Only change the gadget width.

ResizeGadget(0, 150, 100, #PB_Ignore, #PB_Ignore) ; Only move
 the gadget to a new position.
```

84.55 ScrollBarGadget

Syntax

```
Result = ScrollBarGadget(#Gadget, x, y, Width, Height, Minimum,
    Maximum, PageLength [, Flags])
```

Description

Creates a scrollbar gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. It's widely used when displaying only a part of an object. The Minimum-Maximum range should be between 0 and 10,000. The 'PageLength' is the number which defines a page. For example you can have a picture which is 100 pixels width and you only see 25 pixels. What you see is a called a 'page', in this example, the page length will be 25, the Mininum will be 0 and the Maximum will be 100. #Gadget will be the number returned by EventGadget() function.

'Flags' are optionals and can be composed of one of the following constants:

```
#PB_ScrollBar_Vertical : The scrollbar is vertical (instead
 of horizontal, which is the default).
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on this gadget:

- GetGadgetState() : Returns the current slider position (value between the Minimum-Maximum range).
- SetGadgetState() : Changes the current slider position.
- GetGadgetAttribute() : With one of the following attributes:

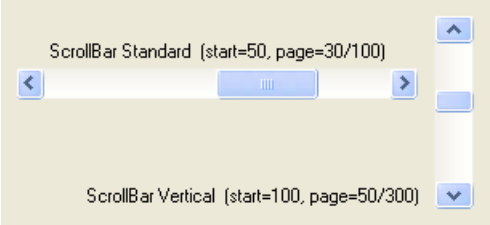
```
#PB_ScrollBar_Minimum   : Returns the minimum scroll position.
#PB_ScrollBar_Maximum   : Returns the maximum scroll position.
#PB_ScrollBar_PageLength: Returns the PageLength value.
```

- SetGadgetAttribute() : With one of the following attributes:

```
#PB_ScrollBar_Minimum   : Changes the minimum scroll position.
#PB_ScrollBar_Maximum   : Changes the maximum scroll position.
#PB_ScrollBar_PageLength: Changes the PageLength value.
```

Example:

```
1  If OpenWindow(0, 0, 0, 305, 140, "ScrollBarGadget",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2   TextGadget       (2,  10, 25, 250,  20, "ScrollBar Standard
    (start=50, page=30/100)",#PB_Text_Center)
3   ScrollBarGadget  (0,  10, 42, 250,  20, 0, 100, 30)
4   SetGadgetState   (0,  50)   ; set 1st scrollbar (ID = 0) to
    50 of 100
5   TextGadget       (3,  10,115, 250,  20, "ScrollBar Vertical
    (start=100, page=50/300)",#PB_Text_Right)
6   ScrollBarGadget  (1, 270, 10,  25, 120 ,0, 300, 50,
    #PB_ScrollBar_Vertical)
7   SetGadgetState   (1, 100)   ; set 2nd scrollbar (ID = 1) to
    100 of 300
8   Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
9  EndIf
```



84.56 ScrollAreaGadget

Syntax

```
Result = ScrollAreaGadget(#Gadget, x, y, Width, Height,
   ScrollAreaWidth, ScrollAreaHeight [, ScrollStep [, Flags]])
```

Description

Creates a ScrollArea gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. It's very useful when a gadget is too big to fit the window dimension. In that case, it can be put into a scrollarea. All the scrolling is handled automatically by the gadget. This is a container gadget, intended to have one or several gadget in its scroll area. Once the gadget is created, all future created gadgets will be created inside the scoll area. When all the needed gadgets have been created, CloseGadgetList() must be called to return to the previous GadgetList. OpenGadgetList() can be used later to add others gadgets on the fly in the scroll area.

'ScrollAreaWidth' and 'ScrollAreaHeight' are the initial dimensions of the scrollable area contained inside the gadget and should be greater than the gadget's 'Width' and 'Height' dimensions. 'ScrollStep' is the number of pixels scrolled when pressing a slider arrow. This parameter only has an effect on Windows. #Gadget will be the number returned by EventGadget() function. When the user moves the scrollbars in the gadget, an event will be received for the gadget.

'Flags' are optional and can be composed of one of the following constants:

```
#PB_ScrollArea_Flat        : Flat frame
#PB_ScrollArea_Raised      : Raised frame
#PB_ScrollArea_Single      : Single sunken frame
#PB_ScrollArea_BorderLess  : Without any border
#PB_ScrollArea_Center      : If the inner size is smaller than
 the outer, the inner area is automatically centered.
```

The following functions can be used to act on a ScrollAreaGadget:

GetGadgetAttribute() : With one of the following attribute:

```
#PB_ScrollArea_InnerWidth  : Returns the width (in pixels) of
 the contained scrollable area.
#PB_ScrollArea_InnerHeight : Returns the height (in pixels)
 of the contained scrollable area.
#PB_ScrollArea_X           : Returns the current horizontal
 scrolling position (in pixels).
#PB_ScrollArea_Y           : Returns the current vertical
 scrolling position (in pixels).
```

SetGadgetAttribute() : With one of the following attribute:

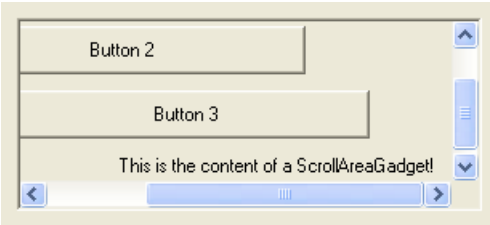
```
#PB_ScrollArea_InnerWidth  : Changes the width (in pixels) of
 the contained scrollable area.
#PB_ScrollArea_InnerHeight : Changes the height (in pixels)
 of the contained scrollable area.
```

```
#PB_ScrollArea_X            : Changes the current horizontal
 scrolling position (in pixels).
#PB_ScrollArea_Y            : Changes the current vertical
 scrolling position (in pixels).
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the #PB_Gadget_BackColor type to change the background color.

Example:

```
If OpenWindow(0, 0, 0, 305, 140, "ScrollAreaGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ScrollAreaGadget(0, 10, 10, 290,120, 375, 155, 30)
    ButtonGadget  (1, 10, 10, 230, 30,"Button 1")
    ButtonGadget  (2, 50, 50, 230, 30,"Button 2")
    ButtonGadget  (3, 90, 90, 230, 30,"Button 3")
    TextGadget    (4,130,130, 230, 20,"This is the content of
 a ScrollAreaGadget!",#PB_Text_Right)
    CloseGadgetList()
  Repeat
    Select WaitWindowEvent()
      Case  #PB_Event_CloseWindow
        End
      Case  #PB_Event_Gadget
        Select EventGadget()
          Case 1
            MessageRequester("Info","Button 1 was
 pressed!",#PB_MessageRequester_Ok)
          Case 2
            MessageRequester("Info","Button 2 was
 pressed!",#PB_MessageRequester_Ok)
          Case 3
            MessageRequester("Info","Button 3 was
 pressed!",#PB_MessageRequester_Ok)
        EndSelect
    EndSelect
  ForEver
EndIf
```



84.57 SetActiveGadget

Syntax

```
SetActiveGadget(#Gadget)
```

Description

Activates (sets the focus on) the gadget specified by the given #Gadget number. This is mainly used with ComboBoxGadget() and StringGadget() . Activating a gadget allows it to become the current object to receive messages and handle keystrokes.

Example:

```
If OpenWindow(0, 0, 0, 270, 140, "SetActiveGadget", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  StringGadget   (0, 10, 10, 250, 20, "bla bla...")
  ComboBoxGadget(1, 10, 40, 250, 21)
  For a = 1 To 5 : AddGadgetItem(1, -1, "ComboBox item " + Str(a)) : Next
  SetGadgetState(1, 2)                  ; set (beginning with 0) the third item as active one
  ButtonGadget   (2, 10,  90, 250, 20, "Activate StringGadget")
  ButtonGadget   (3, 10, 115, 250, 20, "Activate ComboBox")
  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Gadget
      Select EventGadget()
        Case 2 : SetActiveGadget(0)    ; Activate StringGadget
        Case 3 : SetActiveGadget(1)    ; Activate ComboBoxGadget
      EndSelect
    EndIf
  Until Event = #PB_Event_CloseWindow
EndIf
```

84.58 SetGadgetAttribute

Syntax

```
SetGadgetAttribute(#Gadget, Attribute, Value)
```

Description

Changes the 'Attribute' value of the specified #Gadget. This function is available for all gadgets which support attributes:

- ButtonImageGadget()
- CalendarGadget()
- DateGadget()
- EditorGadget()
- ExplorerListGadget()
- ListIconGadget()
- MDIGadget()
- ProgressBarGadget()
- ScrollAreaGadget()
- ScrollBarGadget()
- SpinGadget()
- SplitterGadget()
- TrackBarGadget()
- WebGadget()

84.59 SetGadgetColor

Syntax

```
SetGadgetColor(#Gadget, ColorType, Color)
```

Description

Changes the color attribute specified by 'ColorType' of the given #Gadget. RGB() can be used to get a valid color value. To remove the custom color and go back to the default system color, set the 'Color' parameter to -1.

The following values are possible for the ColorType parameter: (not every gadget supports all of them)

```
  #PB_Gadget_FrontColor      : Gadget text
  #PB_Gadget_BackColor       : Gadget background
  #PB_Gadget_LineColor       : Color for gridlines
  #PB_Gadget_TitleFrontColor: Text color in the title
   (for CalendarGadget()
)
  #PB_Gadget_TitleBackColor : Background color in the title
   (for CalendarGadget()
)
  #PB_Gadget_GrayTextColor  : Color for grayed out text
   (for CalendarGadget()
)
```

This function is supported by the following gadgets: (see each gadget description for the supported ColorType values.)

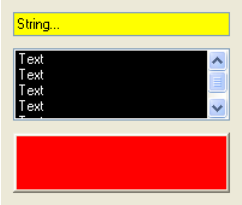
- CalendarGadget()
- ContainerGadget()
- DateGadget()
- EditorGadget()
- ExplorerListGadget()
- ExplorerTreeGadget()
- HyperLinkGadget()
- ListViewGadget()
- ListIconGadget()
- MDIGadget()
- ProgressBarGadget()
- ScrollAreaGadget()
- SpinGadget()
- StringGadget()
- TextGadget()
- TreeGadget()

Note: With activated Windows XP style the color settings will probably be ignored or overwritten by the style.

Example:

```
1   If OpenWindow(0, 0, 0, 200, 170, "SetGadgetColor",
     #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2     StringGadget(0, 10, 10, 180, 20, "String...")
3     ListViewGadget(1, 10, 40, 180, 60)
4     For i = 0 To 4
5       AddGadgetItem(1, -1, "Text")
6     Next i
7     ContainerGadget(2, 10, 110, 180, 50, #PB_Container_Raised)
8
9     SetGadgetColor(0, #PB_Gadget_BackColor, $00FFFF)
10    SetGadgetColor(1, #PB_Gadget_FrontColor, $FFFFFF)
11    SetGadgetColor(1, #PB_Gadget_BackColor, $000000)
12    SetGadgetColor(2, #PB_Gadget_BackColor, $0000FF)
13
```

```
    Repeat
    Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```



84.60 SetGadgetData

Syntax

```
SetGadgetData(#Gadget, Value)
```

Description

Stores the given value with the specified #Gadget. This value can later be read with the GetGadgetData() function. This allows to associate a custom value with any gadget.

This function works with all PureBasic gadgets.

Example:

```
  ; This code uses SetGadgetData to associate an index for the Messages()
  ; array with each button. This makes the event loop simpler as not every
  ; gadget needs to he handled separately.
  ;
  Dim Messages.s(2)
  Messages(0) = "Good morning"
  Messages(1) = "Hello World"
  Messages(2) = "Nothing to say"
  If OpenWindow(0, 0, 0, 190, 100, "SetGadgetData", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
    ButtonGadget(0,  10, 10, 80, 20, "Button"): SetGadgetData(0, 1)
    ButtonGadget(1,  10, 40, 80, 20, "Button"): SetGadgetData(1, 2)
    ButtonGadget(2,  10, 70, 80, 20, "Button"): SetGadgetData(2, 1)
    ButtonGadget(3, 100, 10, 80, 20, "Button"): SetGadgetData(3, 2)
    ButtonGadget(4, 100, 40, 80, 20, "Button") ; will have the value 0 because nothing was set yet
    ButtonGadget(5, 100, 70, 80, 20, "Button")
    Repeat
      Event = WaitWindowEvent()
      If Event = #PB_Event_Gadget
        Value = GetGadgetData(EventGadget())
        MessageRequester("Message", Messages(Value))
      EndIf
    Until Event = #PB_Event_CloseWindow
  EndIf
```

84.61 SetGadgetFont

Syntax

```
SetGadgetFont(#Gadget, FontID)
```

Description

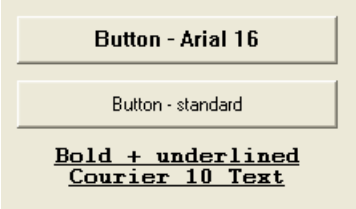
Changes the font of the specified #Gadget. The FontID() function can be used to easily obtain a valid FontID.

If #Gadget is set to #PB_Default, the font used by newly created gadgets is changed to the new one.

If 'FontID' is set to #PB_Default, then the system default font will be used.

Example:

```
If OpenWindow(0, 0, 0, 222, 130, "SetGadgetFont",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If LoadFont(0, "Arial", 16)
    SetGadgetFont(#PB_Default, FontID(0))   ; Set the loaded
 Arial 16 font as new standard
  EndIf
  ButtonGadget(0, 10, 10, 200, 30, "Button - Arial 16")
  SetGadgetFont(#PB_Default, #PB_Default)  ; Set the font
 settings back to original standard font
  ButtonGadget(1, 10, 50, 200, 30, "Button - standard")
  If LoadFont(1,"Courier",10,
 #PB_Font_Bold|#PB_Font_Underline)
    SetGadgetFont(#PB_Default, FontID(1))   ; Set the loaded
 Courier 10 font as new standard
  EndIf
  TextGadget(2, 10, 90, 200, 40, "Bold + underlined Courier
 10 Text", #PB_Text_Center)
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.62 SetGadgetItemAttribute

Syntax

```
SetGadgetItemAttribute(#Gadget, Item, Attribute, Value [,
    Column])
```

Description

Changes the 'Attribute' value of the specified #Gadget item. This function is available for all gadgets which support item attributes:

- ExplorerListGadget() :

```
#PB_Explorer_ColumnWidth : Changes the width of the given
 'Column'. The 'Item' parameter is ignored.
```

- ListIconGadget() :

```
#PB_ListIcon_ColumnWidth : Changes the width of the given
 'Column'. The 'Item' parameter is ignored.
```

84.63 SetGadgetItemColor

Syntax

```
SetGadgetItemColor(#Gadget, Item, ColorType, Color [, Column])
```

Description

Changes the color attribute specified by 'ColorType' of the given #Gadget item. RGB() can be used to get a valid color value. If a gadget item has no color set for a specific ColorType, the item will be drawn using the Color that was set for the whole gadget. To remove the custom color, set the 'Color' parameter to -1.

If the 'Item' parameter is set to -1, the color will be applied to the given 'Column' in all gadget items. If the 'Column' parameter is set to -1, the color will be applied to all columns of the given Item.

The following values are possible for the ColorType parameter:

```
#PB_Gadget_FrontColor      : Item text
#PB_Gadget_BackColor       : Item background
```

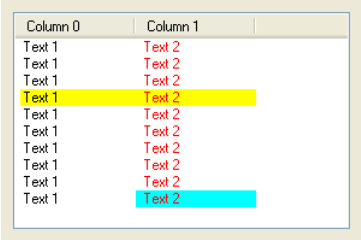
This function is supported by the following gadgets:

- ListIconGadget()
- TreeGadget()

Note: With activated Windows XP style the color settings will probably be ignored or overwritten by the style.

Example:

```
If OpenWindow(0, 0, 0, 300, 300, "SetGadgetItemColor",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ListIconGadget(0, 10, 10, 280, 280, "Column 0", 100)
  AddGadgetColumn(0, 1, "Column 1", 100)
  For i = 1 To 10
    AddGadgetItem(0, -1, "Text 1"+Chr(10)+"Text 2")
  Next

  SetGadgetItemColor(0, -1, #PB_Gadget_FrontColor, $0000FF,
 1)
  SetGadgetItemColor(0,  3, #PB_Gadget_BackColor,  $00FFFF,
 -1)
  SetGadgetItemColor(0,  9, #PB_Gadget_BackColor,  $FFFF00,
 1)
  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.64 SetGadgetItemData

Syntax

```
SetGadgetItemData(#Gadget, Item, Value)
```

Description

Stores the given value with the specified #Gadget item. This value can later be read with the GetGadgetItemData() function. This allows to associate a custom value with the items of a gadget. This value will remain with the item, even if the item changes its index (for example because other items were deleted).

This function works with the following gadgets:
- ComboBoxGadget()
- ListIconGadget()
- ListViewGadget()
- TreeGadget()

Example:

```
; This code uses SetGadgetItemData to store the original position
; of each item to later know it, even if the items index changed.
;
If OpenWindow(0, 0, 0, 280, 250, "SetGadgetItemData", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ButtonGadget(0,  10, 10, 80, 20, "Add")
  ButtonGadget(1, 100, 10, 80, 20, "Remove")
  ButtonGadget(2, 190, 10, 80, 20, "Test")
  ListViewGadget(3, 10, 40, 260, 200)
  For i = 0 To 10
    AddGadgetItem(3, i, "Old Item "+Str(i))
    SetGadgetItemData(3, i, i)
  Next i

  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Gadget
      item = GetGadgetState(3)

      Select EventGadget()
        Case 0 ; Add
          AddGadgetItem(3, item, "New Item")
          If item <> -1
            SetGadgetItemData(3, item, -1)
          Else
            SetGadgetItemData(3, CountGadgetItems(3)-1, -1)
```

```
            EndIf

          Case 1 ; Remove
            If item <> -1
              RemoveGadgetItem(3, item)
            EndIf

          Case 2 ; Test
            If item <> -1
              value = GetGadgetItemData(3, item)
              If value = -1
                MessageRequester("", "Its a new item.")
              Else
                MessageRequester("", "It was item number "+Str(value))
              EndIf
            EndIf

        EndSelect
      EndIf
    Until Event = #PB_Event_CloseWindow
  EndIf
```

84.65 SetGadgetItemState

Syntax

```
SetGadgetItemState(#Gadget, Item, State)
```

Description

Changes the item state of the specified #Gadget. This is a universal function which works for almost all gadgets which handle several items. 'State' can take the following values:
- CalendarGadget() : #PB_Calendar_Bold to make a date appear bold, #PB_Calendar_Normal otherwise.
- ListViewGadget() : 1 if the item should be selected, 0 otherwise.
- ListIconGadget() : Combination of the following values:

```
#PB_ListIcon_Selected: The item should be selected.
#PB_ListIcon_Checked : The item should have its checkbox
 checked (#PB_ListIcon_CheckBoxes flags).
```

- TreeGadget() : Combination of the following values:

```
#PB_Tree_Selected  : The item should be selected.
#PB_Tree_Expanded  : The item should be expanded.
#PB_Tree_Collapsed : The item should be collapsed. If neither
 #PB_Tree_Expanded nor #PB_Tree_Collapsed is set, this state
 will not be changed.
#PB_Tree_Checked   : The items checkbox should be checked.
```

```
SetGadgetItemState(0, 1, #PB_Tree_Expanded |
 #PB_Tree_Selected) ; The 2nd item is selected and expanded.
```

84.66 SetGadgetItemText

Syntax

```
SetGadgetItemText(#Gadget, Item, Text$ [, Column])
```

Description

Changes the item text of the specified #Gadget. This is a universal function which works for almost all gadgets which handle several items:

- ComboBoxGadget()
- EditorGadget()
- ExplorerListGadget() : If Item = -1, the header text of the given column is changed.
- ListIconGadget() : If Item = -1, the header text of the given column is changed.
- ListViewGadget()
- MDIGadget()
- PanelGadget()
- TreeGadget()
- WebGadget() : Change the html code in the gadget with #PB_Web_HtmlCode as 'Item'.

84.67 SetGadgetState

Syntax

```
SetGadgetState(#Gadget, State)
```

Description

Change the current state of the specified #Gadget. This is a universal function which works for almost all gadgets:

- ButtonImageGadget() : change the current state of a #PB_Button_Toggle gadget (1 = toggled, 0 = normal).
- ButtonGadget() : change the current state of a #PB_Button_Toggle gadget (1 = toggled, 0 = normal).
- CalendarGadget() : set the currently selected date.
- CheckBoxGadget() : Change the state of the checkbox. The following values are possible:

```
#PB_CheckBox_Checked   : Set the check mark.
#PB_CheckBox_Unchecked: Remove the check mark.
#PB_CheckBox_Inbetween: Set the "inbetween" state. (Only for
 #PB_CheckBox_ThreeState checkboxes)
```

- ComboBoxGadget() : change the currently selected item.
- DateGadget() : set the currently displayed date/time. If #PB_Date_CheckBox was used, set 'State' to 0 to uncheck the checkbox.
- ImageGadget() : change the current image of the gadget.
- IPAddressGadget() : change the current IP address.
- ListIconGadget() : change the currently selected item. If -1 is specified, all items will be deselected.
- ListViewGadget() : change the currently selected item. . If -1 is specified, it will remove the selection.
- MDIGadget() : Change the currently focused childwindow (by giving the related #Window number), or use one of the following values:

```
#PB_MDI_Cascade          : Cascade the child windows
#PB_MDI_TileVertically   : Tile the childwindows vertically
#PB_MDI_TileHorizontally: Tile the childwindows horizontally
#PB_MDI_Next             : Give focus to the next childwindow
#PB_MDI_Previous         : Give focus to the previous
 childwindow
#PB_MDI_Arrange          : Arrange the iconic (minimized)
 windows
```

- OptionGadget() : 1 to activate it, 0 otherwise.
- PanelGadget() : change the current panel.
- ProgressBarGadget() : change progress bar value.
- ScrollBarGadget() : change the current slider position.
- ShortcutGadget() : Change the current shortcut.
- SpinGadget() : change the current value.
- SplitterGadget() : change the current splitter position, in pixels.
- TrackBarGadget() : change the current cursor position.
- TreeGadget() : change the currently selected item, -1 selects no item.
- WebGadget() : perform some action on the gadget. See WebGadget for further descriptions.

Note: Please take a look at the notes in the chapter Windows Message Handling about the message handling on Windows.

84.68 SetGadgetText

Syntax

```
SetGadgetText(#Gadget, Text$)
```

Description

Change the gadget text content of the specified #Gadget. This function is especially useful for:

- ButtonGadget() : change the text of the ButtonGadget.
- ComboBoxGadget() : Set the displayed text. If the ComboBoxGadget is not editable, the text must be in the dropdown list.
- DateGadget() : change the input mask for the dates in the gadget. See FormatDate() for the format of the Text$ parameter.
- EditorGadget() : change the text content of the editor gadget. If you want to add several lines of text at once, separate them with "Chr(13)+Chr(10)".
- ExplorerComboGadget() : change the current displayed directory.
- ExplorerListGadget() : change the current displayed directory and/or the pattern for files (see ExplorerListGadget() for more details).
- ExplorerTreeGadget() : change the current selected directory and/or the pattern for files (see ExplorerTreeGadget() for more details).
- Frame3DGadget() : change the title of the Frame3DGadget.
- HyperLinkGadget() : change the text of the HyperLinkGadget.
- ListViewGadget() : selects the item that exactly matches the given text.
- StringGadget() : change the content of the StringGadget.
- TextGadget() : change the content of the TextGadget.
- TreeGadget() : change the text of the currently selected item.
- WebGadget() : change the current URL.

84.69 ShortcutGadget

Syntax

```
Result = ShortcutGadget(#Gadget, x, y, Width, Height, Shortcut)
```

Description

Creates a gadget for keyboard shortcut selection in the current GadgetList. The user can select it and hold down a keyboard combination to select a new shortcut. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function when the user changes the shortcut.

'Shortcut' specifies the initial shortcut to display. The possible values are the same as in the AddKeyboardShortcut() function. A value of 0 can be used to indicate that no shortcut is currently set. You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on a ShortcutGadget:

- GetGadgetState() : Get the currently selected keyboard shortcut.
- SetGadgetState() : Change the currently selected keyboard shortcut.

Example:

```
If OpenWindow(0, 0, 0, 240, 70, "ShortcutGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ShortcutGadget(0, 20, 20, 200, 25,
 #PB_Shortcut_Control|#PB_Shortcut_A)
  Repeat
    Event = WaitWindowEvent()
  Until Event = #PB_Event_CloseWindow
EndIf
```

84.70 SpinGadget

Syntax

```
Result = SpinGadget(#Gadget, x, y, Width, Height, Minimum,
   Maximum [, Flags])
```

Description

Creates a spin gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function.

'Flags' is an optional parameter and can be composed of one of the following constants:

```
#PB_Spin_ReadOnly : The StringGadget is not editable, the
 number is only changeable by the arrows
#PB_Spin_Numeric  : The SpinGadget will automatically update
 the text with value of the state, so SetGadgetText is not
 needed.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on a SpinGadget:

GetGadgetState() : Get the current gadget value.
SetGadgetState() : Change the gadget value. For displaying the new value you still must use SetGadgetText() !
GetGadgetText() : Get the text contained in the gadget.
SetGadgetText() : Change the text contained in the gadget.
GetGadgetAttribute() : With one of the following attributes:

```
#PB_Spin_Minimum   : Returns the minimum value.
#PB_Spin_Maximum   : Returns the maximum value.
```

SetGadgetAttribute() : With one of the following attributes:

```
#PB_Spin_Minimum   : Changes the minimum value.
#PB_Spin_Maximum   : Changes the maximum value.
```

The following events are supported through EventType() :

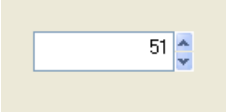
```
#PB_EventType_Change: The text in the edit area has been
 modified by the user.
                    1: The 'Up' button was pressed.
                   -1: The 'Down' button was pressed.
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

```
#PB_Gadget_FrontColor: Textcolor
#PB_Gadget_BackColor : Backgroundcolor
```

Example:

```
1  If OpenWindow(0, 0, 0, 140, 70, "SpinGadget",
    #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2   SpinGadget     (0, 20, 20, 100, 25, 0, 1000)
3   SetGadgetState (0, 5) : SetGadgetText(0, "5")    ; set
    initial value
4   Repeat
5     Event = WaitWindowEvent()
6     If Event = #PB_Event_Gadget
7       If EventGadget() = 0
8         SetGadgetText(0,Str(GetGadgetState(0)))
9       EndIf
10    EndIf
11  Until Event = #PB_Event_CloseWindow
12 EndIf
```



84.71 SplitterGadget

Syntax

```
Result = SplitterGadget(#Gadget, x, y, Width, Height, #Gadget1,
   #Gadget2 [, Flags])
```

Description

Creates a Splitter gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. It contains two gadgets which are separated by an horizontal or vertical bar and which can be resized automatically. #Gadget will be the number returned by EventGadget() function.

'Flags' is an optional parameter and can be composed of one of the following constants:

```
#PB_Splitter_Vertical     : The gadget is split vertically
 (instead of horizontally which is the default).
#PB_Splitter_Separator    : A 3D-looking separator is drawn in
 the splitter.
#PB_Splitter_FirstFixed   : When the splitter gadget is
 resized, the first gadget will keep its size
#PB_Splitter_SecondFixed  : When the splitter gadget is
 resized, the second gadget will keep its size
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on a SplitterGadget:

GetGadgetState() : Get the current splitter position, in pixels.
SetGadgetState() : Change the current splitter position, in pixels.
GetGadgetAttribute() : With one of the following attribute:

```
#PB_Splitter_FirstMinimumSize : Gets the minimum size (in
 pixels) than the first gadget can have.
#PB_Splitter_SecondMinimumSize: Gets the minimum size (in
 pixels) than the second gadget can have.
#PB_Splitter_FirstGadget      : Gets the gadget number of the
 first gadget.
#PB_Splitter_SecondGadget     : Gets the gadget number of the
 second gadget.
```

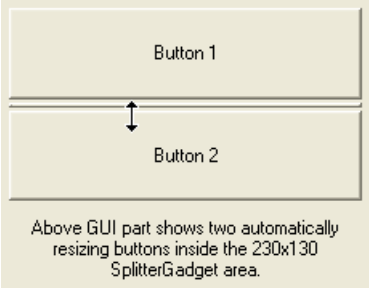
SetGadgetAttribute() : With one of the following attribute:

```
#PB_Splitter_FirstMinimumSize : Sets the minimum size (in
 pixels) than the first gadget can have.
#PB_Splitter_SecondMinimumSize: Sets the minimum size (in
 pixels) than the second gadget can have.
#PB_Splitter_FirstGadget      : Replaces the first gadget
 with a new one.
#PB_Splitter_SecondGadget     : Replaces the second gadget
 with a new one.
```

Note: When replacing a gadget with SetGadgetAttribute() , the old gadget will not be automatically freed. It will instead be put back on the parent window of the Splitter. This allows to switch gadgets between splitters without the need to recreate any of them. If the old gadget should be freed, its number can first be retrieved with GetGadgetAttribute() , and the gadget freed with FreeGadget() after it has been replaced. Note that a gadget cannot be in two splitters at once. So to move a gadget from one splitter to another, it first needs to be replaced in the first splitter so it is on the main window and then it can be put into the second splitter.

Example:

```
If OpenWindow(0, 0, 0, 230, 180, "SplitterGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  #Button1  = 0
  #Button2  = 1
  #Splitter = 2

  ButtonGadget(#Button1, 0, 0, 0, 0, "Button 1") ; No need to
 specify size or coordiantes
  ButtonGadget(#Button2, 0, 0, 0, 0, "Button 2") ; as they
 will be sized automatically
  SplitterGadget(#Splitter, 5, 5, 220, 120, #Button1,
 #Button2, #PB_Splitter_Separator)

  TextGadget(3, 10, 135, 210, 40, "Above GUI part shows two
 automatically resizing buttons inside the 230x130
 SplitterGadget area.",#PB_Text_Center )

  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.72 StringGadget

Syntax

```
Result = StringGadget(#Gadget, x, y, Width, Height, Content$ [,
    Flags])
```

Description

Creates a String gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function. 'Content$' specifies the initial content of this StringGadget. This gadget accepts only one line of text. To get multiline input, use the EditorGadget() function.

'Flags' are always optional and can be composed (using the '||' OR operator) of one of the following constants:

```
#PB_String_Numeric     : Only (positive) integer numbers are
 accepted.
#PB_String_Password    : Password mode, displaying only '*'
 instead of normal characters.
#PB_String_ReadOnly    : Read only mode. No text can be
 entered.
#PB_String_LowerCase   : All characters are converted to lower
 case automatically.
#PB_String_UpperCase   : All characters are converted to upper
 case automatically.
#PB_String_BorderLess : No borders are drawn around the
 gadget.
```

Later the content can be changed with SetGadgetText() and received with GetGadgetText() .

The following events are supported through EventType() :

```
#PB_EventType_Change     : The text has been modified by the
 user.
#PB_EventType_Focus      : The StringGadget got the focus.
#PB_EventType_LostFocus : The StringGadget lost the focus.
```

This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

```
#PB_Gadget_FrontColor: Textcolor
#PB_Gadget_BackColor : Backgroundcolor
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

Example:

```
; Shows possible flags of StringGadget in action...
If OpenWindow(0, 0, 0, 322, 205, "StringGadget Flags",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  StringGadget(0, 8,  10, 306, 20, "Normal StringGadget...")
  StringGadget(1, 8,  35, 306, 20, "1234567",
 #PB_String_Numeric)
  StringGadget(2, 8,  60, 306, 20, "Readonly StringGadget",
 #PB_String_ReadOnly)
  StringGadget(3, 8,  85, 306, 20, "lowercase...",
 #PB_String_LowerCase)
  StringGadget(4, 8, 110, 306, 20, "uppercase...",
 #PB_String_UpperCase)
  StringGadget(5, 8, 140, 306, 20, "Borderless StringGadget",
 #PB_String_BorderLess)
  StringGadget(6, 8, 170, 306, 20, "Password",
 #PB_String_Password)
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

84.73 TextGadget

Syntax

```
Result = TextGadget(#Gadget, x, y, Width, Height, Text$ [,
   Flags])
```

Description

Creates a Text gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. A TextGadget is a basic text area for displaying, not entering, text.

The content can be changed with the function SetGadgetText() and can be obtained with GetGadgetText() . The font of a TextGadget() you easily change with SetGadgetFont() .

'Flags' are always optional and can be composed of one of the following constants:

```
#PB_Text_Center : The text is centered in the gadget.
#PB_Text_Right  : The text is right aligned.
#PB_Text_Border : A sunken border is drawn around the gadget.
```

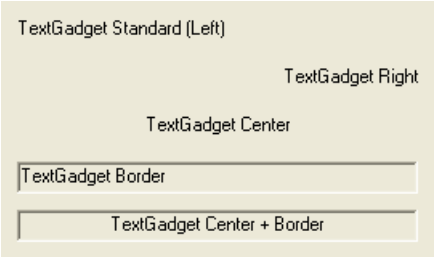
This gadget supports the SetGadgetColor() and GetGadgetColor() functions with the following values as 'ColorType':

```
#PB_Gadget_FrontColor: Textcolor
#PB_Gadget_BackColor : Backgroundcolor
```

Note: This Gadget doesn't receive any user events, and GadgetToolTip() can't be used with it.

Example:

```
If OpenWindow(0, 0, 0, 270, 160, "TextGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  TextGadget(0, 10,  10, 250, 20, "TextGadget Standard
 (Left)")
  TextGadget(1, 10,  70, 250, 20, "TextGadget Center",
 #PB_Text_Center)
  TextGadget(2, 10,  40, 250, 20, "TextGadget Right",
 #PB_Text_Right)
  TextGadget(3, 10, 100, 250, 20, "TextGadget Border",
 #PB_Text_Border)
  TextGadget(4, 10, 130, 250, 20, "TextGadget Center +
 Border", #PB_Text_Center|#PB_Text_Border)
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.74 TrackBarGadget

Syntax

```
Result = TrackBarGadget(#Gadget, x, y, Width, Height, Minimum,
    Maximum [, Flags])
```

Description

Creates a TrackBar gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. It allows you to select a range of values with a slide bar, like ones found in several multimedia players. The Minimum-Maximum range should be between 0 and 10,000. #Gadget will be the number returned by EventGadget() function.

'Flags' are always optional and can be composed of one of the following constants:

```
#PB_TrackBar_Ticks     : Display a 'tick' marker at each step.
#PB_TrackBar_Vertical  : The trackbar is vertical (instead of
 horizontal which is the default).
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on this gadget:

- GetGadgetState() : Returns the current cursor position (value between the Minimum-Maximum range).
- SetGadgetState() : Change the current cursor position.
- GetGadgetAttribute() : With one of the following attributes:

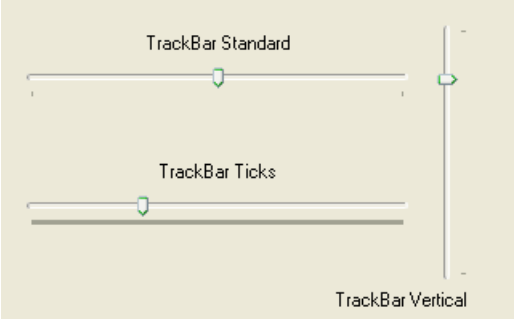
```
#PB_TrackBar_Minimum    : Returns the minimum value.
#PB_TrackBar_Maximum    : Returns the maximum value.
```

- SetGadgetAttribute() : With one of the following attributes:

```
#PB_TrackBar_Minimum    : Changes the minimum value.
#PB_TrackBar_Maximum    : Changes the maximum value.
```

Example:

```
If OpenWindow(0, 0, 0, 320, 200, "TrackBarGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  TextGadget     (3, 10,  20, 250, 20,"TrackBar Standard",
 #PB_Text_Center)
  TrackBarGadget(0, 10,  40, 250, 20, 0, 10000)
  SetGadgetState(0, 5000)
  TextGadget     (4, 10, 100, 250, 20, "TrackBar Ticks",
 #PB_Text_Center)
  TrackBarGadget(1, 10, 120, 250, 20, 0, 30,
 #PB_TrackBar_Ticks)
  SetGadgetState(1, 3000)
  TextGadget     (5,  90, 180, 200, 20, "TrackBar Vertical",
 #PB_Text_Right)
  TrackBarGadget(2, 270, 10, 20, 170, 0, 10000,
 #PB_TrackBar_Vertical)
  SetGadgetState(2, 8000)
  Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```



84.75 TreeGadget

Syntax

```
Result = TreeGadget(#Gadget, x, y, Width, Height [, Flags])
```

Description

Creates a Tree gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function. Once a Tree is created, its list of items is empty.

Each item in the tree has a sublevel value assigned to it, that determines its relation with the item above and below it. Items with the same sublevel belong to the same node, items with a higher sublevel are childitems and so on. This sublevel value can be used to determine the relation between two items by comparing their sublevel values. The 'Flags' parameter of AddGadgetItem() is always required for TreeGadget items and is used to set the sublevel at

which the new item should be added. Note that if the function is called with a sublevel at which the item cannot be added, the item will be added at the sublevel where it is possible to add it.

'Flags' are optional and can be composed (using the '||' OR operator) of one of the following constants:

```
#PB_Tree_AlwaysShowSelection : Even if the gadget isn't
 activated, the selection is still visible.
#PB_Tree_NoLines             : Hide the little lines between
 each nodes.
#PB_Tree_NoButtons           : Hide the '+' node buttons.
#PB_Tree_CheckBoxes          : Add a checkbox before each
 item.
```

You can add a 'mini help' to this gadget by using GadgetToolTip() .

The following functions can be used to act on the tree content:

- AddGadgetItem() : Add an item (with an optional picture in the standard 16x16 icon size).
- RemoveGadgetItem() : Remove an item (and all its child items).
- ClearGadgetItems() : Remove all the items.
- CountGadgetItems() : Return the number of items currently in the #Gadget.
- GetGadgetItemState() : Return the current state of the specified item.
- SetGadgetItemState() : Change the current state of the specified item.
- GetGadgetItemText() : Return the current text of the specified item.
- SetGadgetItemText() : Change the current text of the specified item.
- GetGadgetItemData() : Returns the value that was stored with item.
- SetGadgetItemData() : Stores a value with the item.
- GetGadgetState() : Return the current selected item.
- SetGadgetState() : Change the currently selected item.
- GetGadgetText() : Return the text of the currently selected item.
- SetGadgetText() : Change the text of the currently selected item.
- GetGadgetItemAttribute() : With the following attribute:

```
#PB_Tree_SubLevel: Returns the sublevel value of the given
 item.
```

- GadgetItemID() : Return the OS handle of the specified item (useful for API functions).

The following events are supported through EventType() :

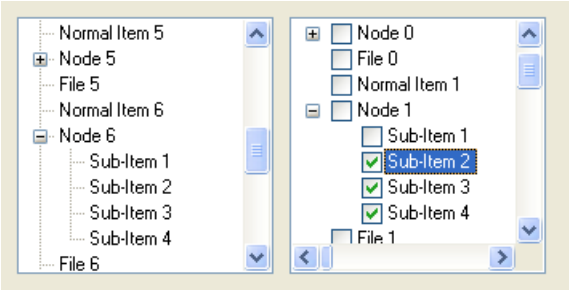
```
#PB_EventType_LeftClick: left click on an item, or a checkbox
 was checked/unchecked
#PB_EventType_LeftDoubleClick
#PB_EventType_RightClick
#PB_EventType_RightDoubleClick
#PB_EventType_Change: the current item changed
#PB_EventType_DragStart: the user tried to start a Drag &
 Drop operation.
```

After a #PB_EventType_DragStart event, the Drag & Drop library can be used to start a Drag & Drop operation.

Example:

```
If OpenWindow(0, 0, 0, 355, 180, "TreeGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
 TreeGadget(0, 10, 10, 160, 160)
                                          ; TreeGadget standard
 TreeGadget(1, 180, 10, 160, 160,
 #PB_Tree_CheckBoxes|#PB_Tree_NoLines)  ; TreeGadget with
 Checkboxes + NoLines
```

```
    For ID = 0 To 1
      For a = 0 To 10
        AddGadgetItem (ID, -1, "Normal Item "+Str(a), 0, 0) ; if you want to add an image, use
        AddGadgetItem (ID, -1, "Node "+Str(a), 0, 0)        ; ImageID(x) as 4th parameter
        AddGadgetItem(ID, -1, "Sub-Item 1", 0, 1)     ; These are on the 1st sublevel
        AddGadgetItem(ID, -1, "Sub-Item 2", 0, 1)
        AddGadgetItem(ID, -1, "Sub-Item 3", 0, 1)
        AddGadgetItem(ID, -1, "Sub-Item 4", 0, 1)
        AddGadgetItem (ID, -1, "File "+Str(a), 0, 0) ; sublevel 0 again
      Next
    Next
    Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```



84.76 UseGadgetList

Syntax

```
Result = UseGadgetList(WindowID)
```

Description

Select the GadgetList window to which gadgets will be added. If there is no GadgetList on this window so far it will be created. (because it was created with the #PB_Window_NoGadgets flag in OpenWindow() or because it is not a PB window)

'WindowID' specifies the new window to add gadgets to. It can be obtained with the WindowID() function. If 'WindowID' is 0, the current GadgetList window will be returned and nothing will be changed.

Returns the WindowID of the previous GadgetList window, or 0 if there was none. This value can be used to go back to the previous GadgetList later.

Example:

This example shows how to use this command to create a new window with gadgets without interrupting the gadget creation on the current window:

```
If OpenWindow(0, 0, 0, 500, 500, "Main Window", #PB_Window_SystemMenu|#PB_Window_ScreenCentered)
 ButtonGadget(0, 10, 10, 150, 25, "Button 1")

 ; Create Window with #PB_Window_NoGadgets to prevent automatic GadgetList creation
 If OpenWindow(1, 0, 0, 300, 200, "Child Window", #PB_Window_TitleBar|#PB_Window_WindowCentered|#PB_Window_NoGadgets, WindowID(0))
    OldGadgetList = UseGadgetList(WindowID(1)) ; Create GadgetList and store old GadgetList

    ButtonGadget(10, 10, 10, 150, 25, "Child Window Button")

    UseGadgetList(OldGadgetList)               ; Return to previous GadgetList
 EndIf

 ButtonGadget(1, 10, 45, 150, 25, "Button 2") ; This will be on the main window again

 Repeat
 Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

84.77 WebGadget

Syntax

```
Result = WebGadget(#Gadget, x, y, Width, Height, URL$ [, Flags])
```

Description

Creates a Web gadget in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'. #Gadget will be the number returned by EventGadget() function.

'Flags' is an optional parameter and can be composed of one of the following constants:

```
#PB_Web_Mozilla : Uses the Mozilla ActiveX instead of the IE one (Windows only).
```

The following functions can be used to act on a WebGadget:

- SetGadgetText() : Change the current URL.
- GetGadgetText() : Get the current URL.
- SetGadgetState() : Perform an action on the gadget. The following constants are valid:

```
#PB_Web_Back    : One step back in the navigation history.
#PB_Web_Forward: One step forward in the navigation history.
#PB_Web_Stop    : Stop loading the current page.
#PB_Web_Refresh: Refresh the current page.
```

Note: The following features do not work with the Mozilla ActiveX on windows (#PB_Web_Mozilla flag)

- SetGadgetItemText() : With #PB_Web_HtmlCode as 'Item' html code can be streamed into the Gadget.
- GetGadgetItemText() : The following constants can be used to get information:

```
#PB_Web_HtmlCode      : Get the html code from the gadget.
#PB_Web_PageTitle     : Get the current title for the
 displayed page.
#PB_Web_StatusMessage: Get the current statusbar message.
#PB_Web_SelectedText : Get the currently selected text inside
 the gadget.
```

- SetGadgetAttribute() : Set the following attributes:

```
#PB_Web_ScrollX        : Set the horizontal scrolling position.
#PB_Web_ScrollY´       : Set the vertical scrolling position.
#PB_Web_BlockPopups    : Block popup windows.
 #PB_EventType_PopupWindow is fired if this setting is
 enabled.
#PB_Web_BlockPopupMenu: Block standard the popup menu.
 #PB_EventType_PopupMenu is fired if this setting is enabled.
#PB_Web_NavigationCallback: Set a callback for monitoring
 (and disabling) navigation.
```

The Navigation callback must have the following format:

```
Procedure NavigationCallback(Gadget, Url$)
  ;
  ; Return #True to allow this navigation or #False to deny it.
  ;
  ProcedureReturn #True
EndProcedure
```

- GetGadgetAttribute() : Get the following attributes:

```
#PB_Web_ScrollX        : Get the horizontal scrolling position.
#PB_Web_ScrollY´       : Get the vertical scrolling position.
#PB_Web_Busy           : Returns nonzero if the gadget is busy
 loading a page.
#PB_Web_Progress       : Returns the current (sometimes
 estimated) progress after a #PB_EventType_DownloadProgress
 event.
#PB_Web_ProgressMax    : Returns the current (sometimes
 estimated) maximum progress after a
 #PB_EventType_DownloadProgress event.
#PB_Web_BlockPopups    : Get the current popupwindow blocking
 setting.
#PB_Web_BlockPopupMenu: Get the current popupmenu blocking
 setting.
#PB_Web_NavigationCallback: Get the current navigation
 callback (if any).
```

Notes: Popup windows and popup menues are not supported on the WebGadget on Linux. However, the #PB_Web_BlockPopups can still be set to enable the #PB_EventType_PopupWindow event. The same also applies for #PB_Web_BlockPopupMenu.

The following types of events can happen for this gadget:

```
#PB_EventType_TitleChange      : The page title changed.
#PB_EventType_StatusChange     : The status message changed.
#PB_EventType_DownloadStart    : A page download started.
#PB_EventType_DownloadProgress: Progress info is available
 with GetGadgetAttribute()
```

.

```
#PB_EventType_DownloadEnd      : A page download ended
 (finished or aborted).
```

```
#PB_EventType_PopupWindow      : A popup window was blocked.
#PB_EventType_PopupMenu        : The popup menu was blocked
 (display a custom menu here).
```

Note: on Microsoft Windows the WebGadget uses the Internet Explorer 4.0+ ActiveX object. This means that IE has to be installed (already present on Win98/Me and Win2000/XP). The ATL.dll, present in the PureBasic\Compilers\ directory is also needed, and must be in the same directory as the executable. As an alternative, the Mozilla ActiveX can be used instead (http://www.iol.ie/* *locka/mozilla/mozilla.htm) if the #PB_Web_Mozilla flag is specified (the Mozilla ActiveX needs to be installed separately, it isn't bundled with FireFox or Mozilla).

Example:

```
If OpenWindow(0, 0, 0, 600, 300, "WebGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  WebGadget(0, 10, 10, 580, 280, "http://www.purebasic.com")
  ; Note: if you want to use a local file, change last
 parameter to "file://" + path + filename
  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

Example 2: (with navigation callback)

```
; This example does display the PureBasic.com website. Inside
 the callback procedure
; the navigation to the 'News' site will be avoided (#False
 returned), but allowed
; for all other sites (#True returned).

Procedure NavigationCallback(Gadget, Url$)
  If Url$= "http://www.purebasic.com/news.php"
    MessageRequester("", "No news today!")
    ProcedureReturn #False
  Else
    ProcedureReturn #True
  EndIf
EndProcedure

If OpenWindow(0, 0, 0, 600, 300, "WebGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  WebGadget(0, 10, 10, 580, 280, "http://www.purebasic.com")
```

```
    SetGadgetAttribute(0, #PB_Web_NavigationCallback,
  @NavigationCallback())
    Repeat
    Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```

84.78 WebGadgetPath

Syntax

```
Result = WebGadgetPath(LibraryFile$ [, MozillaPath$])
```

Description

Specifies the path to the library for the Gtk Mozilla Embedding Widget which is used by the WebGadget() on Linux. This command does not exist on other platforms.

LibraryFile$ specifies the full path (including filename) to the mozilla embedding library. If the argument is empty, a library called "libgtkembedmoz.so" will be searched in the following locations:
- All directories listed in the LD_LIBRARY_PATH environment variable.
- All libraries listed in /etc/ld.so.cache.
- /lib, followed by /usr/lib

MozillaPath$ specifies the path where the mozilla components are located (usually the path where the mozilla or firefox browser is installed). This argument is not needed if the libgtkembedmoz.so is located in the same directory as the mozilla components. Note that if this path is wrong, the WebGadget() will actually be created correctly, but it cannot load any pages.

If the libgtkembedmoz.so is located in the same directory as the mozilla components, and is found in one of the above mentioned search locations, this command is not needed at all.

Returns 1 if the library could be loaded, 0 otherwise. Note that it cannot be verified if the MozillaPath$ was correct or not.

Supported OS

Linux

Chapter 85

Gadget3D

The Gadget3D library allows to create complex Graphical User Interface (GUI) directly over the screen area, using the 3D engine. This is mainly intended for game or application which needs user inputs while running in fullscreen mode. This library is is based on the regular PureBasic gadget library , and offer similar syntax and behaviour. The GUI engine used is CEGUI, which offer some nice options like skinning, good speed and a lot of build-in gadgets. More information about CEGUI can be found here: http://www.cegui.org.uk.

It uses the 3D engine, so InitEngine3D() has to be called succesfully before using these functions.

Before using gadgets there will be normally opened a window first.

85.1 AddGadgetItem3D

Syntax

```
Result = AddGadgetItem3D(#Gadget3D, Position, Text$ [,
    TextureID [, Flags]])
```

Description

Add an item to the specified #Gadget3D. An optional TextureID() can be specified for any gadgets which support it.

The following gadgets are supported:

- ComboBoxGadget3D()
- EditorGadget3D()
- ListViewGadget3D()
- PanelGadget3D()

The position specifies the item index where the new item should be inserted. To add the item at the start, use an index of 0. To add this item to the end of the current item list, use a value of -1. Remember that when you add an item that all current items which come after the new one will have their positions increased by 1.

85.2 ButtonGadget3D

Syntax

```
Result = ButtonGadget3D(#Gadget3D, x, y, Width, Height, Text$
    [, Flags])
```

Description

Create a button gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. #Gadget3D will be the number returned by EventGadget3D() function.

'Flags' are always optional and can be composed of one or several (using the bitwise OR operator '||') of the following constants:

```
#PB_Button3D_Toggle     : Creates a toggle button: one click
 pushes it, another will release it.
```

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

The following functions can be used to control the gadget:

- SetGadgetText3D() : Changes the text of the ButtonGadget.
- GetGadgetText3D() : Returns the text of the ButtonGadget.
- SetGadgetState3D() : Used with #PB_Button3D_Toggle buttons to set the actual state (1 = toggled, 0 = normal).
- GetGadgetState3D() : Used with #PB_Button3D_Toggle buttons to get the actual state of the button (1 = toggled, 0 = normal).

85.3 CheckBoxGadget3D

Syntax

```
Result = CheckBoxGadget3D(#Gadget3D, x, y, Width, Height, Text$)
```

Description

Create a checkbox gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. #Gadget3D will be the number returned by EventGadget3D() function. The 'Text$' is a description of the checkbox which will be displayed to the right of it.

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

- GetGadgetState3D() can be used to get the current gadget state (checked or unchecked).
- SetGadgetState3D() can be used to change the gadget state (checked or unchecked).

85.4 ClearGadgetItems3D

Syntax

```
ClearGadgetItems3D(#Gadget3D)
```

Description

Clears all the items from the specified #Gadget3D. The Gadget must be one of the following types:

- ComboBoxGadget3D()
- EditorGadget3D()
- ListViewGadget3D()
- PanelGadget3D()

85.5 CloseGadgetList3D

Syntax

```
CloseGadgetList3D()
```

Description

Terminate the current gadget list creation and go back to the previous GadgetList.

This is especially useful for the following gadgets:

- ContainerGadget3D()
- PanelGadget3D()
- ScrollAreaGadget3D()

85.6 ComboBoxGadget3D

Syntax

```
Result = ComboBoxGadget3D(#Gadget3D, x, y, Width, Height [,
    Flags])
```

Description

Create a ComboBox gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. #Gadget3D will be the number returned by EventGadget3D() function. Once a ComboBox is created, its list of items is empty. The 'Height' parameter has no effect on the real height of the gadget, which is determined automatically by the font.

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

The following functions can be used to act on the list contents:

- AddGadgetItem3D() : Add an item.
- GetGadgetItemText3D() : Returns the gadget item text.
- CountGadgetItems3D() : Count the items in the current combobox.
- ClearGadgetItems3D() : Remove all the items.
- RemoveGadgetItem3D() : Remove an item.
- SetGadgetItemText3D() : Changes the gadget item text.

- GetGadgetState3D() : Get the index (starting from 0) of the current element.
- GetGadgetText3D() : Get the (text) content of the current element.
- SetGadgetState3D() : Change the selected element.
- SetGadgetText3D() : Set the displayed text. If the ComboBoxGadget is not editable, the text must be in the dropdown list.

'Flags' are always optional and can be composed (using the '||' operator) of one of the following constants:

```
#PB_ComboBox3D_Editable  : Makes the ComboBox editable
```

85.7 ContainerGadget3D

Syntax

```
Result = ContainerGadget3D(#Gadget3D, x, y, Width, Height)
```

Description

Creates a container gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'.

It's a simple panel gadget which can contain other gadgets. Once the gadget is created, all future created gadgets will be created inside the container. When all the needed gadgets have been created, CloseGadgetList3D() must be called to return to the previous GadgetList. OpenGadgetList3D() can be used later to add others gadgets on the fly in the container area.

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

85.8 CountGadgetItems3D

Syntax

```
Result = CountGadgetItems3D(#Gadget3D)
```

Description

Returns the number of items in the specified #Gadget3D. If there are any items in the gadget then the count starts from 1. If the #Gadget3D doesn't contain any items, 0 is returned. This is a universal function which works for all gadgets which handle several items:

- ComboBoxGadget3D()
- EditorGadget3D()
- ListViewGadget3D()
- PanelGadget3D()

85.9 DisableGadget3D

Syntax

```
DisableGadget3D(#Gadget3D, State)
```

Description

Disable or enable the gadget. If State = 1, the gadget will be disabled, if State = 0 it will be enabled.

85.10 EditorGadget3D

Syntax

```
Result = EditorGadget3D(#Gadget3D, x, y, Width, Height [,
    Flags])
```

Description

Creates an Editor gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. #Gadget3D will be the number returned by the EventGadget3D() function.

'Flags' can be the following value:

```
  #PB_Editor3D_ReadOnly: The user cannot edit the text in the
    gadget.
```

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

The following functions can be used to act on the editor content:

- AddGadgetItem3D() : Add a text line.
- CountGadgetItems3D() : Return the number of lines in the editor gadget.
- GetGadgetItemText3D() : Get the specified line text.
- GetGadgetText3D() : Get the text content of the editor gadget.
- RemoveGadgetItem3D() : Remove a line in the editor.
- ClearGadgetItems3D() : Clear the text content.
- SetGadgetItemText3D() : Set the specified line text.
- SetGadgetText3D() : Change the text content of the editor gadget.
- SetGadgetAttribute3D() : With the following attribute:

```
#PB_Editor3D_ReadOnly: Set the readonly state. (0 means
 editable, nonzero means readonly)
```

- GetGadgetAttribute3D() : With the following attribute:

```
#PB_Editor3D_ReadOnly: Get the readonly state. (0 means
 editable, nonzero means readonly)
```

85.11 Frame3DGadget3D

Syntax

```
Result = Frame3DGadget3D(#Gadget3D, x, y, Width, Height, Text$)
```

Description

Creates a Frame3D gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. This kind of gadget is decorative only. The Text$ parameter is only valid if no borders are specfied, else it will be ignored.

Note: As this Gadget is decorative only, GadgetToolTip3D() cannot be used. This Gadget also receives no events.

85.12 FreeGadget3D

Syntax

```
FreeGadget3D(#Gadget3D)
```

Description

Free and remove the #Gadget3D from the display (and free its gadgetlist if the gadget was a container).

Note: all remaining gadgets are automatically freed when the program ends.

85.13 GadgetID3D

Syntax

```
GadgetID = GadgetID3D(#Gadget3D)
```

Description

Returns the unique system identifier of the #Gadget3D.

85.14 GadgetToolTip3D

Syntax

```
GadgetToolTip3D(#Gadget3D, Text$)
```

Description

Associate the specified Text$ with the #Gadget3D. A ToolTip text is text which is displayed when the mouse cursor is over the gadget for a small amount of time (typically a yellow floating box).

The following gadgets are supported:

- ButtonGadget3D()
- CheckBoxGadget3D()
- ComboBoxGadget3D()
- ContainerGadget3D()
- EditorGadget3D()
- ImageGadget3D()
- ListViewGadget3D()
- OptionGadget3D()
- PanelGadget3D()
- ProgressBarGadget3D()
- ScrollBarGadget3D()
- SpinGadget3D()
- StringGadget3D()

85.15 GadgetX3D

Syntax

```
Result = GadgetX3D(#Gadget3D)
```

Description

Returns the X position of the specified #Gadget3D. The X position is measured in pixels, starting from the inside of the left edge of the gadget list that the gadget is in.

85.16 GadgetY3D

Syntax

```
Result = GadgetY3D(#Gadget3D)
```

Description

Returns the Y position of the specified #Gadget3D. The Y position is measured in pixels, starting from the inside of the top edge of the gadget list that the gadget is in.

85.17 GadgetHeight3D

Syntax

```
Result = GadgetHeight3D(#Gadget3D)
```

Description

Returns the height of the specified #Gadget3D, in pixels.

85.18 GadgetType3D

Syntax

```
Result = GadgetType3D(#Gadget3D)
```

Description

Returns the type of gadget that is represented by the specified #Gadget3D number. It can be useful to write generic functions that work with more than one type of gadget.

It returns one of the following values:

```
#PB_GadgetType3D_Button         : ButtonGadget3D()

#PB_GadgetType3D_CheckBox       : CheckBoxGadget3D()

#PB_GadgetType3D_ComboBox       : ComboBoxGadget3D()

#PB_GadgetType3D_Container      : ContainerGadget3D()

#PB_GadgetType3D_Editor         : EditorGadget3D()

#PB_GadgetType3D_Frame3D        : Frame3DGadget3D()

#PB_GadgetType3D_Image          : ImageGadget3D()

#PB_GadgetType3D_ListView       : ListViewGadget3D()

#PB_GadgetType3D_Option         : OptionGadget3D()

#PB_GadgetType3D_Panel          : PanelGadget3D()

#PB_GadgetType3D_ProgressBar    : ProgressBarGadget3D()

#PB_GadgetType3D_ScrollArea     : ScrollAreaGadget3D()

#PB_GadgetType3D_ScrollBar      : ScrollBarGadget3D()

#PB_GadgetType3D_Spin           : SpinGadget3D()

#PB_GadgetType3D_String         : StringGadget3D()

#PB_GadgetType3D_Text           : TextGadget3D()

#PB_GadgetType3D_Unknown        : The type is unknown. Most
 likely it is not a PB gadget at all.
```

85.19 GadgetWidth3D

Syntax

```
Result = GadgetWidth3D(#Gadget3D)
```

Description

Returns the width of the specified #Gadget3D, in pixels.

85.20 GetActiveGadget3D

Syntax

```
Result = GetActiveGadget3D()
```

Description

Returns the #Gadget3D number of the Gadget that currently has the keyboard focus. If no Gadget has the focus, -1 is returned.

85.21 GetGadgetAttribute3D

Syntax

```
Value = GetGadgetAttribute3D(#Gadget3D, Attribute)
```

Description

Gets the 'Attribute' value of the specified #Gadget3D. This function is available for all gadgets which support attributes. See the individual gadget for the supported attributes:

- EditorGadget3D()
- PanelGadget3D()
- ProgressBarGadget3D()
- ScrollAreaGadget3D()
- ScrollBarGadget3D()
- SpinGadget3D()

85.22 GetGadgetData3D

Syntax

```
Result = GetGadgetData3D(#Gadget3D)
```

Description

Returns the 'Data' value that has been stored for this gadget with the SetGadgetData3D() function. This allows to associate a custom value with any gadget. If there was never a data value set for this gadget, the returnvalue will be 0.

This function works with all PureBasic gadgets. See SetGadgetData3D() for an example.

85.23 GetGadgetItemData3D

Syntax

```
Result = GetGadgetItemData3D(#Gadget3D, Item)
```

Description

Returns the value that was previously stored with this #Gadget3D item with the SetGadgetItemData3D() function. This allows to associate a custom value with the items of a gadget. If no value has been stored with the item yet, the returnvalue will be 0.

This function works with the following gadgets:

- ComboBoxGadget3D()
- ListViewGadget3D()

See SetGadgetItemData3D() for an example.

85.24 GetGadgetState3D

Syntax

```
Result = GetGadgetState3D(#Gadget3D)
```

Description

Returns the current state of the gadget. This is a universal function which works for almost all gadgets:

- ButtonGadget3D() : returns 1 if a #PB_Button3D_Toggle button is toggled, else 0.
- CheckBoxGadget3D() : returns 1 if checked, 0 otherwise.
- ComboBoxGadget3D() : returns the currently selected item index, -1 if none is selected.
- ImageGadget3D() : returns the TextureID of the currently displayed image.
- ListViewGadget3D() : returns the currently selected item index, -1 if none is selected.
- OptionGadget3D() : returns 1 if activated, 0 otherwise.
- PanelGadget3D() : returns the current panel index, -1 if no panel.
- ProgressBarGadget3D() : returns the current value of the ProgressBar.
- ScrollBarGadget3D() : returns the current slider position.
- SpinGadget3D() : returns the current value of the SpinGadget.

85.25 GetGadgetItemText3D

Syntax

```
Result\$ = GetGadgetItemText3D(#Gadget3D, Item [, Column])
```

Description

Returns the item text of the specified #Gadget3D. This is a universal function which works for almost all gadgets which handle several items:

- ComboBoxGadget3D() - 'Item' is the index of the item in the ComboBox list. 'Column' will be ignored.
- EditorGadget3D() - 'Item' is the text line in the EditorGadget. 'Column' will be ignored.
- ListViewGadget3D() - 'Item' is the index of the entry from which you want to receive the content. 'Column' will be ignored.
- PanelGadget3D() - 'Item' is the panel from which you want to receive the header text.

Note: All 'Item' index start from 0.

85.26 GetGadgetItemState3D

Syntax

```
Result = GetGadgetItemState3D(#Gadget3D, Item)
```

Description

Returns the item state of the specified #Gadget3D. This is a universal function that works for almost all gadgets which handle several items:

- ListViewGadget3D() : returns 1 if the item is selected, 0 otherwise.

85.27 GetGadgetText3D

Syntax

```
String\$ = GetGadgetText3D(#Gadget3D)
```

Description

Returns the gadget text content of the specified #Gadget3D. This function is especially useful for:

- ButtonGadget3D() : return the text of the ButtonGadget.
- ComboBoxGadget3D() - return the content of the current item.
- EditorGadget3D() - return the text content of the editor gadget. Please note, that several lines of text are normaly separated with "Chr(13)+Chr(10)".
- ListViewGadget3D() - return the content of the current item.
- StringGadget3D() - return contents of the StringGadget.
- TextGadget3D() - return contents of the TextGadget.

85.28 HideGadget3D

Syntax

```
HideGadget3D(#Gadget3D, State)
```

Description

Hide or show a gadget. If State = 1, the gadget will be hidden, if State = 0 it will be shown.

85.29 ImageGadget3D

Syntax

```
Result = ImageGadget3D(#Gadget3D, x, y, Width, Height,
    TextureID [, Flags])
```

Description

Creates an Image gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. The TextureID represents an image and can be obtained with TextureID() from the Texture library. If the TextureID is 0, then no image will be displayed. The gadget dimensions autofit to the image size.

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

- SetGadgetState3D() : Change the current Image of the gadget. A valid TextureID can be easily obtained with the TextureID() function. If the TextureID is 0, then the image is removed from the gadget.

'Flags' are always optional and can be composed of one of the following constants:

```
  #PB_Image3D_Border : display a sunken border around the image.
```

85.30 IsGadget3D

Syntax

```
Result = IsGadget3D(#Gadget3D)
```

Description

Tests if the given #Gadget3D number is a valid and correctly initialized gadget. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a gadget is ready to use.

85.31 ListViewGadget3D

Syntax

```
Result = ListViewGadget3D(#Gadget3D, x, y, Width, Height)
```

Description

Creates a ListView gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. #Gadget3D will be the number returned by the EventGadget3D() function. Once a ListView is created, its list of items is empty.

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

The following functions can be used to act on the list content:

- AddGadgetItem3D() : Add an item.
- RemoveGadgetItem3D() : Remove an item.
- ClearGadgetItems3D() : Remove all the items
- CountGadgetItems3D() : Returns the number of items currently in the #Gadget3D.
- GetGadgetItemData3D() : Get the value that was stored with the gadget item.
- GetGadgetItemState3D() : Returns nonzero if the item is selected, zero otherwise.
- GetGadgetItemText3D() : Get the content of the given item.
- GetGadgetState3D() : Get the index of the selected item or -1 if there is no selected item.
- GetGadgetText3D() : Get the content of the selected item.
- SetGadgetItemData3D() : store a value with the given item.
- SetGadgetItemState3D() : Selects or deselects the given item.
- SetGadgetItemText3D() : Set the text of the given item.
- SetGadgetState3D() : Change the selected item. If -1 is specified, the selection will be removed.
- SetGadgetText3D() : Selects the item with the given text (the text must match exactly).

85.32 OpenGadgetList3D

Syntax

```
OpenGadgetList3D(#Gadget3D [, Item])
```

Description

Use the specified #Gadget3D as a GadgetList, to dynamically add new gadgets to it. Once the all the needed changes are done, CloseGadgetList3D() should be called. The following gadgets are supported by OpenGadgetList3D():

- ContainerGadget3D()
- PanelGadget3D() : The optional 'Item' parameter need to be specified
- ScrollAreaGadget3D()

85.33 OptionGadget3D

Syntax

```
Result = OptionGadget3D(#Gadget3D, x, y, Width, Height, Text$)
```

Description

Creates an OptionGadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. The first time this function is called, a group is created and all following calls of OptionGadget3D() will add a gadget to this group. To finish the group, just create a gadget of another type. These kind of gadgets are very useful as only one gadget from the group can be selected at any time.

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

85.34 PanelGadget3D

Syntax

```
Result = PanelGadget3D(#Gadget3D, x, y, Width, Height)
```

Description

Creates a Panel gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. Once a Panel is created, its list of panels is empty.

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

The following functions can be used to act on the list content:

- AddGadgetItem3D() : Add a panel.
- RemoveGadgetItem3D() : Remove a panel.
- CountGadgetItems3D() : Count the number of panels.
- ClearGadgetItems3D() : Remove all panels.
- GetGadgetItemText3D() : Retrieve the text of the specified item.
- SetGadgetItemText3D() : Change the text of the specified item.
- SetGadgetState3D() : Change the active panel.
- GetGadgetState3D() : Get the index of the current panel.
- GetGadgetAttribute3D() : With one of the following attributes: (there must be at least 1 item for this to work)

```
#PB_Panel3D_ItemWidth : Returns the width of the inner area
 where the gadgets are displayed.
#PB_Panel3D_ItemHeight: Returns the height of the inner area
 where the gadgets are displayed.
#PB_Panel3D_TabHeight : Returns height of the panel tabs on
 top of the gadget.
```

You must call AddGadgetItem3D() to add a panel before you can create other gadgets inside the panel gadget. The next gadgets will then be created automatically in the new panel. This is very convenient. When the PanelGadget item has been filled with all the needed gadgets, CloseGadgetList3D() must be called to return to the previous GadgetList. This means that a PanelGadget can be created inside another PanelGadget...

85.35 ProgressBarGadget3D

Syntax

```
Result = ProgressBarGadget3D(#Gadget3D, x, y, Width, Height,
    Minimum, Maximum)
```

Description

Creates a ProgressBar gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. The possible values that the progress bar can take are fixed between the Minimum and Maximum one.

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

The following functions can be used to act on the gadget:

- SetGadgetState3D() : Change progress bar value.
- GetGadgetState3D() : Get the current progress bar value.
- SetGadgetAttribute3D() : With the following attributes:

```
#PB_ProgressBar3D_Minimum    : Changes the minimum value.
#PB_ProgressBar3D_Maximum    : Changes the maximum value.
```

- GetGadgetAttribute3D() : With the following attributes:

```
#PB_ProgressBar3D_Minimum    : Returns the minimum value.
#PB_ProgressBar3D_Maximum    : Returns the maximum value.
```

85.36 RemoveGadgetItem3D

Syntax

```
RemoveGadgetItem3D(#Gadget3D, Position)
```

Description

Removes an item of the specified #Gadget3D at the given Position. Position index starts at 0. The gadget type can be one of the following:

- ComboBoxGadget3D()
- EditorGadget3D()
- PanelGadget3D()
- ListViewGadget3D()

85.37 ResizeGadget3D

Syntax

```
ResizeGadget3D(#Gadget3D, x, y, Width, Height)
```

Description

Resize the specified #Gadget3D to the given position and dimensions. To ease the building of realtime resizeable Graphical User Interfaces (GUIs), #PB_Ignore can be passed as any parameter (x, y, Width or Height) and this parameter will not be changed. This is very handy because often the gadget should only change in width or in height.

```
[...]

ResizeGadget3D(0, #PB_Ignore, #PB_Ignore, 300, #PB_Ignore) ;
 Only change the gadget width.
```

85.38 ScrollBarGadget3D

Syntax

```
Result = ScrollBarGadget3D(#Gadget3D, x, y, Width, Height,
    Minimum, Maximum, PageLength [, Flags])
```

Description

Creates a scrollbar gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. It's widely used when displaying only a part of an object. The Minimum-Maximum range should be between 0 and 10,000. The 'PageLength' is the number which defines a page. For example you can have a picture which is 100 pixels width and you only see 25 pixels. What you see is a called a 'page', in this example, the page length will be 25, the Mininum will be 0 and the Maximum will be 100. #Gadget3D will be the number returned by EventGadget3D() function.

'Flags' are optionals and can be composed of one of the following constants:

```
  #PB_ScrollBar3D_Vertical : The scrollbar is vertical (instead
   of horizontal, which is the default).
```

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

The following functions can be used to act on this gadget:

- GetGadgetState3D() : Returns the current slider position (value between the Minimum-Maximum range).
- SetGadgetState3D() : Changes the current slider position.
- GetGadgetAttribute3D() : With one of the following attributes:

```
  #PB_ScrollBar3D_Minimum   : Returns the minimum scroll
   position.
  #PB_ScrollBar3D_Maximum   : Returns the maximum scroll
   position.
  #PB_ScrollBar3D_PageLength: Returns the PageLength value.
```

- SetGadgetAttribute3D() : With one of the following attributes:

```
  #PB_ScrollBar3D_Minimum   : Changes the minimum scroll
   position.
  #PB_ScrollBar3D_Maximum   : Changes the maximum scroll
   position.
  #PB_ScrollBar3D_PageLength: Changes the PageLength value.
```

85.39 ScrollAreaGadget3D

Syntax

```
Result = ScrollAreaGadget3D(#Gadget3D, x, y, Width, Height,
    ScrollAreaWidth, ScrollAreaHeight, ScrollStep)
```

Description

Creates a ScrollArea gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. It's very useful when a gadget is too big to fit the window dimension. In that case, it can be put into a scrollarea. All the scrolling is handled automatically by the gadget. This is a container gadget, intended to have one or several gadget in its scroll area. Once the gadget is created, all future created gadgets will be created inside the scoll area. When all the needed gadgets have been created, CloseGadgetList3D() must be called to return to the previous GadgetList. OpenGadgetList3D() can be used later to add others gadgets on the fly in the scroll area.

'ScrollAreaWidth' and 'ScrollAreaHeight' are the initial dimensions of the scrollable area contained inside the gadget and should be greater than the gadget's 'Width' and 'Height' dimensions. 'ScrollStep' is the number of pixels scrolled when pressing a slider arrow. #Gadget3D will be the number returned by EventGadget3D() function. When the user moves the scrollbars in the gadget, an event will be received for the gadget.

The following functions can be used to act on a ScrollAreaGadget:

GetGadgetAttribute3D() : With one of the following attribute:

```
#PB_ScrollArea3D_InnerWidth  : Returns the width (in pixels)
 of the contained scrollable area.
#PB_ScrollArea3D_InnerHeight : Returns the height (in pixels)
 of the contained scrollable area.
#PB_ScrollArea3D_X           : Returns the current horizontal
 scrolling position (in pixels).
#PB_ScrollArea3D_Y           : Returns the current vertical
 scrolling position (in pixels).
```

SetGadgetAttribute3D() : With one of the following attribute:

```
#PB_ScrollArea3D_InnerWidth  : Changes the width (in pixels)
 of the contained scrollable area.
#PB_ScrollArea3D_InnerHeight : Changes the height (in pixels)
 of the contained scrollable area.
#PB_ScrollArea3D_X           : Changes the current horizontal
 scrolling position (in pixels).
#PB_ScrollArea3D_Y           : Changes the current vertical
 scrolling position (in pixels).
```

85.40 SetActiveGadget3D

Syntax

```
SetActiveGadget3D(#Gadget3D)
```

Description

Activates (sets the focus on) the gadget specified by the given #Gadget3D number. This is mainly used with ComboBoxGadget3D() and StringGadget3D() . Activating a gadget allows it to become the current object to receive messages and handle keystrokes.

85.41 SetGadgetAttribute3D

Syntax

```
SetGadgetAttribute3D(#Gadget3D, Attribute, Value)
```

Description

Changes the 'Attribute' value of the specified #Gadget3D. This function is available for all gadgets which support attributes:

- EditorGadget3D()
- ProgressBarGadget3D()
- ScrollAreaGadget3D()
- ScrollBarGadget3D()
- SpinGadget3D()

85.42 SetGadgetData3D

Syntax

```
SetGadgetData3D(#Gadget3D, Value)
```

Description

Stores the given value with the specified #Gadget3D. This value can later be read with the GetGadgetData3D() function. This allows to associate a custom value with any gadget.

This function works with all PureBasic gadgets.

85.43 SetGadgetItemData3D

Syntax

```
SetGadgetItemData3D(#Gadget3D, Item, Value)
```

Description

Stores the given value with the specified #Gadget3D item. This value can later be read with the GetGadgetItemData3D() function. This allows to associate a custom value with the items of a gadget. This value will remain with the item, even if the item changes its index (for example because other items were deleted).

This function works with the following gadgets:
- ComboBoxGadget3D()
- ListViewGadget3D()

85.44 SetGadgetItemState3D

Syntax

```
SetGadgetItemState3D(#Gadget3D, Item, State)
```

Description

Changes the item state of the specified #Gadget3D. This is a universal function which works for almost all gadgets which handle several items. 'State' can take the following values:
- ListViewGadget3D() : 1 if the item should be selected, 0 otherwise.

85.45 SetGadgetItemText3D

Syntax

```
SetGadgetItemText3D(#Gadget3D, Item, Text$ [, Column])
```

Description

Changes the item text of the specified #Gadget3D. This is a universal function which works for almost all gadgets which handle several items:

- ComboBoxGadget3D()
- EditorGadget3D()
- ListViewGadget3D()
- PanelGadget3D()

85.46 SetGadgetState3D

Syntax

```
SetGadgetState3D(#Gadget3D, State)
```

Description

Change the current state of the specified #Gadget3D. This is a universal function which works for almost all gadgets:

- ButtonGadget3D() : change the current state of a #PB_Button3D_Toggle gadget (1 = toggled, 0 = normal).
- CheckBoxGadget3D() : 1 to check it, 0 otherwise.
- ComboBoxGadget3D() : change the currently selected item.
- ImageGadget3D() : change the current image of the gadget.
- ListViewGadget3D() : change the currently selected item. . If -1 is specified, it will remove the selection.
- OptionGadget3D() : 1 to activate it, 0 otherwise.
- PanelGadget3D() : change the current panel.
- ProgressBarGadget3D() : change progress bar value.
- ScrollBarGadget3D() : change the current slider position.
- SpinGadget3D() : change the current value.

85.47 SetGadgetText3D

Syntax

```
SetGadgetText3D(#Gadget3D, Text$)
```

Description

Change the gadget text content of the specified #Gadget3D. This function is especially useful for:

- ButtonGadget3D() : change the text of the ButtonGadget.
- ComboBoxGadget3D() : Set the displayed text. If the ComboBoxGadget is not editable, the text must be in the dropdown list.
- EditorGadget3D() : change the text content of the editor gadget. If you want to add several lines of text at once, separate them with "Chr(13)+Chr(10)".
- Frame3DGadget3D() : change the title of the Frame3DGadget.
- ListViewGadget3D() : selects the item that exactly matches the given text.
- StringGadget3D() : change the content of the StringGadget.
- TextGadget3D() : change the content of the TextGadget.

85.48 SpinGadget3D

Syntax

```
Result = SpinGadget3D(#Gadget3D, x, y, Width, Height, Minimum,
    Maximum [, Flags])
```

Description

Creates a spin gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. #Gadget3D will be the number returned by EventGadget3D() function.

'Flags' is an optional parameter and can be composed of one of the following constants:

```
#PB_Spin3D_ReadOnly : The StringGadget is not editable, the
 number is only changeable by the arrows
#PB_Spin3D_Numeric  : The SpinGadget will automatically
 update the text with value of the state, so SetGadgetText is
 not needed.
```

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

The following functions can be used to act on a SpinGadget:

GetGadgetState3D() : Get the current gadget value.
SetGadgetState3D() : Change the gadget value. For displaying the new value you still must use SetGadgetText3D() !
GetGadgetText3D() : Get the text contained in the gadget.
SetGadgetText3D() : Change the text contained in the gadget.
GetGadgetAttribute3D() : With one of the following attributes:

```
#PB_Spin3D_Minimum   : Returns the minimum value.
#PB_Spin3D_Maximum   : Returns the maximum value.
```

SetGadgetAttribute3D() : With one of the following attributes:

```
#PB_Spin3D_Minimum   : Changes the minimum value.
#PB_Spin3D_Maximum   : Changes the maximum value.
```

85.49 StringGadget3D

Syntax

```
Result = StringGadget3D(#Gadget3D, x, y, Width, Height,
   Content$ [, Flags])
```

Description

Creates a String gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. #Gadget3D will be the number returned by EventGadget3D() function. 'Content$' specifies the initial content of this StringGadget. This gadget accepts only one line of text. To get multiline input, use the EditorGadget3D() function.

'Flags' are always optional and can be composed (using the '||' OR operator) of one of the following constants:

```
#PB_String3D_Numeric    : Only (positive) integer numbers are
 accepted.
#PB_String3D_Password   : Password mode, displaying only '*'
 instead of normal characters.
#PB_String3D_ReadOnly   : Read only mode. No text can be
 entered.
```

Later the content can be changed with SetGadgetText3D() and received with GetGadgetText3D() .

The following events are supported through EventType3D() :

```
#PB_EventType3D_Change    : The text has been modified by the
 user.
#PB_EventType3D_Focus     : The StringGadget got the focus.
#PB_EventType3D_LostFocus : The StringGadget lost the focus.
```

To add a 'mini help' to this gadget, use GadgetToolTip3D() .

85.50 TextGadget3D

Syntax

```
Result = TextGadget3D(#Gadget3D, x, y, Width, Height, Text$)
```

Description

Creates a Text gadget in the current GadgetList. If #PB_Any is used as '#Gadget3D' parameter, the new gadget number will be returned as 'Result'. A TextGadget is a basic text area for displaying, not entering, text.

The content can be changed with the function SetGadgetText3D() and can be obtained with GetGadgetText3D() .

Note: This Gadget doesn't receive any user events, and GadgetToolTip3D() can't be used with it.

Chapter 86

Help

The help feature is a very important component for any type of software. It allows the end-users the opportunity to discover the software's features quickly and easily. The Purebasic help functions allow the programmer the ability to display standard help files with both global and contextual support.

On Microsoft Windows, there are two different help file formats which are supported: The ".hlp" format (which is the old) and the ".chm" format (which is the new). The ".chm" format is based in HTML, which stands for Hyper Text Markup Language.

86.1 CloseHelp

Syntax

```
CloseHelp()
```

Description

Close the help window previously opened with OpenHelp() .

86.2 OpenHelp

Syntax

```
OpenHelp(Filename$, Topic$)
```

Description

Open and display a help window. The Filename$ parameter is the name of either the .chm or .hlp file to be opened. The Topic$ parameter is the internal name of the page to display, if the Topic$ parameter is not a NULL string (i.e. Topic$<>""), the OpenHelp function will display the specified page. This is very useful for online and/or contextual help.

Example:

```
OpenHelp("help.chm", "index.htm")
OpenHelp("help.chm", "print.txt")
```

On Microsoft Windows, there are two different help file formats which are supported: The ".hlp" format (which is the old) and the ".chm" format (which is the new).

Chapter 87

Http

Http is the name of the protocol used by web browsers to access remote information, such as a web page. Each item has an unique address in order to access it from anywhere, this is its: URL (Uniform Resource Locator). The functions within this library are designed to provide easy manipulation of URLs and the capability to retrieve remote files.

87.1 GetHTTPHeader

Syntax

```
Result\$ = GetHTTPHeader(URL$)
```

Description

Get the HTTP headers of the given URL. The content of the headers are dependent on the type of server and therefore their contents vary, but the header that is returned by that server, provides very useful information about the file, such as the date, the server type, the version, and more. The URL must be fully qualified, and must include the "http://" prefix.

InitNetwork() has to be called before using this command.

Example:

```
1  InitNetwork()
2
3  Header$ = GetHTTPHeader("http://www.purebasic.com/index.php3")
4
5  Repeat
6   Index+1
7   Line$ = StringField(Header$, Index, #LF$)
8   Debug Line$
9  Until Line$ = ""
```

The Result$ is a string containing lines, each of which ends with the #LF$ character. StringField() can be used to split it into several strings.

Example of returned headers:

```
HTTP/1.1 200 OK
Date: Fri, 21 Mar 2008 09:49:30 GMT
Server: Apache/1.3.34 (Debian) mod_vhost_online/1.1
 mod_fastcgi/2.4.2 mod_log_online/0.1
```

```
X-Powered-By: PHP/4.4.8-1
Content-Type: text/html
```

87.2 GetURLPart

Syntax

```
Result\$ = GetURLPart(URL$, Parameter$)
```

Description

Get a specific part of the given URL$. A URL$ may contain parameters, which are useful when a scripting language such as: (PHP) is being used on that server. The syntax is as follows: (http://www.purebasic.com/index.php3?test=1) Here, the parameter is named "test" and its associated value is "1". This command will return the value of the given Parameter$. The parameters are not case-sensitive. Moreover, Parameter$ may be one of the following constants, which are used to access a specific part of a URL:

```
#PB_URL_Protocol: returns the protocol from the URL$
#PB_URL_Site: returns the site from the URL$
#PB_URL_Port: returns the port from the URL$ (if specified)
#PB_URL_Parameters: returns all the parameters from the URL$
#PB_URL_Path: returns the path from the URL$
#PB_URL_User: returns the username from the URL$ (if
 specified)
#PB_URL_Password: returns the password from the URL$ (if
 specified)
```

Example:

```
URL$ =
 "http://user:pass@www.purebasic.com:80/index.php3?test=1&ok=2"

Debug GetURLPart(URL$, #PB_URL_Protocol) ; Will print "http"
Debug GetURLPart(URL$, #PB_URL_Site)     ; Will print
 "www.purebasic.com"
Debug GetURLPart(URL$, #PB_URL_Port)     ; Will print "80"
Debug GetURLPart(URL$, #PB_URL_Parameters) ; Will print
 "test=1&ok=2"
Debug GetURLPart(URL$, #PB_URL_Path)     ; Will print
 "index.php3"
Debug GetURLPart(URL$, #PB_URL_User)     ; Will print "user"
Debug GetURLPart(URL$, #PB_URL_Password) ; Will print "pass"
Debug GetURLPart(URL$, "test")           ; Will print "1"
Debug GetURLPart(URL$, "ok")             ; Will print "2"
```

This command can be very useful to parse a URL when writing a CGI program.

87.3 ReceiveHTTPFile

Syntax

```
Result = ReceiveHTTPFile(URL$, Filename$)
```

Description

Download a file to disk from the given URL$. If the URL$ is not found or another error occured, 'Result' will be 0.

InitNetwork() has to be called before using this command.

Example:

```
InitNetwork()

Filename$ = SaveFileRequester("Where to save index.php3 ?", "", "", 0)

If ReceiveHTTPFile("http://www.purebasic.com/index.php3", Filename$)
  Debug "Success"
Else
  Debug "Failed"
EndIf
```

87.4 URLDecoder

Syntax

```
Result\$ = URLDecoder(URL$)
```

Description

Returns a decoded URL$ which has been encoded with the HTTP format.

A URL$ may not contain certain characters such as: tab, space, accent letter etc., therefore these characters must be encoded, which basically involves using "%" as an escape character. If the URL$ is not in an encoded format, this function will have no effect on the given "URL$" and the return value of that "URL$" will remain unchanged.

Example:

```
Debug URLDecoder("http://www.purebasic.com/test%20with%20space.php3")
; Will print "http://www.purebasic.com/test with space.php3"
```

To encode an URL, use URLEncoder()

This command can be very useful to convert a URL when writing a CGI program.

87.5 URLEncoder

Syntax

```
Result\$ = URLEncoder(URL$)
```

Description

Returns the URL$ encoded to the HTTP format.

A URL$ may not contain certain characters such as: tab, space, accent letter etc., therefore these characters must be encoded, which basically involves using "%" as an escape character.

Example:

```
Debug URLEncoder("http://www.purebasic.com/test with space.php3")
; Will print "http://www.purebasic.com/test%20with%20space.php3"
```

To convert an encoded URL back to a unencoded format, use URLDecoder()

This command can be very useful to convert a URL when writing a CGI program.

87.6 SetURLPart

Syntax

```
Result\$ = SetURLPart(URL$, Parameter$, Value$)
```

Description

Set a specific part of the given URL$. A URL$ may contain parameters, which are useful when a scripting language such as: (PHP) is being used on that server. The syntax is as follows: (http://www.purebasic.com/index.php3?test=1) Here, the parameter is named "test" and its associated value is "1". This command will change the value of the given Parameter$.

The parameters are not case-sensitive. Moreover, Parameter$ may be one of the following constants, in order to set a specific part of a URL, the information provided within "URL$" must (at the minimum), be of the following format: ("http://www.purebasic.com")

```
#PB_URL_Protocol: change the protocol of the URL$
#PB_URL_Site: change the site of the URL$
#PB_URL_Port: change the port of the URL$ (or adds it if not specified)
#PB_URL_Parameters: changes all the parameters of the URL$ (or adds it if not specified)
#PB_URL_Path: changes the path of the URL$ (or adds it if not specified)
#PB_URL_User: changes the username of the URL$ (or adds it if not specified)
#PB_URL_Password: changes the password of the URL$  (or adds it if not specified - the "user" needs to exist)
```

Example:

```
URL$ = "http://www.test.com/hello.php3"

URL$ = SetURLPart(URL$, #PB_URL_Protocol, "ftp")
URL$ = SetURLPart(URL$, #PB_URL_Site, "www.purebasic.com")
URL$ = SetURLPart(URL$, #PB_URL_Port, "80")
URL$ = SetURLPart(URL$, #PB_URL_Path, "english/index.php3")
URL$ = SetURLPart(URL$, #PB_URL_User, "user")
URL$ = SetURLPart(URL$, #PB_URL_Password, "pass")
URL$ = SetURLPart(URL$, "test", "1")
URL$ = SetURLPart(URL$, "ok", "2")

Debug URL$ ; Will print "ftp://user:pass@www.purebasic.com:80/english/index.php3?test=1&ok=2"
```

Chapter 88

Image

Images are graphics objects which can be displayed in a window or in several gadgets. PureBasic supports beside BMP and ICON (.ico, Windows only) image types all other image types, which are supported by the ImagePlugin Library .

Transparent PNG images can be used to enable transparency in the gadgets , menu and toolbars images. On Windows, transparent ICON images can be used as well but PNG should be prefered as it works on all platforms.

88.1 CatchImage

Syntax

```
Result = CatchImage(#Image, MemoryAddress [, Size])
```

Description

Load the specified image from the given memory area. If #PB_Any is used as '#Image' parameter, the new image number will be returned as 'Result'. When an image is loaded, it is converted either in 24 bit (if the image depth is less or equal to 24 bit) or in 32 bit (if the image has an alpha-channel).

The image can be in BMP, icon (.ico, only on Windows) or any other format supported by the ImagePlugin library. If something wrong the 0 value is returned. A loaded image can be freed by using the FreeImage() function.

This function is useful when using the 'IncludeBinary' PureBasic keyword. Then images can be packed inside the executable. Nevertheless, use this option with care, as it will take more memory than storing the file in an external file (the file are both in executable memory and load in physical memory).

The optional 'Size' parameter specify the maximum size (in bytes) of the image to load. It can prevent a crash when trying to handle images from an unknown location, which can be corrupted.

The following functions can be used to enable automatically more image formats:

UseJPEGImageDecoder()
UseJPEG2000ImageDecoder()
UsePNGImageDecoder()
UseTIFFImageDecoder()
UseTGAImageDecoder()

Example:

```
CatchImage(0, ?Logo)
End

DataSection
  Logo: IncludeBinary "Logo.bmp"
EndDataSection
```

Note: The "?" is a pointer to a label. More informations about pointers and memory access can be found in the relating chapter here . The image size is currently limited to 8192x8192 pixels (if the available memory allow it). On Windows, if the specified image is an icon (.ico), the following commands will not be available to manipulate it: SaveImage() and @fastimage"ImageOutput()".

88.2 CopyImage

Syntax

```
Result = CopyImage(#Image1, #Image2)
```

Description

Create a new #Image2 identical to the source #Image1. If #PB_Any is used as '#Image2' parameter, the new image number will be returned as 'Result'.

88.3 CreateImage

Syntax

```
Result = CreateImage(#Image, Width, Height [, Depth])
```

Description

Create an empty image (with black background) which can be used to do rendering on it. If #PB_Any is used as '#Image' parameter, the new image number will be returned as 'Result'.

Valid 'Depth' values can be: 24 bit and 32 bit. The default is 24 bit if no depth is specified. The depth value can be combined with #PB_Image_Transparent (using bitwise or operator '||') to create an image with the alpha channel set to full transparent. This only has an effect on 32 bit images.

You can use the several other functions for acting with the created image:
StartDrawing() with ImageOutput() to draw on the created image
CopyImage() to create another image from the actual one
GrabImage() to create another image from a given area of the actual one
DrawImage() with ImageID() to draw the image on actual output channel.
ImageGadget() for displaying image on an application window
ButtonImageGadget() for creating an image button on an application window

Note: the image size is currently limited to 8192x8192 pixels (if the available memory allow it).

88.4 FreeImage

Syntax

```
FreeImage(#Image)
```

Description

Free the specified #Image and release its associated memory.

Note: all remaining images are automatically freed when the program ends.

88.5 GrabImage

Syntax

```
Result = GrabImage(#Image1, #Image2, x, y, Width, Height)
```

Description

Create a new #Image2 with the selected area on the source #Image1. If #PB_Any is used as '#Image2' parameter, the new image number will be returned as 'Result'.

88.6 ImageDepth

Syntax

```
Result = ImageDepth(#Image [, Flags])
```

Description

Returns the depth of the #Image, as it is stored internally by PureBasic. When loading an image, PureBasic converts it either in 24 bit or in 32 bit, depending if the source image got an alpha-channel or not. Every image which have a depth less than 24 bit will be internally converted to 24 bit.

The optional 'Flags' parameter can be one of the following value:

```
PB_Image_InternalDepth: Returns the depth of the decoded
 image (default). Valid values are:
                                  - 24 bit (16 millions
 colors)
                                  - 32 bit (16 millions
 colors + 8 bit alpha channel)
PB_Image_OriginalDepth: Returns the original depth, as it was
 before the decoding. Valid values are:
                                  - 1 bit (2 colors,
 monochrome image)
                                  - 4 bit (16 colors)
                                  - 8 bit (256 colors)
                                  - 16 bit (65536 colors)
                                  - 24 bit (16 millions
 colors)
                                  - 32 bit (16 millions
 colors + 8 bit alpha channel)
```

88.7 ImageHeight

Syntax

```
Result = ImageHeight(#Image)
```

Description

Returns the height, in pixel, of the #Image.

88.8 ImageID

Syntax

```
Result = ImageID(#Image)
```

Description

Returns the ImageID of the #Image.

88.9 ImageOutput

Syntax

```
OutputID = ImageOutput(#Image)
```

Description

Returns the OutputID of the #Image to perform 2D rendering operation on it. It will use the PureBasic 2DDrawing library (see StartDrawing()). This function can't be used if the image is an icon.

An image must have been loaded (with LoadImage() or CatchImage()) or newly created (with CreateImage()) before.

Note: the returned 'OutputID' can't be reused. Every StartDrawing()) will need its own call to ImageOutput().

88.10 ImageWidth

Syntax

```
Result = ImageWidth(#Image)
```

Description

Returns the width, in pixel, of the #Image.

88.11 IsImage

Syntax

```
Result = IsImage(#Image)
```

Description

Tests if the given #Image number is a valid and correctly initialized image. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than an image is ready to use.

88.12 LoadImage

Syntax

```
Result = LoadImage(#Image, Filename$ [, Flags])
```

Description

Load the specified image. If #PB_Any is used as '#Image' parameter, the new image number will be returned as 'Result'.

The image can be in BMP, icon (.ico, only on Windows) or any other format supported by the ImagePlugin Library . If the function fails, 0 is returned, else all is fine. When an image is loaded, it is converted either in 24 bit (if the image depth is less or equal to 24 bit) or in 32 bit (if the image has an alpha-channel).

The following functions can be used to enable automatically more image formats:

UseJPEGImageDecoder()
UseJPEG2000ImageDecoder()
UsePNGImageDecoder()
UseTIFFImageDecoder()
UseTGAImageDecoder()

You can use the several other functions for acting with the loaded image:
StartDrawing() with ImageOutput() to draw on the loaded image
CopyImage() to create another image from the actual one
GrabImage() to create another image from a given area of the actual one
DrawImage() with ImageID() to draw the image on actual output channel.
DrawAlphaImage() with ImageID() to draw the image (with alpha channel) on actual output channel.
ImageGadget() for displaying image on an application window
ButtonImageGadget() for creating an image button on an application window

Note: the image size is currently limited to 8192x8192 pixels (if the available memory allow it). On Windows, if the specified image is an icon (.ico), the following commands will not be available to manipulate it: SaveImage() and @fastimage"ImageOutput()".

88.13 ResizeImage

Syntax

```
ResizeImage(#Image, Width, Height [, Mode])
```

Description

Resize the #Image to the given dimension.

The optional 'Mode' parameter can be one of the following value:

```
#PB_Image_Smooth: Resize the image with smoothing (default if
 'Mode' isn't specified).
#PB_Image_Raw   : Resize the image without any interpolation.
```

Note: This function changes the handle of the used image. Therefore it must be newly assigned e.g. to an ImageGadget() with SetGadgetState() . The image size is currently limited to 8192x8192 pixels (if the available memory allow it).

Note: This function doesn't work if the #Image is an icon (.ico).

88.14 SaveImage

Syntax

```
Result = SaveImage(#Image, Filename$ [, ImagePlugin [, Flags [,
    Depth]]])
```

Description

Saves the specified #Image to disk. By default, the saved image will be in BMP format. If function fails, 0 is returned, else all is fine. 'ImagePlugin' is an optional parameter and can be one of the following constant:

```
  #PB_ImagePlugin_BMP       : Save the image in BMP (default)
  #PB_ImagePlugin_JPEG      : Save the image in JPEG
   (UseJPEGImageEncoder()
 has to be used)
  #PB_ImagePlugin_JPEG2000 : Save the image in JPEG2000
   (UseJPEG2000ImageEncoder()
 has to be used)
  #PB_ImagePlugin_PNG       : Save the image in PNG
   (UsePNGImageEncoder()
 has to be used)
```

'Flags' is an optional parameter depending of the plugin used. For now, only the quality setting is supported: a number from 0 (worste quality) to 10 (maximum quality). Only the JPEG and JPEG 2000 plugins currently support it (default quality is set to '7' if no flags are specified).

'Depth' is an optional parameter to set the depth which should be used when saving the picture. Valid values are 1, 4, 8, 24 and 32. The default value is the original image depth. When an image is saved using palettized depth (1, 4 or 8), the following flag is available:

```
  #PB_Image_FloydSteinberg: Apply a Floyd-Steinberg dithering.
```

For now, only BMP and PNG encoders support palettized image format (1, 4 or 8 bit).

Chapter 89

ImagePlugin

PureBasic supports external image formats through the use of a dynamic native plugin system. By use of this system, only the required encoder or decoder will be added to the final executable, this in turn, dramatically reduces the size of the final program.

If for example, the application only requires the JPEG decoder, only that code which deals with the JPEG decoder is used.

Another nice feature is: if several decoders are being used, these decoders will be detected automatically, through the use of automatic image format detection.

The following functions support image plugins:
LoadImage() , CatchImage() , SaveImage() , LoadSprite() , CatchSprite() and SaveSprite() .

89.1 UseJPEGImageDecoder

Syntax

```
UseJPEGImageDecoder()
```

Description

Enables the JPEG (Joint Picture Expert Group) image support for the CatchImage() , LoadImage() , CatchSprite() and LoadSprite() functions. All formats are supported, including the progressive format. The JPEG format makes use of destructive compression (lossy), what this means, is that the picture loses a certain ammount of the original information when that picture is compressed. This compression algorithm is actually one of the best available, it will dramatically decrease the image size without the introduction of alot of visible artifacts.

89.2 UseJPEGImageEncoder

Syntax

```
UseJPEGImageEncoder()
```

Description

Enables the JPEG (Joint Picture Expert Group) image support for the SaveImage() and SaveSprite() functions. The JPEG format makes use of destructive compression (lossy), what this means, is that the picture loses a certain ammount of the original information when that picture is compressed. This compression algorithm is actually one of the best available, it will dramatically decrease the image size without the introduction of alot of visible artifacts.

89.3 UseJPEG2000ImageDecoder

Syntax

```
UseJPEG2000ImageDecoder()
```

Description

Enables the JPEG 2000 image support for the CatchImage() , LoadImage() , CatchSprite() and LoadSprite() functions. The JPEG 2000 format makes use of destructive compression (lossy), what this means, is that the picture loses a certain ammount of the original information when that picture is compressed. This compression algorithm is actually one of the best available, it will dramatically decrease the image size without the introduction of alot of visible artifacts.

89.4 UseJPEG2000ImageEncoder

Syntax

```
UseJPEG2000ImageEncoder()
```

Description

Enables the JPEG 2000 image support for the SaveImage() and SaveSprite() functions. The JPEG 2000 format makes use of destructive compression (lossy), what this means, is that the picture loses a certain ammount of the original information when that picture is compressed. This compression algorithm is actually one of the best available, it will dramatically decrease the image size without the introduction of alot of visible artifacts. This encoder has the capability to encode 32 bit images with alpha channel.

89.5 UsePNGImageDecoder

Syntax

```
UsePNGImageDecoder()
```

Description

Enables the PNG (Portable Network Graphic) image support for the CatchImage() , LoadImage() , CatchSprite() and LoadSprite() functions. All formats are supported, including the progressive format. The PNG format is well known by web designers, as it is now one of the more popular formats. The PNG format makes use of non-destructive compression (lossless) which means that the picture does not lose any information when that picture is compressed. This is actually the best lossless compression algorithm available.

89.6 UsePNGImageEncoder

Syntax

```
UsePNGImageEncoder()
```

Description

Enables the PNG (Portable Network Graphic) image support for SaveImage() and SaveSprite() . The PNG format is well known by web designers, as it is now one of the more

popular formats. The PNG format makes use of non-destructive compression (lossless) which means that the picture does not lose any information when that picture is compressed. This is actually the best lossless compression algorithm available. This encoder has the capability to encode 32 bit images with alpha channel.

89.7 UseTGAImageDecoder

Syntax

```
UseTGAImageDecoder()
```

Description

Enables the TGA (Targa) image support for the CatchImage() , LoadImage() , CatchSprite() and LoadSprite() functions. The TGA format makes use of non-destructive compression (lossless) which means that the picture does not lose any information when it is compressed. The TGA image size is usually large, since the compression is very weak (Run Length Encoding - RLE). At this point, the TGA alpha channel information (if any) is ignored when using this format.

89.8 UseTIFFImageDecoder

Syntax

```
UseTIFFImageDecoder()
```

Description

Enables the TIFF image support for the CatchImage() , LoadImage() , CatchSprite() and LoadSprite() functions. The TIFF format is very complex and internally supports several sub-compression formats such as: JPEG, LZW, etc. All forms of the TIFF format are supported, using this format will result in programs which are quite large.

Chapter 90

Joystick

PureBasic provides a full access to the joysticks which are connected to the computer. This library supports standard joysticks with up to 8 buttons. On Windows, this library uses the DirectX technology.

90.1 InitJoystick

Syntax

```
Result = InitJoystick()
```

Description

Initialize the joystick environment for later use. This function must be called before any other functions within this library. 'Result' returns the number of joysticks which are available. This function attempts to open DirectX. If this function fails while trying to open DirectX, it is most likely due to the fact that DirectX is not available on the system or the version currently installed is out of date. The DirectX recommendations are: DirectX version v3.0 for NT4.0 compatibility or DirectX version v7.0 under all other circumstances.

90.2 ExamineJoystick

Syntax

```
Result = ExamineJoystick()
```

Description

This function must be called before the following functions are used, in order to reflect the actual state of the joysticks: JoystickButton() , JoystickAxisX() , JoystickAxisY() .

90.3 JoystickAxisX

Syntax

```
Result = JoystickAxisX(#Joystick)
```

Description

Returns the #Joystick X axis state. The first #Joystick number starts at 0. The number of available joysticks is returned by the InitJoystick() function.

'Result' will be one of the following values:

-1: Left movement
0: no movement on the X axis
1: Right movement

ExamineJoystick() must be called before this function is used, in order to reflect the actual state of the joysticks:

90.4 JoystickAxisY

Syntax

```
Result = JoystickAxisY(#Joystick)
```

Description

Returns the #Joystick Y axis state. The first #Joystick number starts at 0. The number of available joysticks is returned by the InitJoystick() function.

'Result' will be one of the following values:

-1: Top movement
0: no movement on the Y axis
1: Bottom movement

ExamineJoystick() must be called before this function is used, in order to update the actual joystick state.

90.5 JoystickButton

Syntax

```
Result = JoystickButton(#Joystick, ButtonNumber)
```

Description

Returns the #Joystick button state. The first #Joystick number starts at 0. The number of available joysticks is returned by the InitJoystick() function.

'Result' will be 0 if the specified button is not pressed, else it returns a non null value. Any number of buttons may be pressed at the same time. ExamineJoystick() must be called before this function is used, in order to update the actual button's state. The first button number starts at 1.

Chapter 91

Keyboard

PureBasic provides fast and easy access to the keyboard. This capability should only be used in applications where raw and extremely fast access is required, such as in games for instance. It use the DirectX technology.

This library was created for games and multimedia applications, which need fast keyboard access based on DirectX. For regular Windows applications, it is better to use the AddKeyboardShortcut() function.

91.1 InitKeyboard

Syntax

```
Result = InitKeyboard()
```

Description

Initialize the keyboard environment for later use. If the 'Result' is 0, then the initialization has failed. You must call this function before any other function in this library. If this function fails while trying to open DirectX, it is most likely due to the fact that DirectX is not available on the system or the version currently installed is out of date. The DirectX recommendations are: DirectX version v3.0 for NT4.0 compatibility or DirectX version v7.0 under all other circumstances.

91.2 ExamineKeyboard

Syntax

```
ExamineKeyboard()
```

Description

Update the keyboard state. This function must be called before using the KeyboardInkey() , KeyboardPushed() or KeyboardReleased() commands.

91.3 KeyboardInkey

Syntax

```
String\$ = KeyboardInkey()
```

Description

Returns the last typed character, very useful when efficient keyboard input is required within a gaming application, such as the name in highscore, in game console, etc.).

Example:

```
If InitSprite() And InitKeyboard() And OpenScreen(800,600,16,"")
  Repeat
    FlipBuffers()
    ClearScreen(RGB(0, 0, 0))

    ExamineKeyboard()
    FullText$ + KeyboardInkey()  ; Add the new text to the current one (if any)

    ; If we press the 'Back' key, we will delete the last character
    ;
    If KeyboardReleased(#PB_Key_Back)
      FullText$ = Left(FullText$, Len(FullText$)-1)
    EndIf

    ; Display the result
    ;
    If StartDrawing(ScreenOutput())
      DrawingMode(1)
      FrontColor(RGB(128, 255, 0))
      DrawText(20, 20, "Just type some text...:")
      DrawText(20, 40, FullText$)
      StopDrawing()
    EndIf
  Until KeyboardPushed(#PB_Key_Escape)
EndIf
```

91.4 KeyboardMode

Syntax

```
KeyboardMode(Flags)
```

Description

Changes the current behavior of the keyboard. This function affects KeyboardPushed() and KeyboardReleased() .

'Flags' can be a combination (using the '||' operator) of the following values:

```
#PB_Keyboard_Qwerty          : The keyboard ignores the default language layout and always uses the QWERTY one (default behavior).
#PB_Keyboard_International   : The keyboard uses the default language layout to map the keys (can be useful for non QWERTY keyboards).
```

```
  #PB_Keyboard_AllowSystemKeys: The 'OS' system keys are
   allowed (like Win+R etc.). This can be annoying in
   fullscreen mode
 if the user
          presses on them accidentally.
```

Supported OS

Windows

91.5 KeyboardPushed

Syntax

```
Result = KeyboardPushed(KeyID)
```

Description

Returns 0 if the specified key is not pressed, otherwise it returns a non null value. Any number of keys may be pressed at the same time. The function ExamineKeyboard() must be called before this function in order to update the keyboard state. The keyboard behavior can be changed with KeyboardMode() .

If you want to check, if a specified key has been pushed and released, take a look at the function KeyboardReleased() .

Example:

```
1   If InitSprite() And InitKeyboard() And
     OpenScreen(800,600,16,"")
2     Repeat
3       FlipBuffers()
4
5       If StartDrawing(ScreenOutput())
6         DrawText(0, 0, "Press ESC to quit")
7         StopDrawing()
8       EndIf
9
10      ExamineKeyboard()
11      If KeyboardPushed(#PB_Key_Escape)    ; press Esc to quit
12        End
13      EndIf
14    ForEver
15  EndIf
```

List of the valid KeyID values:

```
  #PB_Key_All    ; All keys are tested. Very useful for any key
   checks.

  #PB_Key_1
  #PB_Key_2
  #PB_Key_3
  #PB_Key_4
  #PB_Key_5
  #PB_Key_6
  #PB_Key_7
```

```
#PB_Key_8
#PB_Key_9
#PB_Key_0

#PB_Key_A
#PB_Key_B
#PB_Key_C
#PB_Key_D
#PB_Key_E
#PB_Key_F
#PB_Key_G
#PB_Key_H
#PB_Key_I
#PB_Key_J
#PB_Key_K
#PB_Key_L
#PB_Key_M
#PB_Key_N
#PB_Key_O
#PB_Key_P
#PB_Key_Q
#PB_Key_R
#PB_Key_S
#PB_Key_T
#PB_Key_U
#PB_Key_V
#PB_Key_W
#PB_Key_X
#PB_Key_Y
#PB_Key_Z

#PB_Key_Escape
#PB_Key_Minus
#PB_Key_Equals
#PB_Key_Back
#PB_Key_Tab
#PB_Key_LeftBracket
#PB_Key_RightBracket
#PB_Key_Return
#PB_Key_LeftControl
#PB_Key_SemiColon
#PB_Key_Apostrophe
#PB_Key_Grave
#PB_Key_LeftShift
#PB_Key_BackSlash
#PB_Key_Comma
#PB_Key_Period
#PB_Key_Slash
#PB_Key_RightShift
#PB_Key_Multiply
#PB_Key_LeftAlt
#PB_Key_Space
#PB_Key_Capital
#PB_Key_F1
#PB_Key_F2
#PB_Key_F3
#PB_Key_F4
#PB_Key_F5
#PB_Key_F6
```

```
#PB_Key_F7
#PB_Key_F8
#PB_Key_F9
#PB_Key_F10
#PB_Key_F11
#PB_Key_F12
#PB_Key_NumLock
#PB_Key_Scroll
#PB_Key_Pad0
#PB_Key_Pad1
#PB_Key_Pad2
#PB_Key_Pad3
#PB_Key_Pad4
#PB_Key_Pad5
#PB_Key_Pad6
#PB_Key_Pad7
#PB_Key_Pad8
#PB_Key_Pad9
#PB_Key_Add
#PB_Key_Subtract
#PB_Key_Decimal
#PB_Key_PadEnter
#PB_Key_RightControl
#PB_Key_PadComma
#PB_Key_Divide
#PB_Key_RightAlt
#PB_Key_Pause
#PB_Key_Home
#PB_Key_Up
#PB_Key_Down
#PB_Key_Left
#PB_Key_Right
#PB_Key_End
#PB_Key_PageUp
#PB_Key_PageDown
#PB_Key_Insert
#PB_Key_Delete
```

91.6 KeyboardReleased

Syntax

```
Result = KeyboardReleased(KeyID)
```

Description

Returns 1 if the specified key has been pushed and released, otherwise it returns 0. This function is very useful for switch key checks, like a 'Pause' key in a game (one time the game is paused, next time it will continue). The function ExamineKeyboard() must be called before this function to update the keyboard state. The keyboard behavior can be changed with KeyboardMode() . For a full list of KeyID values, take a look at KeyboardPushed() .

Example:

```
1  If InitSprite() And InitKeyboard() And
    OpenScreen(800,600,16,"")
2    Paused = #False
```

```
    Repeat
      FlipBuffers()

      If StartDrawing(ScreenOutput())

        ExamineKeyboard()
        If KeyboardReleased(#PB_Key_P)
          If Paused = #False
            Paused = #True
          Else
            Paused = #False
          EndIf
        EndIf

        DrawingMode(0)

        If Paused = #False
          DrawText(20, 20, "Program is running...")
        Else
          DrawText(20, 20, "Program paused...    ")
        EndIf

        StopDrawing()
      EndIf
    Until KeyboardPushed(#PB_Key_Escape)
  EndIf
```

Chapter 92

Library

Libraries are shared OS components which contain specific functions available to the programmer. For example, a library may contain functions designed to handle and manipulate pictures with ease. Each OS provides a number of shared libraries which help to ease the programmers life. With PureBasic, it is possible to access these external libraries not only easily, but also dynamically!

The reason that libraries are so useful, is due to the fact that they are separate from the applications which use them, but at the same time shared between these applications. A library file requires only that it be loaded into memory once, this therefore saves memory, especially when it is a commonly used library. The programmer also benefits, since there is no need to keep re-inventing the wheel every time a clever feature is called for within an application.

In addition, libraries are an excellent place to store procedures which are used by several programs. These libraries are easy to update, and when changes are made, there is no need to change the main executable which makes use of this library.

Under the Windows OS, shared libraries are well known as: 'DLL's, or Dynamic Link Libraries.

92.1 CloseLibrary

Syntax

```
CloseLibrary(#Library)
```

Description

Closes a library previously opened using: OpenLibrary() , and frees any memory associated with this library.

Note: all remaining opened libraries are automatically closed when the program ends.

#Library - The identifying number of the library. This should be the same value which was used to open the library.

This function does not return any value.

92.2 CallCFunction

Syntax

```
Result = CallCFunction(#Library, FunctionName$ [,Parameter1 [,
    Parameter2...]])
```

Description

Calls a function, in the specified library, in such a manner that the parameters are handled in the same fashion as a normal 'C' language function. The library must have previously been opened with the OpenLibrary() function. The function from the DLL is called by using that functions name. If the function is described as 'cdecl' then this function must be used. However, most DLLs used within the Microsoft Windows OS do not use this form, so CallFunction() should be sufficient in most cases.

Any number of parameters may be specified, but the number of these specified parameters, must be identical to the number of parameters required by the function. For example, if a function requires 2 parameters, then 2 parameters must be passed, even if these 2 values are null.
Note: Currently the maximum number of parameters is 20.

#Library - The number of the library from which the function will be called. This number must be identical to that used previously with: OpenLibrary() .

FunctionName$ - A string containing the name of the function to be called.

Parameter1, Parameter2, ... - The parameters for the function to be called. The number of these parameters, must be identical to the number of parameters required by the function, as described in the developer documentation for that library.

Result - The result of the library function which has been called.

Note: This function is not very flexible and does not handle string/float/double/quad parameters or string/float/double/quad returns. The use of: prototypes is now strongly recommended.

92.3 CallCFunctionFast

Syntax

```
Result = CallCFunctionFast(*FunctionPointer [,Parameter1 [,
    Parameter2...]])
```

Description

Calls a function directly, using its address. This pointer may be found using the GetFunction() , GetFunctionEntry() or LibraryFunctionAddress() functions. The use of this function is the fastest method with which to call library functions, especially when the results of a call to: GetFunction() or LibraryFunctionAddress() have been stored. This is due to the fact that this function is not required to search for the name of the library function. This function should be used only if the library function uses the 'C' calling convention. If the function is described as 'cdecl' then the use of this function is required. However, most DLLs used within the Microsoft Windows OS do not use this form, so CallFunctionFast() should be sufficient in most cases.

Any number of parameters may be specified, but the number of these specified parameters, must be identical to the number of parameters required by the function. For example, if a function requires 2 parameters, then 2 parameters must be passed, even if these 2 values are null.
Note: Currently the maximum number of parameters is 20.

*FunctionPointer - The address in memory of the function to be called.

Parameter1, Parameter2, ... - The parameters for the function to be called. The number of these parameters, must be identical to the number of parameters required by the function.

Result - The result of the function that has been called.

Note: This function is not very flexible and does not handle string/float/double/quad parameters or string/float/double/quad returns. The use of: prototypes is now strongly recommended.

92.4 CallFunction

Syntax

```
Result = CallFunction(#Library, FunctionName$ [,Parameter1 [,
    Parameter2...]])
```

Description

Calls a function in the specified library, by using its name. The specified library must have previously been opened with the OpenLibrary() function,

Any number of parameters may be specified, but the number of these specified parameters, must be identical to the number of parameters required by the function. For example, if a function requires 2 parameters, then 2 parameters must be passed, even if these 2 values are null.
Note: Currently the maximum number of parameters is 20.

#Library - The number of the library from which the function will be called. This number must be identical to that used previously with: OpenLibrary() .

FunctionName$ - A string containing the name of the function to be called.

Parameter1, Parameter2, ... - The parameters for the function to be called. The number of these parameters, must be identical to the number of parameters required by the function, as described in the developer documentation for that library.

Result - The result of the function that has been called.

Note: This function is not very flexible and does not handle string/float/double/quad parameters or string/float/double/quad returns. The use of: prototypes is now strongly recommended.

92.5 CallFunctionFast

Syntax

```
Result = CallFunctionFast(*FunctionPointer [,Parameter1 [,
   Parameter2...]])
```

Description

Calls a function directly, using its address. This pointer may be found using the GetFunction() , GetFunctionEntry() or LibraryFunctionAddress() functions. The use of this function is the fastest method with which to call library functions, especially when the results of a call to: GetFunction() or LibraryFunctionAddress() have been stored. This is due to the fact that this function is not required to search for the name of the library function.

This function also gives the programmer the ability to call PureBasic procedures from within the applications code. The address of a PureBasic procedure may be obtained by using the '@'symbol in front of the procedure name, such as "@myprocedure()".

Any number of parameters may be specified, but the number of these specified parameters, must be identical to the number of parameters required by the function. For example, if a function requires 2 parameters, then 2 parameters must be passed, even if these 2 values are null.
Note: Currently the maximum number of parameters is 20.

*FunctionPointer - The address in memory of the function to be called.

Parameter1, Parameter2, ... - The parameters for the function to be called. The number of these parameters, must be identical to the number of parameters required by the function, as described in the developer documentation for that library.

Result - The result of the function that has been called.

Note: This function is not very flexible and does not handle string/float/double/quad parameters or string/float/double/quad returns. The use of: prototypes is now strongly recommended.

92.6 CountLibraryFunctions

Syntax

```
Result = CountLibraryFunctions(#Library)
```

Description

Counts the number of functions available in a library. The library must be open when this function is called.

#Library - The number of the library which contains the functions to count. This number must be identical to that used previously with: OpenLibrary() .

Result - The number of functions available in the library.

Supported OS

Windows, Linux

92.7 ExamineLibraryFunctions

Syntax

```
Result = ExamineLibraryFunctions(#Library)
```

Description

Initiates the process of examining the functions contained within a library. The library must have already been opened using: OpenLibrary() before this function may be called. If this function succeeds, the programmer may then step through the functions in the library by using: NextLibraryFunction() . At that point, information such as the name and address of each function may be obtained by using the LibraryFunctionName() and LibraryFunctionAddress() functions respectively.

#Library - The number of the library which contains the functions to examine. This number must be identical to that used previously with: OpenLibrary() .

Result - This value determines if the library functions may be examined. If the value is zero then they can not be examined. If the value is non-zero then they may be examined.

Supported OS

Windows, Linux

92.8 GetFunction

Syntax

```
Result.l = GetFunction(#Library, FunctionName$)
```

Description

Checks if the library, previously opened using the OpenLibrary() function, contains the given function. Note that the function name is always case sensitive.

#Library - The number of the library which contains the functions to find. This number must be identical to that used previously with: OpenLibrary() .

FunctionName$ - A string containing the name of the function to be found.

Result - The address in memory of the function from the library, which is useful with the CallFunctionFast() or CallCFunctionFast() functions. If the address is zero then the function can not be found and should not be called.

92.9 GetFunctionEntry

Syntax

```
Result = GetFunctionEntry(#Library, FunctionEntry)
```

Description

Checks if the library, previously opened using the OpenLibrary() function, contains the given function entry. This searches for a library function by its position within the libraries function table, rather than by its name, such as is the case with the GetFunction() function. This function is often useful when calling undocumented API functions.

#Library - The number of the library which contains the functions whose entries are to be found. This number must be identical to that used previously with: OpenLibrary() .

FunctionEntry - A number representing the function index in the library. The first function is at index 1.

Result - The address in memory of the function from the library, which is useful in conjuction with the CallFunctionFast() or CallCFunctionFast() functions. If the address is zero then the function can not be found and should not be called.

Supported OS

Windows

92.10 IsLibrary

Syntax

```
Result = IsLibrary(#Library)
```

Description

Tests if the given #Library number is valid and if the library has been correctly initialized. This function is bullet-proof and may be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a library is ready for use.

92.11 LibraryFunctionAddress

Syntax

```
Result.l = LibraryFunctionAddress()
```

Description

Returns the address of the function in the library currently being examined with the ExamineLibraryFunctions() and NextLibraryFunction() functions.

None.

Result - The address in memory of the function from the library, which is useful for passing to the CallFunctionFast() or CallCFunctionFast() functions.

Supported OS

Windows, Linux

92.12 LibraryFunctionName

Syntax

```
Result\$ = LibraryFunctionName()
```

Description

Returns the name of the function in the library currently being examined with the ExamineLibraryFunctions() and NextLibraryFunction() functions.

None.

Result$ - The name of the function from the library, which is useful for passing to the CallFunction() or CallCFunction() functions.

Supported OS

Windows, Linux

92.13 LibraryID

Syntax

```
Result = LibraryID(#Library)
```

Description

Returns the unique ID which identifies the specified #Library in the operating system.

92.14 NextLibraryFunction

Syntax

```
Result = NextLibraryFunction()
```

Description

This function may only be called after successful use of the ExamineLibraryFunctions() function. This function allows the programmer to go step by step through the library and examine its contents. At that point, information such as the name and address of each function may be obtained by using the LibraryFunctionName() and LibraryFunctionAddress() functions respectively.

None.

Result - Zero if there are no more functions available to be examined (end of listing). If it is non-zero, then additional functions remain to be examined.

Supported OS

Windows, Linux

92.15 OpenLibrary

Syntax

```
Result.l = OpenLibrary(#Library, Filename$)
```

Description

Opens a shared library in order that the functions within it may be accessed. If a library is opened with the same identifying number as another library which was already open, the library previously opened will then be automatically closed.

#Library - The identifying number for this library. If #PB_Any is used as the '#Library' parameter, the new library number will be returned as 'Result'. This number should then be used in all function calls which make use of this library.

Filename$ - A string containing the filename of the shared library to open. Typically, if a path is not provided as part of the filename, then the library will be searched for within the application directory. If at this point, the library still has not been found, the system library directory will be searched.

Result - A value which determines whether the library could be opened. If the value is zero, then the library could not be opened. This indicates that either the file could not be found, or that the file is corrupt. Any non-zero value indicates the library has been successfully opened.

Advanced users only:
The value returned by this function, is the OS handle or pointer to the library, which then becomes available after the library is loaded. At that point, the programmer has the ability to use the opened library directly, using API function calls.

Chapter 93

Light

Lights are essential components in order to create worlds which look realistic. As with real life light, it is possible to specify many attributes of this light such as: the color, specular color, direction and more. InitEngine3D() should be called successfully before using the Light functions.

93.1 CopyLight

Syntax

```
Result = CopyLight(#Light, #NewLight)
```

Description

Creates a #NewLight which is the exact copy of the specified #Light. All the #Light attributes such as: color, specular color, position etc.. are duplicated. If #PB_Any is used as '#NewLight' parameter, the new light number will be returned as 'Result'. If #NewLight was already created, it will be automatically freed and replaced by the new one.

93.2 CreateLight

Syntax

```
Result = CreateLight(#Light, Color [, x, y, z])
```

Description

Creates a new #Light of the given Color in the current world. If #PB_Any is used as '#Light' parameter, the new light number will be returned as 'Result'.

Valid colors may be easily created using the RGB() function. The new light is created at the 0,0,0 location unless the optional 'x,y,z' paremeters are specified. If the #Light was already created, it is automatically freed and replaced by the new one.

Example:

```
CreateLight(0, RGB(255,0,0)) ; Creates a red light

CreateLight(1, RGB(0,255,0),  0, 100.7, 50) ; Creates a green light, at the position (0, 100.7, 50)
```

93.3 FreeLight

Syntax

```
FreeLight(#Light)
```

Description

Frees the specified #Light. All its' associated memory is released and at this point, the object no longer may be used.

Note: all remaining lights are automatically freed when the program ends.

93.4 HideLight

Syntax

```
HideLight(#Light, State)
```

Description

Hides or Shows the specified #Light. 'State' can take the following values:

```
1: the #Light is hidden
0: the #Light is shown
```

93.5 IsLight

Syntax

```
Result = IsLight(#Light)
```

Description

Tests if the given #Light is a valid and correctly initialized light.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that a light is ready to use.

93.6 LightColor

Syntax

```
LightColor(#Light, Color)
```

Description

Changes the #Light color. Valids colors may be created easily with the RGB() function.

93.7 LightLocate

Syntax

```
LightLocate(#Light, x, y, z)
```

Description

Changes the #Light absolute location in the world.

93.8 LightSpecularColor

Syntax

```
LightSpecularColor(#Light, Color)
```

Description

Changes the #Light specular color. Valids colors may be created easily with the RGB() function.

93.9 MoveLight

Syntax

```
MoveLight(#Light, x, y, z)
```

Description

Moves the #Light in the world according to the specified x,y,z values. This is a relative move based on the previous Light location. To perform an absolute move in the world, use LightLocate() .

Chapter 94

LinkedList

Linked Lists are structures for storing data which are dynamically allocated depending of your need. It is a list of elements (the data you want to store) and each element is fully independant of the others. You can add as many elements you want (or as many as will fit into the memory of your computer), insert elements at the position you need, delete some other and more. This kind of data management is very useful as it's one of the best ways to handle data when you do not know how many elements you will need to store, or if you are often changing how many elements there are.

Before you can work with LinkedLists, you must declare them first. This could be done with the keyword NewList . For saving the contents are also often used structures .

To specifically search the contents of a LinkedList, using of loops is recommended: For : Next , ForEach : Next , Repeat : Until or While : Wend .

94.1 AddElement

Syntax

```
AddElement(LinkedList())
```

Description

Adds a new empty element after the current element or as the first item in the list if there are no elements in it. This new element becomes the current element of the list.

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
The value returned by this function can be used to show whether the new element could be created or not (it may not be created if, for example, you have no more free memory on your computer). If the new element was created, this function will return a value which is not equal to zero. If the new element could not be created, then it will return a value of zero.

Advanced users only:
The value that this function returns is a pointer to the new element data or zero if the new element could not be created.

Example:

```
; The simplest way to use AddElement
NewList simple.w()
AddElement(simple())     ; Creates the first new element in
 the list
```

```
simple() = 23

AddElement(simple())    ; Current position is the first element, so we add one to the second position
simple() = 45


; This shows how to use the return value of AddElement
NewList advanced.l()
If AddElement(advanced()) <> 0
  advanced() = 12345
Else
  MessageRequester("Error!", "Unable to allocate memory for new element", #PB_MessageRequester_OK)
EndIf


; A small structure to demonstrate the "advanced users" description (above)
Structure Programmer
  Name.s
  Strength.b
EndStructure

NewList Programmers.Programmer()  ; The list for storing the elements

*Element.Programmer = AddElement(Programmers())
If *Element<>0
  *Element\Name = "Dave"
  *Element\Strength = 3   ; Wow, super-strong geek! ;)
Else
  MessageRequester("Error!", "Unable to allocate memory for new element", #PB_MessageRequester_OK)
EndIf
```

94.2 ChangeCurrentElement

Syntax

```
ChangeCurrentElement(LinkedList(), *NewElement)
```

Description

Changes the current element of the specified list to the given new element. The new element must be a pointer to another element which exists in this list. This function is very useful if you want to "remember" an element, and restore it after performing other processing. It should be used with care and only by advanced users.

Parameters:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.
NewElement - The new element to set as the current element for the linked list. You should get this address by using the @ operator on the list name and not through any other method.

Return value:
This function does not return anything.

Example:

```
*Old_Element = @mylist()    ; Get the address of the current element

ResetList(mylist())         ; Perform a search for all elements named
While NextElement(mylist())    ; "John" and change them to "J"
  If mylist()\name = "John"
    mylist()\name = "J"
  EndIf
Wend

ChangeCurrentElement(mylist(), *Old_Element) ; Restore previous current element (from before the search)
```

94.3 ClearList

Syntax

```
ClearList(LinkedList())
```

Description

Clears all the elements in this list and releases their memory. After this call the list is still usable, but the list is empty (i.e. there are no elements in it).

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value: This function does not return a value.

Example:

```
NewList Numbers.w()

; A small loop to add many items to the list
For i=1 To 100
  AddElement(Numbers())
  Numbers() = i
Next

; Proof that items have been added to the list
MessageRequester("Information", "There are "+Str(ListSize(Numbers()))+" elements in the list", #PB_MessageRequester_OK)

; Clear the list and show that the list really is empty
ClearList(Numbers())
MessageRequester("Information", "There are "+Str(ListSize(Numbers()))+" elements in the list", #PB_MessageRequester_OK)
```

Note:
PureBasic will only free the memory for the elements. If you have been using the linked list for something such as storing handles of objects that you create directly with the OS, there is no way PureBasic (or any other language) can know what they are. Therefore, in cases such as that, you should go through the elements in the list and free the objects yourself.

94.4 CopyList

Syntax

```
Result = CopyList(SourceList(), DestinationList())
```

Description

Copy every elements of 'SourceList()' into 'DestinationList()'. If the two lists are not of the same type (native or structured) the copy will fail and 'Result' will be 0. Every elements previously found in 'DestinationList()' will be deleted. After a successful copy, the two lists are identicals.

Example:

```
NewList Friends$()
NewList FriendsCopy$()

AddElement(Friends$())
Friends$() = "John"

AddElement(Friends$())
Friends$() = "Elise"

CopyList(Friends$(), FriendsCopy$())

ForEach FriendsCopy$()
  Debug FriendsCopy$()
Next
```

94.5 FreeList

Syntax

```
FreeList(List())
```

Description

Free the specified 'List()' and release all its associated memory. To use it again NewList has to be called.

94.6 ListSize

Syntax

```
Result = ListSize(LinkedList())
```

Description

Returns the number of elements in the linked list. It does not change the current element. This function is very fast (it doesn't iterates all the list but uses a cached result) and can be safely used to determine if a list is empty or not.

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
The total number of elements in the linked list.

Example:

```
NewList countme.w()

; Small loop to add some elements to the list.
For i=0 To 10
  AddElement(countme())
  countme() = i * 23
Next

; Show how many elements there are in the list. I hope you thought
; of the same value as this example ;)
MessageRequester("Information", "There are "+Str(ListSize(countme()))+" elements in the list", #PB_MessageRequester_OK)
```

94.7 DeleteElement

Syntax

```
DeleteElement(LinkedList() [, Flags])
```

Description

Remove the current element from the list. After this call, the new current element is the previous element (the one before the deleted element). If that element does not exist (in other words, you deleted the first element in the list) then there is no more current element, as it will be before the first element, like after a ResetList() . But, if the parameter 'Flags' is set to 1 and the first element is deleted, the new current element will be the second one. This flag ensure than there will be always a valid current element after a delete. If there was only one element in the list when you deleted it then you are left with no current element!

Parameters:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
This function does not return any value.

Example:

```
NewList people.s()

AddElement(people()) : people() = "Tom"
AddElement(people()) : people() = "Dick"
AddElement(people()) : people() = "Harry"
AddElement(people()) : people() = "Bob"

FirstElement(people())
DeleteElement(people(), 1)
MessageRequester("Information", "First person in list is "+people(), #PB_MessageRequester_Ok)

LastElement(people()) ; Moves to "Bob"
PreviousElement(people()) ; Moves to "Harry"
DeleteElement(people()) ; And deletes him. there is an element before Harry, so it becomes the current
```

```
MessageRequester("Information", "Current person in list is "+people(), #PB_MessageRequester_Ok)
```

94.8 FirstElement

Syntax

```
FirstElement(LinkedList())
```

Description

Changes the current list element to the first list element.

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
The value returned by this function can be used to show whether the first element exists or not (it will not exist if the list is empty). If the first item exists, this function will return a value which is not equal to zero. If the first element does not exist then it will return a value of zero.

Advanced users only:
The value that this function returns is a pointer to the first element data or zero if the first element does not exist.

Example:

```
; An example of simple usage
NewList Numbers.w()

AddElement(Numbers())
Numbers() = 5
AddElement(Numbers())
Numbers() = 8

FirstElement(Numbers())
MessageRequester("Information", "First element value is "+Str(Numbers()), #PB_MessageRequester_OK)


; An example which uses the return value
NewList Numbers.w()

If FirstElement(Numbers()) <> 0
  MessageRequester("Information", "First element value is "+Str(Numbers()), #PB_MessageRequester_OK)
Else
  MessageRequester("Information", "List is empty", #PB_MessageRequester_OK)
EndIf

AddElement(Numbers())
Numbers() = 5
AddElement(Numbers())
Numbers() = 8

If FirstElement(Numbers()) <> 0
  MessageRequester("Information", "First element value is "+Str(Numbers()), #PB_MessageRequester_OK)
```

```
Else
  MessageRequester("Information", "List is empty", #PB_MessageRequester_OK)
EndIf


; An example which is only for advanced users
NewList Numbers.w()

AddElement(Numbers())
Numbers() = 5
AddElement(Numbers())
Numbers() = 8

*Element.Word = FirstElement(Numbers())
If *Element
  MessageRequester("Information", "First element value is "+Str(*Element\w), #PB_MessageRequester_OK)
Else
  MessageRequester("Information", "List is empty", #PB_MessageRequester_OK)
EndIf
```

94.9 InsertElement

Syntax

```
InsertElement(LinkedList())
```

Description

Inserts a new empty element before the current element, or at the start of the list if the list is empty (i.e. has no elements in it). This new element becomes the current element of the list.

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
The value returned by this function can be used to show whether the new element could be created or not (it may not be created if, for example, you have no more free memory on your computer). If the new element was created, this function will return a value which is not equal to zero. If the new element could not be created, then it will return a value of zero.

Advanced users only:
The value that this function returns is a pointer to the new element data or zero if the new element could not be created.

Example:

```
; The simplest way to use InsertElement
NewList simple.w()
InsertElement(simple())     ; Creates the first new element in the list
simple() = 23

InsertElement(simple())     ; Current position is the first element, so we add this element to the start of the list
simple() = 45               ; The old first element is now the second element in the list

```

```

; This shows how to use the return value of InsertElement
NewList advanced.l()
If InsertElement(advanced()) <> 0
  advanced() = 12345
Else
  MessageRequester("Error!", "Unable to allocate memory for new element", #PB_MessageRequester_OK)
EndIf


; A small structure to demonstrate the "advanced users" description (above)
Structure Programmer
  Name.s
  Strength.b
EndStructure

NewList Programmers.Programmer()  ; The list for storing the elements

*Element.Programmer = InsertElement(Programmers())
If *Element<>0
  *Element\Name = "Dave"
  *Element\Strength = 3   ; Wow, super-strong geek! ;)
Else
  MessageRequester("Error!", "Unable to allocate memory for new element", #PB_MessageRequester_OK)
EndIf
```

94.10 LastElement

Syntax

```
LastElement(LinkedList())
```

Description

Change the current list element to the last list element.

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
The value returned by this function can be used to show whether the last element exists or not (it will not exist if the list is empty). If the last item exists, this function will return a value which is not equal to zero. If the last element does not exist then it will return a value of zero.

Advanced users only:
The value that this function returns is a pointer to the last element data or zero if the last element does not exist.

Example:

```
; An example of simple usage
NewList Numbers.w()

AddElement(Numbers())
```

```
Numbers() = 5
AddElement(Numbers())
Numbers() = 8

LastElement(Numbers())
MessageRequester("Information", "Last element value is "+Str(Numbers()), #PB_MessageRequester_OK)


; An example which uses the return value
NewList Numbers.w()

If LastElement(Numbers()) <> 0
  MessageRequester("Information", "Last element value is "+Str(Numbers()), #PB_MessageRequester_OK)
Else
  MessageRequester("Information", "List is empty", #PB_MessageRequester_OK)
EndIf

AddElement(Numbers())
Numbers() = 5
AddElement(Numbers())
Numbers() = 8

If LastElement(Numbers()) <> 0
  MessageRequester("Information", "Last element value is "+Str(Numbers()), #PB_MessageRequester_OK)
Else
  MessageRequester("Information", "List is empty", #PB_MessageRequester_OK)
EndIf


; An example which is only for advanced users
NewList Numbers.w()

AddElement(Numbers())
Numbers() = 5
AddElement(Numbers())
Numbers() = 8

*Element.Word = LastElement(Numbers())
If *Element
  MessageRequester("Information", "Last element value is "+Str(*Element\w), #PB_MessageRequester_OK)
Else
  MessageRequester("Information", "List is empty", #PB_MessageRequester_OK)
EndIf
```

94.11 ListIndex

Syntax

```
Index = ListIndex(LinkedList())
```

Description

Find out the position of the current element in the list, considering that the first element is at the position 0. This function is very fast, and can be used a lot without performance issue (it doesn't iterate the list but uses a cached value).

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value: A number containing the position of the current element within the list. The first element is at position 0, the next at 1 and so on. A value of -1 means there is no current element (either the list is empty or ResetList() has been used).

Example:

```
NewList fruit.s()

AddElement(fruit()) : fruit() = "oranges"
AddElement(fruit()) : fruit() = "bananas"
AddElement(fruit()) : fruit() = "apples"
AddElement(fruit()) : fruit() = "pears"

FirstElement(fruit())
MessageRequester("Fruit: "+fruit(), "Now at position
 "+Str(ListIndex(fruit())),  #PB_MessageRequester_OK)

NextElement(fruit())
MessageRequester("Fruit: "+fruit(), "Now at position
 "+Str(ListIndex(fruit())),  #PB_MessageRequester_OK)

NextElement(fruit())
MessageRequester("Fruit: "+fruit(), "Now at position
 "+Str(ListIndex(fruit())),  #PB_MessageRequester_OK)

NextElement(fruit())
MessageRequester("Fruit: "+fruit(), "Now at position
 "+Str(ListIndex(fruit())),  #PB_MessageRequester_OK)
```

94.12 NextElement

Syntax

```
Result = NextElement(LinkedList())
```

Description

Moves from the current element to the next element in the list, or onto the first element if you have previously called ResetList()

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
The value returned by this function can be used to show whether the next element exists or not (it will not exist if you are at the end of the list). If the next item exists, this function will return a value which is not equal to zero. If the next element does not exist then it will return a value of zero.

Advanced users only:

The value that this function returns is a pointer to the next element data or zero if the next element does not exist.

Example:

```
NewList Scores.w()

For i=1 To 10
  AddElement(Scores())
  Scores() = 100 - i
Next

ResetList(Scores())
While NextElement(Scores())
  ; This is OK since the first call to NextElement() will move the current element to the first item in the list
  MessageRequester("Score", Str(Scores()), #PB_MessageRequester_OK)
Wend
```

94.13 PreviousElement

Syntax

```
Result = PreviousElement(LinkedList())
```

Description

Change the current list element with the previous element and return its address or return 0 if the current element is the first element or if no elements are in the list..

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
The value returned by this function can be used to show whether the previous element exists or not (it will not exist if you are at the start of the list). If the previous item exists, this function will return a value which is not equal to zero. If the previous element does not exist then it will return a value of zero.

Advanced users only:
The value that this function returns is a pointer to the previous element data or zero if the previous element does not exist.

Example:

```
NewList Numbers.w()

For i=1 To 10
  AddElement(Numbers())
  Numbers() = i
Next

Repeat
  MessageRequester("Number", Str(Numbers()), #PB_MessageRequester_OK)
Until PreviousElement(Numbers()) = 0
```

94.14 ResetList

Syntax

```
ResetList(LinkedList())
```

Description

Resets the current list element to be before the first element. This means than no element is actually valid. However, this is very useful to allow you to process all the elements by using NextElement() .

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.

Return value:
This function does not return any value.

Example:

```
NewList Friends.s()

AddElement(Friends())
Friends() = "Arnaud"

AddElement(Friends())
Friends() = "Seb"

ResetList(Friends())
While NextElement(Friends())
  Debug Friends() ; Display all the list elements
Wend
```

94.15 SelectElement

Syntax

```
SelectElement(LinkedList(), Position)
```

Description

Change the current list element to the element at the specified position. The first list element is at position 0, the next at position 1 and so on. This is very useful if you want to jump to a specific position in the list without using an own loop for this.

As linked lists don't use an index, it will jump compulsory to every element until the target position is reached. If a faster method is needed, ChangeCurrentElement() should be used.

Parameter:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.
Position - The position to move to in the list, considering that the first item in the list is at position 0, the next is at 1 and so on. You must make sure that you do not specify a position that is outside of the number of elements in the list!

Return value:
This function does not return any value.

Example:

```
NewList mylist.l()

AddElement(mylist()) : mylist() = 23
AddElement(mylist()) : mylist() = 56
AddElement(mylist()) : mylist() = 12
AddElement(mylist()) : mylist() = 73

SelectElement(mylist(), 0)
MessageRequester("Position", "At position 0, the value is "+Str(mylist()),0)

SelectElement(mylist(), 2)
MessageRequester("Position", "At position 2, the value is "+Str(mylist()),0)

SelectElement(mylist(), 1)
MessageRequester("Position", "At position 1, the value is "+Str(mylist()),0)

SelectElement(mylist(), 3)
MessageRequester("Position", "At position 3, the value is "+Str(mylist()),0)
```

94.16 SwapElements

Syntax

```
SwapElements(LinkedList(), *FirstElement, *SecondElement)
```

Description

Swaps 2 elements of the specified list. The '*FirstElement' and '*SecondElement' parameters must be pointers to valid list elements. This function is very useful if you want to sort or reorganize quickly a list. It should be used with care and only by advanced users.

Parameters:
LinkedList() - The name of your linked list variable, created with the NewList function. You must include the brackets after the list name.
FirstElement - Address of the first element to swap. You can get this address by using the @ operator on the list name.
SecondElement - Address of the second element to swap. You can get this address by using the @ operator on the list name.

Return value:
This function does not return anything.

Example:

```
NewList Numbers()

For k=0 To 10
  AddElement(Numbers())
  Numbers() = k
Next

SelectElement(Numbers(), 3) ; Get the 4th element (first element is 0)
*FirstElement = @Numbers()

```

```
SelectElement(Numbers(), 9) ; Get the 10th element
*SecondElement = @Numbers()

; Swap the 3 with the 9
;
SwapElements(Numbers(), *FirstElement, *SecondElement)

; Prove it
;
ForEach Numbers()
  Debug Numbers()
Next
```

Chapter 95

Mail

Mail is now a common way for information to be exchanged between two distant computers. The functions within this library allow for not only the creation of mail, but also the ability send this mail to either a single recipient or to multiple recipients. In addition, optional attachments may be made when desired. This may then be sent through a mail server.

All the mail commands are based on the network library , therefore the: InitNetwork() function must be called before any of them are used.

95.1 AddMailAttachment

Syntax

```
Result = AddMailAttachment(#Mail, Description$, Filename$ [,
    MimeType$])
```

Description

Add a file attachement to the #Mail. 'Description$' will be the information string displayed in the mail. 'Filename$' is the file to be added as an attachement. Once the attachment is added, the local file may changed or deleted, as the entire content of the file is copied and attached to the mail. If the attachment can not be created, 'Result' will be 0.

The optional 'MimeType$' parameter allows for the specification of which type of file is to be attached. If this parameter is omitted, then the file extension will be used to determine that files mime type. Below there is a list of the available mime types. If the extension of the file does not match any of the available mime types, then the "application/octet-stream" mime type will be used.

```
application/acad          | AutoCAD    dwg
application/clariscad     | ClarisCAD  ccad
application/drafting      | MATRA Prelude drafting   drw
application/dxf           | AutoCAD  dxf
application/i-deas        | SDRC I-deas  unv
application/iges          | Format d'échange CAO IGES
 igs,iges
application/oda           | ODA  oda
application/pdf           | Adobe Acrobat  pdf
application/postscript    | PostScript   ai,eps,ps
application/pro_eng       | ProEngineer  prt
application/rtf           | Rich text   rtf
application/set           | CAO SET  set
application/sla           | stéréolithographie   stl
```

```
application/solids         |  MATRA Solids   dwg
application/step           | de données STEP  step
application/vda            | de surface   vda
application/x-mif          | Framemaker   mif
application/x-csh          | Script C-Shell (UNIX)   dwg
application/x-dvi          | texte dvi  dvi
application/hdf            | de données   hdf
application/x-latex        | LaTEX  latex
application/x-netcdf       | netCDF   nc,cdf
application/x-sh           | Script Bourne Shell   dwg
application/x-tcl          | Script Tcl  tcl
application/x-tex          | fichiers Tex  tex
application/x-texinfo      | eMacs  texinfo,texi
application/x-troff        | Troff  t,tr,troff
application/x-troff-man    | Troff/macro man  man
application/x-troff-me     | Troff/macro ME   me
application/x-troff-ms     | Troff/macro MS   ms
application/x-wais-source  | Source Wais   src
application/x-bcpio        | CPIO binaire  bcpio
application/x-cpio         | CPIO Posix  cpio
application/x-gtar         | Tar GNU   gtar
application/x-shar         | Archives Shell  shar
application/x-sv4cpio      | CPIO SVR4n  sv4cpio
application/x-sv4crc       | CPIO SVR4 avec CRC  sc4crc
application/x-tar          | compressés tar   tar
application/x-ustar        | compressés tar Posix   man
application/zip            | compressés ZIP   man
audio/basic                | audio basiques   au,snd
audio/x-aiff               | audio AIFF   aif,aiff,aifc
audio/x-wav                | audio Wave   wav
image/gif                  | Images gif  man
image/ief                  | Images exchange format  ief
image/jpeg                 | Images Jpeg   jpg,jpeg,jpe
image/png                  | Images Png  png
image/tiff                 | Images Tiff   tiff,tif
image/x-cmu-raster         | Raster cmu  cmu
image/x-portable-anymap    | Anymap PBM   pnm
image/x-portable-bitmap    | Bitmap PBM   pbm
image/x-portable-graymap   | Graymap PBM  pgm
image/x-portable-pixmap    | Pixmap PBM   ppm
image/x-rgb                | Image RGB   rgb
image/x-xbitmap            | Images Bitmap X   xbm
image/x-xpixmap            | Images Pixmap X   xpm
image/x-xwindowdump        | Images dump X Window  man
multipart/x-zip            | archive zip  zip
multipart/x-gzip           | archive GNU zip  gz,gzip
text/html                  | HTML   htm,html
text/plain                 | texte sans mise en forme
 txt,g,h,c,cc,hh,m,f90
text/richtext              | texte enrichis   rtx
text/tab-separated-value   | texte avec séparation des valeurs
  tsv
text/x-setext              | texte Struct   etx
video/mpeg                 | Vidéos MPEG   mpeg,mpg,mpe
video/quicktime            | Vidéos QuickTime  qt,mov
video/msvideo              | Vidéos Microsoft Windows  avi
video/x-sgi-movie          | Vidéos MoviePlayer  movie
```

Note: any number of attachments may be added to the mail, but the size limit available for a

single attachment is currently set at 100MB. Most servers and clients do not have the capacity to handle attachments of that size, therefore it is advised that each attachment be kept to a reasonable size

Example:

```
InitNetwork()

If CreateMail(0, "test@purebasic.com", "Hello")
  If AddMailAttachment(0, "Attachment 1",
 OpenFileRequester("Choose an attachment", "", "", 0))
    Debug "Attachment correctly added"
  Else
    Debug "Attachment failed"
  EndIf
EndIf
```

95.2 AddMailAttachmentData

Syntax

```
Result = AddMailAttachmentData(#Mail, Description$, *Buffer,
   BufferLength [, MimeType$])
```

Description

Add memory data as an attachement to the #Mail. 'Description$' will be the information string displayed in the mail. '*Buffer' and 'BufferLength' specify the memory area and its size when added to the mail as an attachment. Once the attachment has been added, the memory area may be changed or freed, since its entire contents are copied into the mail. If the attachment can not be created, 'Result' will be 0.

More information about the optional 'MimeType$' parameter may be found at AddMailAttachment() .

Note: any number of attachments may be added to the mail, but the size limit available for a single attachment is currently set at 100MB. Most servers and clients do not have the capacity to handle attachments of that size, therefore it is advised that each attachment be kept to a reasonable size

Example:

```
InitNetwork()

If CreateMail(0, "test@purebasic.com", "Hello")

  If AddMailAttachmentData(0, "Attachment 1", ?Hello, 5)
    Debug "Attachment correctly added"
  Else
    Debug "Attachment failed"
  EndIf
EndIf


DataSection
  Hello:
    Data.b 'H', 'e', 'l', 'l', 'o'
```

95.3 AddMailRecipient

Syntax

```
AddMailRecipient(#Mail, Address$, Flags)
```

Description

Add a recipient to the specified #Mail. 'Address$' must be in a valid address format or the mail will not be sent correctly. 'Flags' may be a combination the following values:

```
#PB_Mail_To : Main(s) recipient(s) of the mail
#PB_Mail_Cc : Recipient(s) which are in copy of the mail (and
 everybody see it)
#PB_Mail_Bcc: Recipient(s) which are in copy of the mail (but
 nobody see it)
```

Example:

```
InitNetwork()

If CreateMail(0, "test@purebasic.com", "Hello")
  AddMailRecipient(0, "andre@purebasic.com", #PB_Mail_To) ;
 Andre is the main recipient
  AddMailRecipient(0, "fred@purebasic.com", #PB_Mail_Cc)  ;
 Fred is in copy
  AddMailRecipient(0, "timo@purebasic.com", #PB_Mail_Bcc) ;
 Timo is in copy as well, but Andre and Fred doesn't know it
EndIf
```

95.4 CreateMail

Syntax

```
Result = CreateMail(#Mail, From$, Subject$ [, Encoding])
```

Description

Create a new, empty #Mail. If #PB_Any is used as '#Mail' parameter, the new mail number will be returned as 'Result'. 'From$' will be the reply address to this mail. 'Subject$' will be the short text which is displayed to quickly resume the mail content. 'Encoding' is an optional parameter and can be one of the following value:

```
PB_Ascii : The mail body will be in ascii (default)
PB_UTF8  : The mail body will be in UTF-8
```

If the mail can not be correctly created, 'Result' will be 0.

SetMailBody() , SetMailAttribute() , AddMailAttachment() and AddMailAttachmentData() can be used to change the content of the #Mail.

Note: According to the RFC 2822 standard a line break in an e-mail need to be done always using the CRLF chars.

Example:

```
InitNetwork()

If CreateMail(0, "test@purebasic.com", "Hello")
  SetMailBody(0, "This is a body !" + #CRLF$ + "Second line")
  Debug "Mail created"
Else
  Debug "Can't create the mail"
EndIf
```

95.5 FreeMail

Syntax

```
FreeMail(#Mail)
```

Description

Free the specified #Mail and release its associated memory.

Note: if there is any mail remaining, they are all automatically freed when the program ends.

95.6 GetMailAttribute

Syntax

```
Result\$ = GetMailAttribute(#Mail, Attribute)
```

Description

Return the specified #Mail attribute. 'Attribute' may be one of the following values:

```
  #PB_Mail_From    : Get the 'From' attribute, set with
   CreateMail()
.
  #PB_Mail_Subject: Get the 'Subject' attribute, set with
   CreateMail()
.
  #PB_Mail_XMailer: Get the 'X-Mailer' attribute (if any)
  #PB_Mail_Date    : Get the 'Date' attribute (if any)
  #PB_Mail_Custom : Get customs fields (if any)
```

Example:

```
InitNetwork()

If CreateMail(0, "test@purebasic.com", "Hello")
  Debug GetMailAttribute(0, #PB_Mail_From) ; Will print
 "test@purebasic.com"
  Debug GetMailAttribute(0, #PB_Mail_Subject) ; Will print
 "Hello"
EndIf
```

SetMailAttribute() may be used to change the #Mail attributes.

95.7 GetMailBody

Syntax

```
Result\$ = GetMailBody(#Mail)
```

Description

Return the specified #Mail body, previously set with SetMailBody() .

Example:

```
InitNetwork()

If CreateMail(0, "test@purebasic.com", "Hello")
  SetMailBody(0, "This is the body")
  Debug GetMailBody(0) ; Will print "This is the body"
EndIf
```

95.8 IsMail

Syntax

```
Result = IsMail(#Mail)
```

Description

Tests if the given #Mail number is valid and if the mail has been correctly initialized. This function is bullet-proof and may be used with any value. If the 'Result' is not zero then the object is valid and has been initialized, otherwise it returns zero. This is the correct way to ensure that the mail is ready for use.

95.9 MailProgress

Syntax

```
Result = MailProgress(#Mail)
```

Description

Return the progress of the specified #Mail transfer, started with SendMail() .

'Result' is the actual number of bytes which have been sent or one of the following values:

```
#PB_Mail_Connected: The mail transfer is in its
 initialization phase.
#PB_Mail_Finished : The mail transfer is finished correctly.
#PB_Mail_Error    : The mail transfer is finished but an
 error occured.
```

95.10 RemoveMailRecipient

Syntax

```
RemoveMailRecipient(#Mail, Address$, Flags)
```

Description

Remove a recipient from the specified #Mail. 'Address$' must be in a valid address format or the mail will not be sent correctly. 'Flags' may be a combination the following values:

```
#PB_Mail_To : Main(s) recipient(s) of the mail
#PB_Mail_Cc : Recipient(s) which are in copy of the mail (and
 everybody see it)
#PB_Mail_Bcc: Recipient(s) which are in copy of the mail (but
 nobody see it)
```

Example:

```
1   InitNetwork()
2
3   If CreateMail(0, "test@purebasic.com", "Hello")
4     AddMailRecipient(0, "andre@purebasic.com", #PB_Mail_To) ;
     Andre is the main recipient
5     AddMailRecipient(0, "fred@purebasic.com", #PB_Mail_Cc)  ;
     Fred is in copy
6     AddMailRecipient(0, "timo@purebasic.com", #PB_Mail_Bcc) ;
     Timo is in copy as well, but Andre and Fred are not aware of
     it
7
8     ; Ensure Fred is removed from every destination :-)
9     RemoveMailRecipient(0, "fred@purebasic.com", #PB_Mail_To |
     #PB_Mail_Cc | #PB_Mail_Bcc)
10  EndIf
```

95.11 SendMail

Syntax

```
Result = SendMail(#Mail, Smtp$ [, Port [, Asynchronous]])
```

Description

Send the specified #Mail. 'Smtp$' is the address of the mail server which will be used to send the mail. 'Port' is an optional parameter used to specify the server port (default is 25). By default, the program flow is blocked until the mail is completely sent. If 'Asynchronous' is set to 1, then the mail is sent in the background and then MailProgress() may be used to provide additional information regarding the #Mail status. If the mail could not be sent, then 'Result' is 0.

Example:

```
1   InitNetwork()
2
3   ; Note: change the address and smtp to have a working example
4   ;
```

```
If CreateMail(0, "test@youraddress.com", "Hello")
  AddMailRecipient(0, "youraddess@youraddress.com", #PB_Mail_To)

  Debug SendMail(0, "smtp.yourfavoritesmtp.com")
EndIf
```

95.12 SetMailAttribute

Syntax

```
SetMailAttribute(#Mail, Attribute, Value$)
```

Description

Change the specified #Mail attribute with the new 'Value$'. 'Attribute' may be one of the following values:

```
  #PB_Mail_From    : Change the 'From' attribute, set with CreateMail()
.
  #PB_Mail_Subject: Change the 'Subject' attribute, set with CreateMail()
.
  #PB_Mail_XMailer: Change the 'X-Mailer' attribute (not sent by default).
  #PB_Mail_Date    : Change the 'Date' attribute (default is the computer date).
  #PB_Mail_Custom : Add customs fields (can be multi-line).
```

Example:

```
InitNetwork()

If CreateMail(0, "test@purebasic.com", "Hello")
  SetMailAttribute(0, #PB_Mail_XMailer, "PureMailer")
  Debug GetMailAttribute(0, #PB_Mail_XMailer) ; Will print "PureMailer"
EndIf
```

95.13 SetMailBody

Syntax

```
SetMailBody(#Mail, Body$)
```

Description

Change the #Mail body. GetMailBody() can be used to get it back.

Example:

```
InitNetwork()

If CreateMail(0, "test@purebasic.com", "Hello")
  SetMailBody(0, "This is the body")
  Debug GetMailBody(0) ; Will print "This is the body"
EndIf
```

Chapter 96

Map

Maps (also known as hashtable or dictionnary) are structures for storing data which are dynamically allocated depending of your need. It is a collection of elements (the data you want to store) and each element is fully independant of the others. You can add as many elements as you want (or as many as will fit into the memory of your computer), and accessing it back using a key. This kind of data management is very useful when you need fast access to a random element. The inserting order of the elements are not kept when using a map (unlike a LinkedList) and therefore they can't be sorted.

Before you can work with Maps, you must declare them first. This could be done with the keyword NewMap . structures are also often used to store multiple data in a single element.

To specifically search the contents of a Map, using of loops is recommended: For : Next , ForEach : Next , Repeat : Until or While : Wend .

96.1 AddMapElement

Syntax

```
Result = AddMapElement(Map(), Key$ [, Flags])
```

Description

Adds a new empty element in the Map() using the specified 'Key$'. This new element becomes the current element of the map. If Result is 0, then the element has not been created (out of memory), else it returns the pointer to the element data (for advanced use only).

'Flags' is an optional parameter which can be one of the following value:

```
  #PB_Map_ElementCheck  : Checks if an element with a same key
   already exists, and replaces it (default).
  #PB_Map_NoElementCheck: No element check, so if a previous
   element with the same key was already present, it
                      will be not replaced but kept in the map
   without be reachable with direct access. It will remain
   unreachable
                      until the newly added element has been
   deleted. Such unreachable elements will still be listed when
   enumerating
                      all the map elements with ForEach
 or NextMapElement()
. This mode is faster but also more
                      error prone, so use it with caution.
```

Note: This function isn't mandatory when dealing with maps, as elements are automatically added when affecting a value to them.

Example:

```
NewMap Country.s()

; Regular way to add an element
Country("US") = "United State"

; The same using AddMapElement()
AddMapElement(Country(), "FR")
Country() = "France"

ForEach Country()
  Debug Country()
Next
```

96.2 ClearMap

Syntax

```
ClearMap(Map())
```

Description

Clears all the elements in the specified 'Map()' and releases their memory. After this call the map is still usable, but is empty (i.e. there are no more elements in it).

Example:

```
NewMap Country.s()

Country("FR") = "France"
Country("US") = "United States"

; Proof that items have been added to the map
MessageRequester("Information", "There are
 "+Str(MapSize(Country()))+" elements in the map")

; Clear the map and show that the map really is empty
ClearMap(Country())
MessageRequester("Information", "There are
 "+Str(MapSize(Country()))+" element in the map")
```

Note: PureBasic will only free the memory for the elements. If you have been using the map for something such as storing handles of objects that you create directly with the OS, there is no way PureBasic (or any other language) can know what they are. Therefore, in cases such as that, you should go through the elements in the map and free the objects yourself.

96.3 CopyMap

Syntax

```
Result = CopyMap(SourceMap(), DestinationMap())
```

Description

Copy every elements of 'SourceMap()' into 'DestinationMap()'. If the two maps are not of the same type (native or structured) the copy will fail and 'Result' will be 0. Every elements previously found in 'DestinationMap()' will be deleted. After a successful copy, the two maps are identicals.

Example:

```
NewMap Age()
NewMap AgeCopy()

Age("John")  = 15
Age("Elise") = 30

CopyMap(Age(), AgeCopy())

Debug AgeCopy("John")
Debug AgeCopy("Elise")
```

96.4 FreeMap

Syntax

```
FreeMap(Map())
```

Description

Free the specified 'Map()' and release all its associated memory. To use it again NewMap has to be called.

96.5 MapSize

Syntax

```
Result = MapSize(Map())
```

Description

Returns the number of elements in the specified 'Map()'. It does not change the current element. This function is very fast (it doesn't iterates all the map but uses a cached result) and can be safely used to determine if a map is empty or not.

Example:

```
NewMap Country.s()

Country("FR") = "France"
Country("US") = "United States"

; Will print '2'
Debug "Size of the map: " + Str(MapSize(Country()))
```

96.6 DeleteMapElement

Syntax

```
DeleteMapElement(Map() [, Key$])
```

Description

Removes the current element from the specified 'Map()'. After this call, the new current element is the previous element (the one before the deleted element), which is an arbitrary element, as a map isn't sorted. If that element does not exist (in other words, you deleted the first element in the map) then there is no more current element, as it will be before the first element, like after a ResetMap() . If there was only one element in the map when you deleted it then you are left with no current element!

If the optional 'Key$' parameter is specified, the element associated to this key will be deleted from the map (if it exists). There will be no more current element after this call.

Example:

```
NewMap Country.s()

Country("US") = "United States"
Country("FR") = "France"
Country("GE") = "Germany"

; Delete a country
DeleteMapElement(Country(), "FR")

ForEach Country()
  Debug Country()
Next
```

96.7 FindMapElement

Syntax

```
Result = FindMapElement(Map(), Key$)
```

Description

Change the current map element to the element associated at the specified 'Key$'. If 'Result' is 0, there is no such 'Key$' in the map, else it returns the pointer to the element data (for advanced use only).

Example:

```
NewMap Country.s()

Country("US") = "United States"
Country("FR") = "France"
Country("GE") = "Germany"

If FindMapElement(Country(), "US")
  Debug "'US' is in the country list."
Else
  Debug "'US' is NOT in the country list !"
EndIf

If FindMapElement(Country(), "UK")
```

```
    Debug "'UK' is in the country list."
  Else
    Debug "'UK' is NOT in the country list !"
  EndIf
```

96.8 MapKey

Syntax

```
Key\$ = MapKey(Map())
```

Description

Returns the key of the current map element. If there is no current element, an empty string is returned.

Example:

```
  NewMap Country.s()

  Country("US") = "United States"
  Country("FR") = "France"
  Country("GE") = "Germany"

  ForEach Country()
    Debug MapKey(Country())
  Next
```

96.9 NextMapElement

Syntax

```
Result = NextMapElement(Map())
```

Description

Moves from the current element to the next element in the specified 'Map()', or onto the first element if ResetMap() was previously called. If Result is 0, there is no next element in the map, else it returns the pointer to the element data (for advanced use only).

Example:

```
  NewMap Country.s()

  Country("US") = "United States"
  Country("FR") = "France"
  Country("GE") = "Germany"

  ResetMap(Country())
  While NextMapElement(Country())
    Debug Country()
  Wend
```

96.10 ResetMap

Syntax

```
ResetMap(Map())
```

Description

Resets the current element of the specified 'Map()' to be before the first element. This means than no current element. However, this is very useful to process all the elements by using NextMapElement() .

Example:

```
NewMap Country.s()

Country("US") = "United States"
Country("FR") = "France"
Country("GE") = "Germany"

ResetMap(Country())
While NextMapElement(Country())
  Debug Country()
Wend
```

Chapter 97

Material

Materials are composed of one or several textures and sometimes of some colors. They are widely used by the other objects of the 3D world like the entities , billboards and particles to give them a skin.

Each material has a lot a properties like the shading, ambient and specular color, etc. to allow realistic looking material like wood, water, glass and more.

InitEngine3D() should be called successfully before using the Material functions.

97.1 AddMaterialLayer

Syntax

```
Result = AddMaterialLayer(#Material, TextureID [, Mode])
```

Description

Adds a new layer to the #Material and put the specified texture in it. A valid 'TextureID' can be easily obtained with the TextureID() function.

'Mode' is an optional parameter and can take one of the following value:

```
#PB_Material_Add        : Performs a pîxel 'Add' operation
 over previous layer (black color is like transparent)
#PB_Material_Replace    : Performs a pîxel 'Replace'
 operation over previous layer
#PB_Material_AlphaBlend : Use the alphachannel layer of the
 texture (should be a TGA or PNG one) and blend it with the
 previous layer
#PB_Material_Modulate   : Performs a pîxel 'Modulate'
 operation over previous layer
```

97.2 CopyMaterial

Syntax

```
Result = CopyMaterial(#Material, #NewMaterial)
```

Description

Creates a #NewMaterial which is the exact copy of the specified #Material. If #PB_Any is used as '#NewMaterial' parameter, the new material number will be returned as 'Result'.

If the 'Result' is 0, the Material copy has failed. If the #NewMaterial was already created, it is automatically freed and replaced by the new one.

97.3 CountMaterialLayers

Syntax

```
Result = CountMaterialLayers(#Material)
```

Description

Returns the number of layers the #Material contains.

97.4 CreateMaterial

Syntax

```
Result = CreateMaterial(#Material, TextureID)
```

Description

Creates a new #Material based on the specified texture. If #PB_Any is used as '#Material' parameter, the new material number will be returned as 'Result'. A valid 'TextureID' can be easily obtained with the TextureID() function.

If the 'Result' is 0, the Material creation has failed. If the #Material was already created, it is automatically freed and replaced by the new one.

97.5 DisableMaterialLighting

Syntax

```
DisableMaterialLighting(#Material, State)
```

Description

Enables or disables the dynamic #Material lighting. This means than the object which will use this material will be not affected by a dynamic light, created with the CreateLight() function. Dynamic lighting is enabled by default, when a material is created. 'State' can take the following values:

```
1: Dynamic lighting is disabled
0: Dynamic lighting is enabled
```

97.6 FreeMaterial

Syntax

```
FreeMaterial(#Material)
```

Description

Frees the specified #Material. All its associated memory is released and this object can't be used anymore.

Note: all remaining materials are automatically freed when the program ends.

97.7 IsMaterial

Syntax

```
Result = IsMaterial(#Material)
```

Description

Tests if the given #Material is a valid and correctly initialized material.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that a material is ready to use.

97.8 MaterialAmbientColor

Syntax

```
MaterialAmbientColor(#Material, Color)
```

Description

Changes the #Material ambient color. The ambient material color is the color the material gets by default, without other lighting.

A valid color can be easily obtained with the RGB() function. If 'Color' is set to #PB_Material_AmbientColors, then the color informations associated to each vertex (as specified in the mesh creation) are used.

97.9 MaterialBlendingMode

Syntax

```
MaterialBlendingMode(#Material, Mode)
```

Description

Changes the way the #Material will be blended with the scene (screen background). 'Mode' can take one of the following value:

```
#PB_Material_Add        : Performs a pîxel 'Add' operation
 over the scene (black color is like transparent).
#PB_Material_AlphaBlend: Uses the alphachannel layer of the
 texture (should be a TGA or PNG one) to blend it with the
 scene.
#PB_Material_Color      : Uses the texture transparent color
 value when blending the material with the scene.
```

97.10 MaterialDiffuseColor

Syntax

```
MaterialDiffuseColor(#Material, Color)
```

Description

Changes the #Material diffuse color. The diffuse material color is the color than the material will reflect when it will be dynamically lighted. For example, using a full white color (RGB(255,255,255)) will result at normal lighting (all colors are reflected). If you use a red (RGB(255,0,0)), then only the red colors of the material will be reflected, resulting as a full red material (or black one, if the material doesn't contain any red color). A valid color value can be easily obtained with the RGB() function.

97.11 MaterialFilteringMode

Syntax

```
MaterialFilteringMode(#Material, Mode)
```

Description

Changes the #Material filtering mode. 'Mode' can take one of the following value:

```
#PB_Material_None     : Don't filter the material which
 becomes very pixelized when the camera gets close.
#PB_Material_Bilinear : Performs a bilinear filtering when
 the camera gets close, resulting to a smooth, a bit blured
 picture.
#PB_Material_Trilinear: Performs a trilinear filtering when
 the camera gets close, resulting in the best picture quality
 possible.
```

When a material is created, the bilinear filtering is used by default. Using a filter doesn't have a big performance impact on the rendering, as many graphics cards do it using the hardware.

97.12 MaterialID

Syntax

```
MaterialID = MaterialID(#Material)
```

Description

Returns the MaterialID of the specified #Material. This is the unique Material identifier needed by other functions: EntityMaterial() .

97.13 MaterialShadingMode

Syntax

```
MaterialShadingMode(#Material, Mode)
```

Description

Changes the #Material shading mode. 'Mode' can take one of the following value:

```
#PB_Material_Flat   : The material use the flat mode, the
 lighting is done face by face.
#PB_Material_Gouraud: Performs a shading using the Gouraud
 algorithme.
#PB_Material_Phong  : Performs a shading using the Phong
 algorithme.
```

When a material is created, the Gouraud shading is used by default.

97.14 MaterialSpecularColor

Syntax

```
MaterialSpecularColor(#Material, Color)
```

Description

Changes the #Material specular color. The specular material color is the color than the material will reflect when it will be dynamically lighted by a light which has a specular color value. For example, using a full white color (RGB(255,255,255)) will result at normal lighting (all colors are reflected). If you use a red (RGB(255,0,0)), then only the red colors of the material will be reflected, resulting as a full red material (or black one, if the material doesn't contain any red color). A valid color value can be easily obtained with the RGB() function.

97.15 ScrollMaterial

Syntax

```
ScrollMaterial(#Material, x, y, Mode [, Layer])
```

Description

Scrolls the material layer according to x,y values. By default the layer 0 is scrolled except if the optional 'Layer' parameter is used. 'Mode' can take one of the following value:

```
#PB_Material_Fixed    : The material is scrolled of the
 specified x,y offset, without regards of the previous scroll
 function.
#PB_Material_Animated: Each frame is the material is scrolled
 of the specified x,y offset automatically.
```

97.16 RemoveMaterialLayer

Syntax

```
RemoveMaterialLayer(#Material)
```

Description

Removed the top most (last added) #Material layer.

97.17 RotateMaterial

Syntax

```
RotateMaterial(#Material, Angle, Mode [, Layer])
```

Description

Rotates the material layer according to the Angle value. By default the layer 0 is rotated except if the optional 'Layer' parameter is used. 'Mode' can take one of the following value:

```
#PB_Material_Fixed    : The material is rotated of the
 specified 'Angle', without regards of the previous rotate
 function.
#PB_Material_Animated: Each frame is the material is rotated
 of the specified 'Angle' automatically.
```

Chapter 98

Math

Math library provides basic mathematical functions such as: Cos(), Sin(), Pow(), Log() etc... Almost all these functions work with floats or doubles, i.e. numbers with the .f or .d extensions. If a function is used with a double value as input or output, it will automatically switch to double precision instead of single precision for its calculations.

98.1 Abs

Syntax

```
Result.f = Abs(Number.f)
```

Description

Returns the absolute value of the given float value. The return value is always positive. This function will only work correctly with float numbers. With integers, it will fail if the integer is too large (due to a loss of precision).

Example:

```

Debug Abs(3.14159)  ; Will display '3.14159'
Debug Abs(-3.14159) ; will display '3.14159'
```

98.2 ACos

Syntax

```
Result.f = ACos(Value.f)
```

Description

Returns the arc-cosine of the specified value. Value.f must be between -1 and 1 (-1 and 1 included), the result will be in radians.

Example:

```

Debug ACos(1)   ; will display '0'
Debug Acos(-1) ; will display 'pi' (approximately 3.14159)
```

This is the inverse function of Cos() .

98.3 ACosH

Syntax

```
Result.f = ACosH(Value.f)
```

Description

Returns the area hyperbolic cosine of the specified value. Value.f must be between greater than or equal to 1. The result is a hyperbolic angle.

Example:

```

Debug ACosH(1)                  ; will display '0'
Debug Exp(ACosH(0.5 * Sqr(5))) ; will display '1.618033' (the
 golden ratio)
```

This is the inverse function of CosH() .

98.4 ASin

Syntax

```
Result.f = ASin(Value.f)
```

Description

Returns the arc-sine of the specified value. Value.f must be between -1 and 1 (-1 and 1 included), the result will be in radians.

Example:

```

Debug ASin(1) ; will display '1.570796' (pi/2)
Debug ASin(0) ; will display '0'
```

This is the inverse function of Sin() .

98.5 ASinH

Syntax

```
Result.f = ASinH(Value.f)
```

Description

Returns the area hyperbolic sine of the specified value. The range of Value.f is not limited. The result is a hyperbolic angle.

Example:

```

  Debug ASinH(0)         ; will display '0'
  Debug Exp(ASinH(0.5)) ; will display '1.618033' (the golden
    ratio)

Debug
Debug
```

This is the inverse function of SinH() .

98.6 ATan

Syntax

```
Result.f = ATan(Value.f)
```

Description

Returns the arc-tangent of the specified value. Value.f can be any value, the result will be in radians.

Example:

```
  Debug ATan(1) ; will display '0.785398' (pi/4)
```

This is the inverse function of Tan() .

98.7 ATan2

Syntax

```
Result.f = ATan2(x.f, y.f)
```

Description

Calculates the angle in radian between the x axis and a line drawn in the direction specified by 'x' and 'y'. It can be used to calculate angles between lines in 2D or to transform rectangular coordinates into polar coordinates.

This function calculates the value ATan(y/x) and examines the sign of x and y to place the angle in the correct quadrant. It also handles the cases where y is zero to avoid division by zero errors.

The result is always between -#PI and +#PI. Negative angles indicate that the line is below the x axis, positive values indicate that line is above the x axis. If 'x' and 'y' are zero then the function retuns 0.

Example:

```
  Debug ATan2(10, 10) ; will display #PI/4 (45 degrees in
    radian)
```

98.8 ATanH

Syntax

```
Result.f = ATanH(Value.f)
```

Description

Returns the area hyperbolic tangent of the specified value. Value.f must be between -1 and 1 (not including -1 and 1). The result is a hyperbolic angle.

Example:

```
Debug Exp(ATanH(0.2 * Sqr(5))) ; will display '1.618033' (the golden ratio)
```

This is the inverse function of TanH() .

98.9 Cos

Syntax

```
Result.f = Cos(Angle.f)
```

Description

Returns the cosine of the specified angle. Angle.f must be in radians, and the result will be between -1 and 1.

Example:

```
Debug Cos(3.141593) ; will display '-1'
```

The inverse function of Cos() is ACos() .

98.10 CosH

Syntax

```
Result.f = CosH(Angle.f)
```

Description

Returns the hyperbolic cosine of the specified hyperbolic angle. The result will be greater than or equal to 1.

Example:

```
Debug CosH(0)  ; will display 1
```

The inverse function of CosH() is ACosH() .

98.11 Degree

Syntax

```
Result.f = Degree(Angle.f)
```

Description

Converts the given angle from radian to degree. The reverse transformation can be made with the Radian() function.

Example:

```
Debug Degree(#PI/4) ; will display 45
```

98.12 Exp

Syntax

```
Result.f = Exp(Number.f)
```

Description

Returns the result of the exponential function. This is the value e raised to the power 'Number'. This is the inverse function of Log() .

98.13 Infinity

Syntax

```
Result = Infinity()
```

Description

Returns the special floating point value representing positive infinity. Negative infinity can be calculated using "-Infinity()".

Infinity and negative infinity are special values. They behave in calculations in the way you would generally expect. For example dividing infinity by any positive number (except 0 or infinity) will result in infinity again. The IsInfinity() function can be used to check if a value represents positive or negative infinity.

Example:

```
Debug IsInfinity(Infinity() / 1000) ; return 1
```

98.14 Int

Syntax

```
Result = Int(Number.f)
```

Description

Returns the integer part of a float number.

Example:

```
Result = Int(10.565)  ; will display '10'
```

98.15 IntQ

Syntax

```
Result = IntQ(Number.d)
```

Description

Returns the integer part of a double number.

Example:

```
Result.q = IntQ(12345678901.565)  ; will display '12345678901'
```

98.16 IsInfinity

Syntax

```
Result.f = IsInfinity(Value.f)
```

Description

Returns nonzero if the given value represents positive or negative infinity.

Checking for the infinity values should not be done using normal comparison, because it depends on the hardware implementation wether infinity is considered equal to itself or not. The value of positive infinity can be generated with the Infinity() function.

Example:

```
Result = IsInfinity(Infinity())  ; infinity
Result = IsInfinity(Log(0))      ; -infinity
Result = IsInfinity(1234.5)      ; a finite number
Result = IsInfinity(NaN())       ; NaN is not the same as
  infinity
```

98.17 IsNaN

Syntax

```
Result.f = IsNaN(Value.f)
```

Description

Returns nonzero if the given value 'Not a Number'. This value is the result of some invalid calculations. It can also be generated using the NaN() function.

NaN is a special value. Testing for it should not be done using normal comparisons because there are actually many different values for NaN and wether or not NaN is considered equal with itself in comparisons depends on the hardware.

Example:

```
Result = IsNAN(NaN())        ; NaN
Result = IsNAN(Sqr(-1))      ; NaN
Result = IsNAN(1234.5)       ; a normal number
Result = IsNAN(Infinity())   ; infinity is not NaN
```

98.18 Pow

Syntax

```
Result.f = Pow(Number.f, Power.f)
```

Description

Returns the Number.f, raised to the Power.f.

Example:

```
Debug Pow(2.0, 3.0)  ; will display '8.0'
```

98.19 Log

Syntax

```
Result.f = Log(Number.f)
```

Description

Returns the natural Log (ie log to the base e) of the Number. This is the inverse function of Exp() .

98.20 Log10

Syntax

```
Result.f = Log10(Number.f)
```

Description

Returns the log in base 10 of the 'Number'.

98.21 NaN

Syntax

```
Result.f = NaN()
```

Description

Returns the special floating point value representing 'Not a Number'. This value is returned from invalid calculations such as calculating the square root of a negative number.

NaN is a special value. Using NaN in any calculation with other values will again return the value NaN. The IsNaN() function can be used to check wether a variable represents the value NaN.

Example:

```
Debug IsNaN(NaN() * 5 + 2) ; will display 1
```

98.22 Radian

Syntax

```
Result.f = Radian(Angle.f)
```

Description

Converts the given angle from degrees into radian. The reverse transformation can be made with the Degree() function.

Example:

```
Debug Radian(90) ; will display #PI/2
```

98.23 Round

Syntax

```
Result.f = Round(Number.f, Mode)
```

Description

Round the specified float number according to the given mode. 'Mode' can be one of the following values:

```
#PB_Round_Down    : rounds down the number
#PB_Round_Up      : rounds up the number
#PB_Round_Nearest: rounds to the nearest integer number (0.5
 and up will round up)
```

Example:

```
Debug Round(11.6, #PB_Round_Down)     ; Will print '11'
Debug Round(-3.6, #PB_Round_Down)     ; Will print '-4'

Debug Round(11.6, #PB_Round_Up)       ; Will print '12'
Debug Round(-1.6, #PB_Round_Up)       ; Will print '-1'

Debug Round(11.6, #PB_Round_Nearest) ; Will print '12'
Debug Round(11.4, #PB_Round_Nearest) ; Will print '11'
Debug Round(11.5, #PB_Round_Nearest) ; Will print '12'
Debug Round(-7.5, #PB_Round_Nearest) ; Will print '-8'
```

98.24 Sign

Syntax

```
Result.f = Sign(Number.f)
```

Description

Returns a floating point value representing the sign of the given number.

- Returns 0 if Number is zero.
- Returns 1 if Number is positive.
- Returns -1 if Number is negative.

Example:

```
Debug Sign(-7)    ; will display -1.0
Debug Sign(0)     ; will display 0.0
Debug Sign(7)     ; will display 1.0
```

98.25 Sin

Syntax

```
Result.f = Sin(Angle.f)
```

Description

Returns the sine of the specified angle. Angle.f must be in radians, and the result will be between -1 and 1.

Example:

```
Debug Sin(1.5708) ; will display '1'
```

The inverse function of Sin() is ASin() .

98.26 SinH

Syntax

```
Result.f = SinH(Angle.f)
```

Description

Returns the hyperbolic sine of the specified hyperbolic angle.

Example:

```
Debug SinH(Log(1.618033)) ; Will be approximately 0.5
```

The inverse function of SinH() is ASinH() .

98.27 Sqr

Syntax

```
Result.f = Sqr(Number.f)
```

Description

Returns the square root of the specified number.

98.28 Tan

Syntax

```
Result.f = Tan(Angle.f)
```

Description

Returns the tangent of the specified angle. 'Angle' has to be in radians.

Example:

```
Debug Tan(0.785398) ; will display '1'
```

The inverse function of Tan() is ATan() .

98.29 TanH

Syntax

```
Result.f = TanH(Angle.f)
```

Description

Returns the hyperbolic tangent of the specified hyperbolic angle.

Example:

```
Debug TanH(Log(1.618033)) ; will display '0.447213' (1/5 *
 Sqr(5))
```

The inverse function of TanH() is ATanH() .

Chapter 99

Memory

Sometimes, it's very useful have access to the system memory (RAM) to process some time consumming operations and speed them up. This library allows to allocate any number of memory buffers and to use them directly within PureBasic.

99.1 AllocateMemory

Syntax

```
*MemoryID = AllocateMemory(Size)
```

Description

Allocates a contiguous memory area according to the specified size (in bytes). If the needed memory is available, the allocated memory will be initialized with 'Null' (0) and *MemoryID will get the start address of the memory area, else it will be 0.

Note: All the allocated memory areas are automatically freed when the programs ends.

Example:

```
*MemoryID = AllocateMemory(5000)
If *MemoryID
  Debug "Starting address of the 5000 Byte memory area:"
  Debug *MemoryID
  PokeS(*MemoryID, "Store this string in the memory area")
  FreeMemory(*MemoryID)  ; will also be done automatically at the end of program
Else
  Debug "Couldn't allocate the requested memory!"
EndIf
```

99.2 CompareMemory

Syntax

```
Result = CompareMemory(*MemoryID1, *MemoryID2, Length)
```

Description

Compare two memory areas and returns 1 if its equal or 0 if it doesn't match.

The *MemoryID are results of AllocateMemory() or ReAllocateMemory() .

99.3 CompareMemoryString

Syntax

```
Result = CompareMemoryString(*String1, *String2 [, Mode [,
   Length [, Flags]]])
```

Description

For advanced programmers. Compares two string at the specified memory address. The strings should be zero terminated or the 'Length' should be specified to tell how many characters should be compared. This function can be very useful to compare string in memory buffer for performance reasons.

'Mode' can can be one of the following values:

```
  #PB_String_NoCase: String comparison is case insensitive (a =
   A). By default the comparison is case sensitive.
```

'Result' can can be one of the following values:

```
  #PB_String_Equal  : if String1 equals String2
  #PB_String_Lower  : if String1 is lower than String2
  #PB_String_Greater: if String1 greater than String2
```

The optional 'Flags' parameter can be one of the following values:

```
  #PB_Ascii  : Compares the strings as ascii, even when the
   program is compiled in unicode
 mode
  #PB_UTF8   : Compares the strings as UTF8 (the program can be
   either in unicode or ascii mode)
  #PB_Unicode: Compares the strings as unicode string, even
   when the program is compiled in ascii mode
```

Note: when the 'Flags' parameter is specified, the 'Length' can be put to -1, to compare the strings until a null character is found.

99.4 CopyMemory

Syntax

```
CopyMemory(*SourceMemoryID, *DestinationMemoryID, Length)
```

Description

Copy a memory area starting from the *SourceMemoryID (the start memory address) to the *DestinationMemoryID (address of the destination memory) with the specified length (in bytes).

If the source and destination buffers overlap, it is safer to use MoveMemory() , else the result can be wrong.

99.5 CopyMemoryString

Syntax

```
Result = CopyMemoryString(*String [, @*DestinationMemoryID])
```

Description

For advanced programmers. Copy the String at the specified address to the destination memory address if specified, or at the end of the current buffer if omitted. If *DestinationMemoryID is specified, the internal buffer pointer is resetted the new value. The internal pointer is automatically updated after a copy, so it's very handy to do a stream copy of strings, while keeping the pointer control. This is function is especially optimized to manipulate memory text buffer extremely quickly.

Example:

```
*Buffer = AllocateMemory(1000)
*Pointer = *Buffer
CopyMemoryString("Hello", @*Pointer)
CopyMemoryString(" World")  ; This one will be put just after "Hello"
*Pointer-2                  ; Come back from 2 characters (on the 'l' of 'World')
CopyMemoryString("LD")      ; Finally the last letter will be uppercase
Debug PeekS(*Buffer)
```

99.6 FillMemory

Syntax

```
FillMemory(*Memory, Size [, Value [, Type]])
```

Description

Fills the '*Memory' area with the specified 'Value'. If no value is specified, it fills the memory area with 0.

The optional 'Type' parameter can be one of the following constants:

```
#PB_Byte: Fills the memory using a byte (1 byte) value (default).
#PB_Word: Fills the memory using a word (2 bytes) value.
#PB_Long: Fills the memory using a long (4 bytes) value.
```

Example:

```
*Buffer = AllocateMemory(500)

FillMemory(*Buffer, 500) ; Fill 500 bytes of value 0 (clear the memory area)
FillMemory(*Buffer, 500, $FF) ; Fill 500 bytes of value $FF
FillMemory(*Buffer, 500, $BADF00D, #PB_Long) ; Fill 500 bytes of value $BADF00D
```

99.7 FreeMemory

Syntax

```
FreeMemory(*MemoryID)
```

Description

Free the memory previously allocated with AllocateMemory() or ReAllocateMemory() .

Note: all remaining allocated memory blocks are automatically freed when the program ends.

99.8 MemorySize

Syntax

```
Result = MemorySize(*MemoryID)
```

Description

Returns the length (in bytes) of the given '*MemoryID'.

This memory area has to be allocated either with AllocateMemory() or ReAllocateMemory ().

99.9 MemoryStringLength

Syntax

```
Result = MemoryStringLength(*String [, Flags])
```

Description

For advanced programmers. Returns the length (in characters) of the given zero terminated string. If the 'Flags' parameter isn't specified, the string is reads as unicode if the program is compiled in unicode mode, else it's reads as ascii.

The optional 'Flags' parameter can be one of the following values:

```
#PB_Ascii   : Returns the length of an ascii string, even when
 the program is compiled in unicode mode
#PB_UTF8    : Returns the length of an UTF8 string (the
 program can be either in unicode or ascii mode)
#PB_Unicode: Returns the length of an unicode string, even
 when the program is compiled in ascii mode
```

99.10 MoveMemory

Syntax

```
MoveMemory(*SourceMemoryID, *DestinationMemoryID, Length)
```

Description

Copy a memory area starting from the *SourceMemoryID (the start memory address) to the *DestinationMemoryID (address of the destination memory) with the specified length (in bytes).

Unlike CopyMemory() , the source and destinations memory area can overlap. MoveMemory() is slower than CopyMemory() , so it's better to use it only when necessary.

99.11 ReAllocateMemory

Syntax

```
*MemoryID = ReAllocateMemory(*MemoryID, Size)
```

Description

Allocates (when parameter *MemoryID is 0) a new or re-allocates (with a valid *MemoryID as parameter) an existing contiguous memory area according to the specified size (in bytes). If the needed memory is available, the return value *MemoryID will get the start address of the memory area, else it will be 0 (the existing memory area will remain then, as well its previous *MemoryID stay valid).

The content of the previous memory area '*MemoryID' is kept. If the new 'size' of the memory area is smaller than the previous one, the end is cutted. If the new 'size' is larger than the previous one, the memory area will be filled up with Null.

Note: All the allocated memory areas are automatically freed when the programs ends.

Example:

```
*MemoryID = AllocateMemory(1000)
PokeS(*MemoryID, "Store this string")
; do something more with it here...
;
*NewMemoryID = ReAllocateMemory(*MemoryID, 2000) ; need more memory
If *NewMemoryID
  ; work with *NewMemoryID now with size 2000
  Debug "The old contents are still here:"
  Debug PeekS(*NewMemoryID)
  ;
  FreeMemory(*NewMemoryID)
Else
  ; resizing failed, keep working with *MemoryID (size 1000)
  ;
  FreeMemory(*MemoryID)
EndIf
```

99.12 PeekA

Syntax

```
Value.a = PeekA(*MemoryBuffer)
```

Description

For advanced programmers. PeekA() reads an ascii character (1 byte) from the specified memory address.

99.13 PeekB

Syntax

```
Value.b = PeekB(*MemoryBuffer)
```

Description

For advanced programmers. PeekB() reads a byte (1 byte) number from the specified memory address.

99.14 PeekC

Syntax

```
Value.c = PeekC(*MemoryBuffer)
```

Description

For advanced programmers. PeekC() reads a character (1 byte in ascii, 2 bytes in unicode) number from the specified memory address.

99.15 PeekD

Syntax

```
Value.d = PeekD(*MemoryBuffer)
```

Description

For advanced programmers. PeekD() reads a double (8 bytes) number from the specified memory address.

99.16 PeekI

Syntax

```
Value.i = PeekI(*MemoryBuffer)
```

Description

For advanced programmers. PeekI() reads an integer (4 bytes on 32 bits processors, 8 bytes on 64 bits processors) number from the specified memory address.

99.17 PeekL

Syntax

```
Value.l = PeekL(*MemoryBuffer)
```

Description

For advanced programmers. PeekL() reads a long (4 bytes) number from the specified memory address.

99.18 PeekW

Syntax

```
Value.w = PeekW(*MemoryBuffer)
```

Description

For advanced programmers. PeekW() reads a word (2 bytes) number from the specified memory address.

99.19 PeekF

Syntax

```
Value.f = PeekF(*MemoryBuffer)
```

Description

For advanced programmers. PeekF() reads a float (4 bytes) from the specified memory address.

99.20 PeekQ

Syntax

```
Value.q = PeekQ(*MemoryBuffer)
```

Description

For advanced programmers. PeekQ() reads a quad (8 bytes) number from the specified memory address.

99.21 PeekS

Syntax

```
Text\$ = PeekS(*MemoryBuffer [, Length [, Flags]])
```

Description

For advanced programmers. Very useful to read a string at the specified memory address. The string should be ended by a '0' character, else it will read the memory until a '0' character is encounter.

An optional 'Length' parameter (in characters) can be specified. If the 'Flags' parameter isn't specified, the string is reads as unicode if the program is compiled in unicode mode, else it's reads as ascii.

The optional 'Flags' parameter can be one of the following values:

```
#PB_Ascii   : Reads the string as ascii, even when the program
 is compiled in unicode mode
#PB_UTF8    : Reads the string as UTF8 (the program can be
 either in unicode or ascii mode)
#PB_Unicode: Reads the string as unicode, even when the
 program is compiled in ascii mode
```

When the 'Flags' parameter is specified, the 'Length' can be put to -1, to read the string until a null character is found.

Note: a combination of PokeS() and PeekS() can be used to perform conversion between the different supported string modes (Ascii, UTF8 and Unicode).

99.22 PeekU

Syntax

```
Value.u = PeekU(*MemoryBuffer)
```

Description

For advanced programmers. PeekU() reads an unicode character (2 bytes) from the specified memory address.

99.23 PokeA

Syntax

```
PokeA(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeA() writes an ascii character (1 byte) to the specified memory address.

99.24 PokeB

Syntax

```
PokeB(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeB() writes a byte (1 byte) number to the specified memory address.

99.25 PokeC

Syntax

```
PokeC(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeC() writes a character (1 byte in ascii, 2 bytes in unicode) number to the specified memory address.

99.26 PokeD

Syntax

```
PokeD(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeD() writes a double (8 bytes) number to the specified memory address.

99.27 PokeI

Syntax

```
PokeI(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeI() writes an integer (4 bytes on 32 bits processors, 8 bytes on 64 bits processors) number to the specified memory address.

99.28 PokeL

Syntax

```
PokeL(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeL() writes a long (4 bytes) number to the specified memory address.

99.29 PokeQ

Syntax

```
PokeQ(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeQ() writes a quad (8 bytes) number to the specified memory address.

99.30 PokeW

Syntax

```
PokeW(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeW() a word (2 bytes) number to the specified memory address.

99.31 PokeF

Syntax

```
PokeF(*MemoryBuffer, Number.f)
```

Description

For advanced programmers. PokeF() writes a float (4 bytes) to the specified memory address.

99.32 PokeS

Syntax

```
PokeS(*MemoryBuffer, Text$ [, Length [, Flags]])
```

Description

For advanced programmers. Writes a string (including the ending '0') to the specified memory address.

An optional length parameter (in characters) can be specified that defines the length of the string itself, a zero will still be placed in memory after the poked string (ie. specifying 6 will write 7 bytes, 6 for the string, and the last byte will be zero).

If the 'Flags' parameter isn't specified, the string is written as unicode if the program is compiled in unicode mode, else it is written as ascii.

The optional 'Flags' parameter can be one of the following values:

```
#PB_Ascii   : Writes the string as ascii, even when the
 program is compiled in unicode mode
#PB_UTF8    : Writes the string as UTF8 (the program can be
 either in unicode or ascii mode)
#PB_Unicode: Writes the string as unicode, even when the
 program is compiled in ascii mode
```

When the 'Flags' parameter is specified, the 'Length' can be put to -1, PokeS() will then evaluate the needed string length automatically.

Note: a combination of PokeS() and PeekS() can be used to perform conversion between the different supported string modes (Ascii, UTF8 and Unicode).

99.33 PokeU

Syntax

```
PokeU(*MemoryBuffer, Number)
```

Description

For advanced programmers. PokeU() writes an unicode character (2 bytes) to the specified memory address.

Chapter 100

Menu

Creating and managing menus in PureBasic is simple. You can, of course, tailor the menus to your specific needs.

To use a menu you must first start by creating one with either CreateMenu() for normal menus, or CreatePopupMenu() for pop-up menus.

Mac OSX:
There are predefined menu events #PB_Menu_Quit, #PB_Menu_About and #PB_Menu_Preferences to represent the entries in the application menu which is present on every OSX program. Their values are negative to not conflict with any menu entries defined in the program. They are reported from EventMenu() as regular menu events.

100.1 CloseSubMenu

Syntax

```
CloseSubMenu()
```

Description

Close the current sub menu and return to the previous one. OpenSubMenu() must alread have been called!

For an example and a preview image see OpenSubMenu() .

100.2 CreateMenu

Syntax

```
Result = CreateMenu(#Menu, WindowID)
```

Description

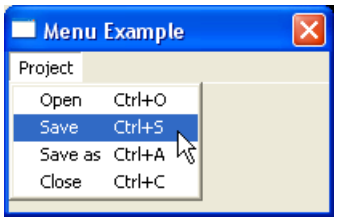
Creates a new empty menu on the given window. If #PB_Any is used as the '#Menu' parameter, the new menu number will be returned as 'Result' - if the menu can't be created 'Result' will be 0. WindowID can be obtained with the WindowID() function from the Window library. Once created, this menu becomes the current menu for further item additions. It's now possible to use functions such as MenuTitle() , MenuItem() , MenuBar() , OpenSubMenu() etc...

To handle menu events properly, see the description of following functions: WaitWindowEvent() (alternatively WindowEvent())

EventWindow()
EventMenu()

Example:

```
If OpenWindow(0, 200, 200, 200, 100, "Menu Example")
  If CreateMenu(0, WindowID(0))     ; menu creation starts....
    MenuTitle("Project")
      MenuItem(1, "Open"    +Chr(9)+"Ctrl+O")
      MenuItem(2, "Save"    +Chr(9)+"Ctrl+S")
      MenuItem(3, "Save as"+Chr(9)+"Ctrl+A")
      MenuItem(4, "Close"   +Chr(9)+"Ctrl+C")
  EndIf
  Repeat : Until WaitWindowEvent()=#PB_Event_CloseWindow
EndIf
```



100.3 CreateImageMenu

Syntax

```
Result = CreateImageMenu(#Menu, WindowID [, Flags])
```

Description

Creates a new empty menu on the given window, with image support for its items. If #PB_Any is used as the '#Menu' parameter, the new menu number will be returned as 'Result' - if the menu can't be created 'Result' will be 0. WindowID can be obtained with the WindowID() function from the Window library. Once created, this menu becomes the current menu for further item additions. It's now possible to use functions such as MenuTitle() , MenuItem() , MenuBar() , OpenSubMenu() etc.

'Flags' can be a combination of the following values:

```
#PB_Menu_ModernLook: Enable gradient and modern look
```

To handle menu events properly, see the description of following functions:
WaitWindowEvent() (alternatively WindowEvent())
EventWindow()
EventMenu()

Example:

```
If LoadImage(0, OpenFileRequester("Choose an icon file", "", "", 0))
  If OpenWindow(0, 200, 200, 200, 100, "Image menu Example")
    If CreateImageMenu(0, WindowID(0))     ; menu creation starts....
      MenuTitle("Project")
        MenuItem(1, "Open"    +Chr(9)+"Ctrl+O", ImageID(0))
        MenuItem(2, "Save"    +Chr(9)+"Ctrl+S")
        MenuItem(3, "Save as"+Chr(9)+"Ctrl+A")
        MenuItem(4, "Close"   +Chr(9)+"Ctrl+C")
    EndIf
```

```

      Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
    EndIf
  EndIf
```

100.4 CreatePopupMenu

Syntax

```
Result = CreatePopupMenu(#Menu)
```

Description

Creates a new empty popup menu. If #PB_Any is used as '#Menu' parameter, the new menu number will be returned as 'Result'. If the menu can't be created then 'Result' will equal 0. When created this menu becomes active, further menu additions will occur on it. It's now possible to use functions such as MenuItem() , MenuBar() , OpenSubMenu() etc...

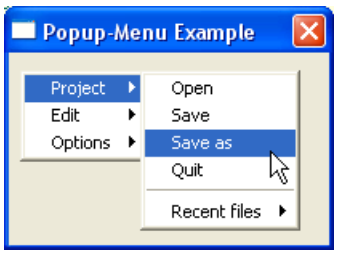
DisplayPopupMenu() can be used to display this popup menu at any position on the screen.

To handle menu events properly, see the description of following functions:
WaitWindowEvent() (alternatively WindowEvent())
EventWindow()
EventMenu()

Example:

```
  If OpenWindow(0, 200, 200, 200, 120, "Popup-Menu Example")
    If CreatePopupMenu(0)       ; creation of the pop-up menu begins...
      MenuItem(1, "Open")       ; You can use all commands for creating a menu
      MenuItem(2, "Save")       ; just like in a normal menu...
      MenuItem(3, "Save as")
      MenuItem(4, "Quit")
      MenuBar()
      OpenSubMenu("Recent files")
        MenuItem(5, "PureBasic.exe")
        MenuItem(6, "Test.txt")
      CloseSubMenu()
    EndIf
    Repeat
      Select WaitWindowEvent()      ; check for window events
        Case #WM_RButtonDown        ; right mouse button was clicked =>
          DisplayPopupMenu(0, WindowID(0))  ; now display the popup-menu
        Case #PB_Event_Menu         ; an item of the popup-menu was clicked
          Select EventMenu()      ; get the clicked menu item...
            Case 1 : Debug "Menu: Open"
            Case 2 : Debug "Menu: Save"
            Case 3 : Debug "Menu: Save as"
            Case 4 : Quit = 1
            Case 5 : Debug "Menu: PureBasic.exe"
            Case 6 : Debug "Menu: Text.txt"
          EndSelect
        Case #PB_Event_CloseWindow
          Quit = 1
      EndSelect
```

```
    Until Quit = 1
  EndIf
```



100.5 CreatePopupImageMenu

Syntax

```
Result = CreatePopupImageMenu(#Menu [, Flags])
```

Description

Creates a new empty popup menu, with image support for its items. If #PB_Any is used as '#Menu' parameter, the new menu number will be returned as 'Result'. If the menu can't be created then 'Result' will equal 0. When created this menu becomes active, further menu additions will occur on it. It's now possible to use functions such as MenuItem() , MenuBar() , OpenSubMenu() etc...

'Flags' can be a combination of the following values:

```
#PB_Menu_ModernLook: Enable gradient and modern look
```

DisplayPopupMenu() can be used to display this popup menu at any position on the screen.

To handle menu events properly, see the description of following functions:
WaitWindowEvent() (alternatively WindowEvent())
EventWindow()
EventMenu()

Example:

```
If OpenWindow(0, 200, 200, 200, 120, "Image Popup-Menu Example")
  If CreatePopupImageMenu(0, #PB_Menu_ModernLook)      ; creation of the pop-up menu begins...
    MenuItem(1, "Open")       ; You can use all commands for creating a menu
    MenuItem(2, "Save")       ; just like in a normal menu...
    MenuItem(3, "Save as")
    MenuItem(4, "Quit")
    MenuBar()
    OpenSubMenu("Recent files")
      MenuItem(5, "PureBasic.exe")
      MenuItem(6, "Test.txt")
    CloseSubMenu()
  EndIf
  Repeat
    Select WaitWindowEvent()      ; check for window events
      Case #WM_RBUTTONDOWN        ; right mouse button was clicked =>
        DisplayPopupMenu(0, WindowID(0))  ; now display the popup-menu
```

```
        Case #PB_Event_Menu         ; an item of the popup-menu was clicked
          Select EventMenu()      ; get the clicked menu item...
            Case 1 : Debug "Menu: Open"
            Case 2 : Debug "Menu: Save"
            Case 3 : Debug "Menu: Save as"
            Case 4 : Quit = 1
            Case 5 : Debug "Menu: PureBasic.exe"
            Case 6 : Debug "Menu: Text.txt"
          EndSelect
        Case #PB_Event_CloseWindow
          Quit = 1
      EndSelect
    Until Quit = 1
  EndIf
```

100.6 DisplayPopupMenu

Syntax

```
DisplayPopupMenu(#Menu, WindowID [, x, y])
```

Description

Displays a PopupMenu (which must already have been created) under the current mouse position. Optionally, (x,y) co-ordinates can be specified to display the menu at a different position on screen. These co-ordinates are in pixels, starting from the upper-left corner of the screen.

For an example and a preview image see the CreatePopupMenu() help.

100.7 DisableMenuItem

Syntax

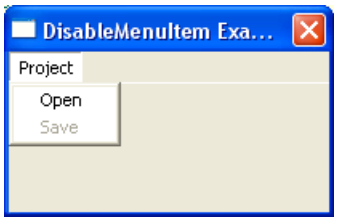
```
DisableMenuItem(#Menu, MenuItem, State)
```

Description

Disable (or enable) the #Menu item with the 'MenuItem' number. If State equals 1 then the menu item is disabled, while if State equals 0 then the menu item is enabled.

Example:

```
  If OpenWindow(0, 200, 200, 200, 100, "DisableMenuItem Example")
    If CreateMenu(0, WindowID(0))
      MenuTitle("Project")
        MenuItem(1, "Open")
        MenuItem(2, "Save")
        DisableMenuItem(0, 2, 1)      ; disable the second menu item (Save)
    EndIf

    Repeat
    Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```



100.8 FreeMenu

Syntax

```
FreeMenu(#Menu)
```

Description

This function deletes the specified menu. After calling this function that particular menu can no longer be displayed, however that menu number can now be reused.

Note: all remaining menus are automatically freed when the program ends.

100.9 GetMenuItemState

Syntax

```
Result = GetMenuItemState(#Menu, MenuItem)
```

Description

Returns the menuitem state. This is only useful for 'check' like menu item. If the menu item is checked, then the 'Result' is 1, else the 'Result' is 0. The checkbox can be put on a menu item with the SetMenuItemState() function.

Example:

```
If OpenWindow(0, 200, 200, 200, 100, "GetMenuItemState Example")
 If CreateMenu(0, WindowID(0))
   MenuTitle("Project")
     MenuItem(1, "Changed")
     SetMenuItemState(0,1,1)     ; set check mark for the previously created menu item
 EndIf
 Repeat
   Event = WaitWindowEvent()           ; wait for an event
   If Event = #PB_Event_Menu           ; a menu event appeared
     If EventMenu() = 1              ; first menu item was clicked
         If GetMenuItemState(0,1) = 1    ; actual item state = check marked
           SetMenuItemState(0,1,0)       ; now remove the check mark
         Else                            ; actual item state = no check mark
           SetMenuItemState(0,1,1)       ; now set the check mark
         EndIf
     EndIf
```

```
      EndIf
    Until Event = #PB_Event_CloseWindow
  EndIf
```

100.10 GetMenuItemText

Syntax

```
Text\$ = GetMenuItemText(#Menu, Item)
```

Description

Returns the text from the specified #Menu item.

100.11 GetMenuTitleText

Syntax

```
Text\$ = GetMenuTitleText(#Menu, Title)
```

Description

Returns the title text of the specified #Menu item.

100.12 HideMenu

Syntax

```
HideMenu(#Menu, State)
```

Description

Hide or show the specified #Menu. If State equals 0 then the menu is shown, else its hidden.

100.13 IsMenu

Syntax

```
Result = IsMenu(#Menu)
```

Description

Tests if the given #Menu number is a valid, and correctly initialized, menu. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, otherwise it will equal zero. This is the correct way to ensure that a menu is ready to use.

100.14 MenuBar

Syntax

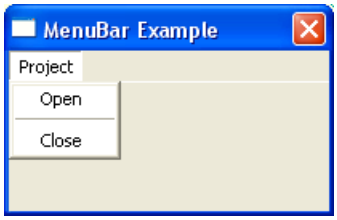
```
MenuBar()
```

Description

Creates a separator bar in the current menu.

Example:

```
If OpenWindow(0, 200, 200, 200, 100, "MenuBar Example")
  If CreateMenu(0, WindowID(0))  ; here the menu creating starts....
    MenuTitle("Project")
      MenuItem(1, "Open")
      MenuBar()                  ; here the separator bar will be inserted
      MenuItem(4, "Close")
  EndIf
  Repeat : Until WaitWindowEvent()=#PB_Event_CloseWindow
EndIf
```



100.15 MenuHeight

Syntax

```
MenuHeight()
```

Description

Returns the height in pixels of the menu title bar. This allows the correct height of a window to be calculated when using a menu.

100.16 MenuItem

Syntax

```
MenuItem(MenuItemID, Text$ [, ImageID])
```

Description

Creates a new item on the menu specified by MenuItemID. In the Text$ argument, you can use the special '&' character to underline a specific letter:

"&File" will actually display: File

To have a keyboard shortcut (will be activated with the AddKeyboardShortcut() function) aligned to the right side of the menu (for example, "Save Ctrl+S") use the tab character to give the correct space. The tab character has an ASCII code of 9 so use the function Chr() with the number 9 to insert a tab character. The code may look something like this:

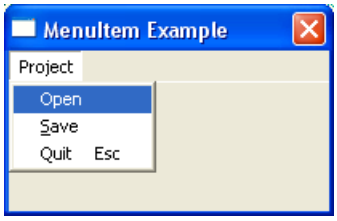
```
MenuItem(1, "&Open" + Chr(9) + "Ctrl+O")
```

'ImageID' is optional and specify which image will be used for this menu item. The menu has to be created with CreateImageMenu() or CreatePopupImageMenu() . 'ImageID' can be obtained with the ImageID() function from the Image library.

Example:

```
If OpenWindow(0, 200, 200, 200, 100, "MenuItem Example")
  If CreateMenu(0, WindowID(0))
    MenuTitle("Project")
      MenuItem(1, "Open")     ; normal item
      MenuItem(2, "&Save")    ; item with underlined character, the underline will only
                              ; be displayed, if menu is called with F10 + arrow keys
      MenuItem(3, "Quit"+Chr(9)+"Esc")    ; item with separate shortcut text
  EndIf
  Repeat : Until WaitWindowEvent()=#PB_Event_CloseWindow
EndIf
```

MacOS X: the 'Quit', 'Preferences' and 'About' items are considered as specials and need to be placed in the 'Application' menu to have the look'n'feel of OS X applications. PureBasic support the #PB_Menu_Quit, #PB_Menu_Preferences and #PB_Menu_About constants (to be specified as the 'MenuItemID') for these kind of menu items. When one of these constants is detected, the item isn't inserted in the current menu, but in the 'Application' menu. If a shortcut was specified, it is simply ignored and replaced by the standard one. These 3 constants aren't defined on others OS, to allow flexible numbering on these OS.



100.17 MenuID

Syntax

```
MenuID = MenuID(#Menu)
```

Description

Returns the unique system identifier of the #Menu.

This result is sometimes also known as 'Handle'. Look at the extra chapter "Handles and Numbers" for more information.

100.18 MenuTitle

Syntax

```
MenuTitle(Title$)
```

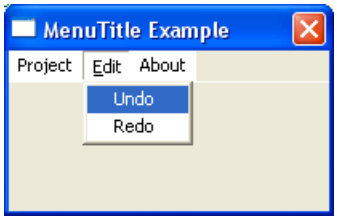
Description

Creates a new title on the menu. In the Text$ argument, you can use the special '&' character to underline a specific letter:

"&File" will actually display: File

Example:

```
If OpenWindow(0, 200, 200, 200, 100, "MenuTitle Example")
  If CreateMenu(0, WindowID(0))
    MenuTitle("Project")       ; normal menu title with following item
      MenuItem(1, "Open")
      MenuItem(2, "Close")
    MenuTitle("&Edit")         ; menu title with underlined character, the underline
                               ; will only be displayed, when called with F10 key
      MenuItem(3, "Undo")
      MenuItem(4, "Redo")
    MenuTitle("About")         ; only menu title
  EndIf
  Repeat : Until WaitWindowEvent()=#PB_Event_CloseWindow
EndIf
```



100.19 OpenSubMenu

Syntax

```
OpenSubMenu(Text$)
```

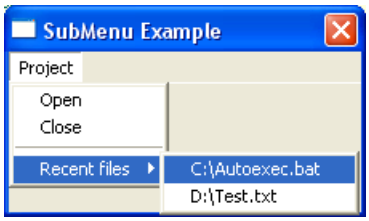
Description

Creates an empty sub-menu in the current menu. In the Text$ argument, you can use the special '&' character to underline a specific letter:

"&File" will actually display: File

Example:

```
If OpenWindow(0, 200, 200, 220, 100, "SubMenu Example")
  If CreateMenu(0, WindowID(0))
    MenuTitle("Project")
      MenuItem(1, "Open")
      MenuItem(2, "Close")
      MenuBar()
      OpenSubMenu("Recent files")        ; begin sub-menu
        MenuItem( 3, "C:\Autoexec.bat")
        MenuItem( 4, "D:\Test.txt")
      CloseSubMenu()                     ; end sub-menu
  EndIf
  Repeat : Until WaitWindowEvent()=#PB_Event_CloseWindow
EndIf
```



100.20 SetMenuItemState

Syntax

```
Result = SetMenuItemState(#Menu, MenuItem, State)
```

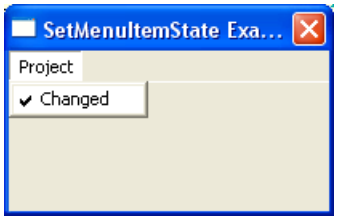
Description

Changes the specified MenuItem state. This functions allows you to display a 'check mark' next to the menu item text. The check is displayed when State equals 1, if State equals something else then the 'check mark' will not be displayed.

GetMenuItemState() can be used to retrieve the current menu item state.

Example:

```
If OpenWindow(0, 200, 200, 200, 100, "SetMenuItemState Example")
  If CreateMenu(0, WindowID(0))
    MenuTitle("Project")
      MenuItem(1, "Changed")
      SetMenuItemState(0,1,1)     ; set check mark before the previously created menu item
  EndIf
  Repeat : Until WaitWindowEvent()=#PB_Event_CloseWindow
EndIf
```



100.21 SetMenuItemText

Syntax

```
SetMenuItemText(#Menu, Item, Text$)
```

Description

Changes the specified #Menu item text.

100.22 SetMenuTitleText

Syntax

```
SetMenuTitleText(#Menu, Title, Text$)
```

Description

Changes the specified #Menu title text.

Chapter 101

Mesh

Meshs are 3D objects composed of many vertices (and triangles) which are linked together to make a shape. A mesh may have an optional skeleton, with bones to allow realtime animation. A mesh may not be moved or displayed directly, but requires to be manipulated through an entity .

InitEngine3D() must be called successfully before using the Mesh functions.

101.1 CreateMesh

Syntax

```
Result = CreateMesh(#Mesh, BoundingBoxRadius)
```

Description

Creates a new empty #Mesh. If #PB_Any is used as the '#Mesh' parameter, then the new mesh number will be returned as 'Result'.

SetMeshData() may be used to build a mesh from scratch by inserting the required vertices, normals, uv textures, etc. 'BoundingBoxRadius' specifies the bounding box radius applied to the mesh.

If Result equals 0, the creation of the mesh has failed. If the #Mesh was already created, then it is freed and replaced by a new one.

101.2 CopyMesh

Syntax

```
Result = CopyMesh(#Mesh, #NewMesh)
```

Description

Creates a #NewMesh which is the exact copy of the specified #Mesh. If #PB_Any is used as the '#NewMesh' parameter, then the new mesh number will be returned as 'Result'.

If the Result equals 0, then the mesh copy has failed. If #NewMesh already exists, then it is freed and replaced by a new one.

101.3 FreeMesh

Syntax

```
FreeMesh(#Mesh)
```

Description

Free the specified #Mesh. All its associated memory is released and the object may not be used anymore.

Note: all remaining meshes are automatically freed when the program ends.

101.4 IsMesh

Syntax

```
Result = IsMesh(#Mesh)
```

Description

Tests if the given #Mesh is valid and the mesh has been correctly initialized.

This function is bullet-proof and may be used with any value. If Result equals zero then the given mesh has not been properly created or initialized. This is the correct way to ensure that a mesh is ready to use.

101.5 LoadMesh

Syntax

```
Result = LoadMesh(#Mesh, Filename$)
```

Description

Loads a new #Mesh from disk. If #PB_Any is used as the '#Mesh' parameter, then the new mesh number will be returned as 'Result'. Before loading a mesh, an archive must be specified with Add3DArchive() .

Mesh is required to be in the OGRE .mesh format. it is possible to create or convert existing meshes into this format then you may use an existing exporter, such as Milkshape, Lightwave or 3DS Max. More information may be found on the OGRE website.

If Result equals 0, then the Mesh may not be loaded. If the #Mesh already exists, then it is freed and replaced by a new one.

101.6 MeshID

Syntax

```
Result = MeshID(#Mesh)
```

Description

Returns the unique MeshID of the #Mesh. The use of this function is required especially by the CreateEntity() function.

101.7 SetMeshData

Syntax

```
SetMeshData(#Mesh, Type, Data, DataLength)
```

Description

Set specific data to a #Mesh. 'Data' is the memory address of the data to be set in the #Mesh. 'DataLength' is the length of the data, depending of its type. This function is mainly used in combination with CreateMesh() .

'Type' may be one of the following values:

```
  #PB_Mesh_Vertex       : 'Data' is an array of vertices,
   'DataLength' is the number of elements
  #PB_Mesh_Face         : 'Data' is an array of triangles, each
   triangle vertex is an index of the vertices array. The array
\\


  must be of 'word' type (.w). 'DataLength' is the number of
   elements
  #PB_Mesh_UVCoordinate : 'Data' is an array of UVCoordinates,
   'DataLength' is the number of elements
  #PB_Mesh_Normal       : 'Data' is an array of Normals,
   'DataLength' is the number of elements
  #PB_Mesh_Color        : 'Data' is an array of Colors,
   'DataLength' is the number of elements
```

Chapter 102

Misc

Here are a set of various functions which fall outside the other headings within this document.

102.1 Delay

Syntax

```
Delay(Time)
```

Description

Halts the program excecution by the amount of time (in milliseconds) specified by the Time argument.

Delay is especially useful in event loops with WindowEvent() or ExamineKeyboard() , so these functions do not "eat" the entire CPU power.

102.2 ElapsedMilliseconds

Syntax

```
Result = ElapsedMilliseconds()
```

Description

Returns the number of milliseconds that have ellapsed since a specific time in the past. The absolute value returned is of no use since it varies depending on the operating system. Instead, this function should be used to calculate time differences between multiple ElapsedMilliseconds() calls.

This function is relatively accurate: it may have a slight variation, depending on which operating system it is excecuted on, this is due to the fact that some systems have a lower timer resolution than others.

```
StartTime = ElapsedMilliseconds()              ; Get the actual value
Delay(1000)                                    ; Wait 1000 milliseconds
ElapsedTime = ElapsedMilliseconds()-StartTime  ; 'ElapsedTime' value should be about 1000 milliseconds
```

102.3 OSVersion

Syntax

```
Result = OSVersion()
```

Description

Returns the version of the operating system on which the program has been launched.

'Result' can be one of the following values:

```
#PB_OS_Windows_NT3_51
#PB_OS_Windows_95
#PB_OS_Windows_NT_4
#PB_OS_Windows_98
#PB_OS_Windows_ME
#PB_OS_Windows_2000
#PB_OS_Windows_XP
#PB_OS_Windows_Server_2003
#PB_OS_Windows_Vista
#PB_OS_Windows_Server_2008
#PB_OS_Windows_7
#PB_OS_Windows_Future  ; New Windows version (not existing
 when the program was written)
```

Example:

```
Select OSVersion()
  Case #PB_OS_Windows_98
    MessageRequester("Info", "Windows 98")

  Case #PB_OS_Windows_2000
    MessageRequester("Info", "Windows 2000")

  Case #PB_OS_Windows_XP
    MessageRequester("Info", "Windows XP")

  Default
    MessageRequester("Info", "Unsupported Windows version")
EndSelect
```

Note: the value of these constants are established before the release date of each version, therefore tests such as the one which follows, may be used, in order to catch all versions which are older or newer than a given one:

```
If OsVersion() < #PB_OS_Windows_2000
  ;
  ; All versions older than Windows 2000
  ;
EndIf
```

102.4 Random

Syntax

```
Result = Random(Maximum)
```

Description

Returns a random number (integer) which lies between (and including) 0 and the Maximum value. 'Maximum' may not exceed the positive long value: 2147483647.

Additionally RandomSeed() may be used to change the random number seed. RandomData() can be used to fill a memory buffer with random data.

Note: This command uses a pseudorandom number generator which is very fast and produces randomly looking output but it is not strong enough for cryptographic purposes. The slower CryptRandom() command can be used if more secure random number generation is required.

Example:

```
Repeat
  Dice = Random(5) + 1  ; get a value between 1 and 6
  Choice = MessageRequester("Roll the dice", "You got a "+Str(Dice)+", roll again?", #PB_MessageRequester_YesNo)
Until Choice = #PB_MessageRequester_No
```

102.5 RandomData

Syntax

```
RandomData(*Buffer, Length)
```

Description

Fills the specified memory buffer with random data. This command uses the same random seed as the Random() command. RandomSeed() may be used to change the random number seed.

Note: This command uses a pseudorandom number generator which is very fast and produces randomly looking output but it is not strong enough for cryptographic purposes. The slower CryptRandomData() command can be used if more secure random number generation is required.

Example:

```
; Create an image with random content
;
CreateImage(0, 200, 200)
If StartDrawing(ImageOutput(0))
  *Buffer = DrawingBuffer()
  Pitch   = DrawingBufferPitch()

  RandomData(*Buffer, Pitch*200)

  StopDrawing()
EndIf

OpenWindow(0, 0, 0, 200, 200, "Random Image", #PB_Window_SystemMenu|#PB_Window_ScreenCentered)
ImageGadget(0, 0, 0, 200, 200, ImageID(0))

Repeat
Until WaitWindowEvent() = #PB_Event_CloseWindow
```

102.6 RandomSeed

Syntax

```
RandomSeed(Value)
```

Description

Changes the actual random number seed for the values returned with Random() .

Note: each time a program is started, a new seed is generated, therefore RandomSeed() is useful only when the goal is to generate the same random numbers in the same order every time the program is executed.

102.7 Red

Syntax

```
Result = Red(Color)
```

Description

Returns the red value of a 24 bit RGB color (between 0 and 255).

To combine red, green and blue values in order to create a 24 bit RGB color, use the RGB() function. These functions are useful to perform Drawing operations .

102.8 Green

Syntax

```
Result = Green(Color)
```

Description

Returns the green value of a 24 bit RGB color (between 0 and 255).

To combine red, green and blue values in order to create a 24 bit RGB color, use the RGB() function. These functions are useful to perform Drawing operations .

102.9 Blue

Syntax

```
Result = Blue(Color)
```

Description

Returns the blue value of a 24 bit RGB color (between 0 and 255).

To combine red, green and blue values in order to create a 24 bit RGB color, use the RGB() function. These functions are useful to perform Drawing operations .

102.10 Alpha

Syntax

```
Result = Alpha(Color)
```

Description

Returns the alpha value of a 32 bit RGBA color (between 0 and 255). A value of 0 means fully transparent and a value of 255 means fully opaque.

To combine red, green, blue and alpha values in order to create a 32 bit RGB color, use the RGBA() function. These functions are useful to perform Drawing operations .

102.11 RGB

Syntax

```
Color = RGB(Red, Green, Blue)
```

Description

Returns the 24bit color value corresponding to the Red, Green, Blue values. As always each value is between 0 and 255, making available up to 16.7 Million colors.

To extract the red, green and blue values from a 24 Bit color value, use the Red() , Green() and Blue() functions. These functions are useful to perform Drawing operations .

A color table with common colors is available here .

102.12 RGBA

Syntax

```
Color = RGBA(Red, Green, Blue, Alpha)
```

Description

Returns the 32bit color value corresponding to the Red, Green, Blue and Alpha values. As always each value is between 0 and 255, making available up to 16.7 Million colors with 256 levels of alpha transparency. An Alpha value of 0 means fully transparent and a value of 255 means fully opaque.

To extract the red, green, blue or alpha values from a 32 Bit color value, use the Red() , Green() , Blue() and Alpha() functions. These functions are useful to perform Drawing operations .

102.13 AlphaBlend

Syntax

```
Color = AlphaBlend(Color1, Color2)
```

Description

Returns the resulting 32 bit color from blending the 32 bit color Color1 onto the 32 bit color Color2. The RGBA() function can be used to create 32 bit colors with alpha transparency.

These functions are useful to perform Drawing operations .

Chapter 103

Module

PureBasic can replay standard music modules to have nice music background during a game or an application. The modules are well known from Demo makers, as its an efficient way to build music on a computer. The tools used to create the modules are named 'Trackers' (ProTracker, FastTracker, Impulse Tracker...). The advantages of a module against a .wav/mp3 file are its very low size, a virtually endless length, the very fast replay, possible jump on a certain music part to fit to the screen action etc... It's of course possible to mix standard sound and module sound to have both played simultaneously.

The ModPlug library is used to have a very high quality module playback and multiple tracker support.

The command InitSound() has to be called with success before using any of the module related functions.

103.1 CatchModule

Syntax

```
Result = CatchModule(#Module, MemoryAddress, Size)
```

Description

Load the specified module from the given memory area. If #PB_Any is used as '#Module' parameter, the new module number will be returned as 'Result'. If the 'Result' is not 0, the module has been correctly loaded and is ready to be played with the function PlayModule() , else the module loading has failed. If another module was previously loaded with the same #Module number, then the previous module is automatically freed.

ModPlug supports a lot of module formats, which includes: Protracker (4 channels), FastTracker (up to 32 channels, 16 bit quality), Impulse Tracker, etc.

This function is useful when using the 'IncludeBinary ' PureBasic keyword. Then modules can be packed inside the executable. Nevertheless, use this option with care, as it will take more memory than storing the module in an external file (the module is both in executable memory and load in physical memory).

Example:

```
CatchModule(0, ?Module, ?ModuleEnd-?Module)
End

DataSection
  Module:
    IncludeBinary "Module.xm"
```

```
    ModuleEnd:
  EndDataSection
```

103.2 FreeModule

Syntax

```
FreeModule(#Module)
```

Description

Stop and remove the specified module previously loaded with the LoadModule() function from memory. Once a module has been freed, it can't be played anymore.

Note: all remaining modules are automatically freed when the program ends.

103.3 GetModulePosition

Syntax

```
Position = GetModulePosition(#Module)
```

Description

Returns the current pattern position of the playing #Module. The first pattern begin to 0.

103.4 GetModuleRow

Syntax

```
Row = GetModuleRow(#Module)
```

Description

Returns the row position in the pattern of the playing #Module. The first row begin to 0.

103.5 IsModule

Syntax

```
Result = IsModule(#Module)
```

Description

Tests if the given #Module number is a valid and correctly initialized module. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a module is ready to use.

103.6 LoadModule

Syntax

```
Result = LoadModule(#Module, Filename$)
```

Description

Load the specified module into memory. If #PB_Any is used as '#Module' parameter, the new module number will be returned as 'Result'. If the 'Result' is not 0, the module has been correctly loaded and is ready to be played with the function PlayModule() , else the module loading has failed. If another module was previously loaded with the same #Module number, then the previous module is automatically freed.

ModPlug supports a lot of module formats, which includes: Protracker (4 channels), FastTracker (up to 32 channels, 16 bit quality), Impulse Tracker, etc.

103.7 ModuleVolume

Syntax

```
Result = ModuleVolume(#Module, Volume)
```

Description

Change the master volume of the specified module, in realtime. Volume value can be between 0 and 100. Useful for nice fadein/fadeout.

103.8 PlayModule

Syntax

```
PlayModule(#Module)
```

Description

Start to play the specified module previously loaded with the LoadModule() function.

103.9 SetModulePosition

Syntax

```
SetModulePosition(#Module, Position)
```

Description

Change the current pattern position of the playing #Module to the new one. The first pattern begin to 0.

103.10 StopModule

Syntax

```
StopModule(#Module)
```

Description

Stop the #Module (if it was playing). If '#Module' is equal to -1, then all the modules currently playing are stopped.

Chapter 104

Mouse

PureBasic provides a full access to mouses plugged into the computer. It supports standard mouses with up to 3 buttons. This library is optimized and uses very low level functions especially for games. Do not use the functions of this library in a regular application, in this case carry out the mouse queries with WindowMouseX() , WindowMouseY() and EventType() . Under Windows the DirectX is used.

104.1 InitMouse

Syntax

```
Result = InitMouse()
```

Description

Initialize the Mouse environment for later use. You should call this function before any other functions in this library. If the result is null, no mouse is available. This function tries to open DirectX (v3.0 with NT4.0 compatibility or v7.0 else); if it fails then DirectX is either not avaliable or is too old be used.

104.2 ExamineMouse

Syntax

```
Result = ExamineMouse()
```

Description

Update the mouses' status. If it returns 0, the mouse can't be used. This function should be used before MouseDeltaX() , MouseDeltaY() , MouseX() , MouseY() or MouseButton() . For now, only the default mouse is supported.

104.3 MouseButton

Syntax

```
Result = MouseButton(ButtonNumber)
```

Description

Returns 0 if the specified button number is not pressed, else the button is pressed. Any number of buttons can be pressed at the same time. ExamineMouse() must be called before this function to update the actual button's state.

'ButtonNumber' can be one of the following constants:

```
#PB_MouseButton_Left   : Tests if the left mouse button is
 pressed
#PB_MouseButton_Right  : Tests if the right mouse button is
 pressed
#PB_MouseButton_Middle: Tests if the middle mouse button is
 pressed
```

104.4 MouseDeltaX

Syntax

```
Result = MouseDeltaX()
```

Description

Returns the mouse X movement (in pixels) since the last call of this function. This means that the Result can be either a negative or positive value, depening on whether or not the mouse has been moved to left or right since the last call. ExamineMouse() should be called before this function to update the actual mouse position.

104.5 MouseDeltaY

Syntax

```
Result = MouseDeltaY()
```

Description

Returns the mouse Y movement (in pixels) since the last call of this function. This means that the Result can be either a negative or positive value, depening on whether or not the mouse has been moved to top or bottom since the last call.. ExamineMouse() should be called before this function to update the actual mouse position.

104.6 MouseLocate

Syntax

```
Result = MouseLocate(x, y)
```

Description

Changes the absolute position (in pixels) of the mouse in the current screen. This is useful when using the MouseX() or MouseY() functions.

104.7 MouseWheel

Syntax

```
Result = MouseWheel()
```

Description

Returns the number of ticks the mouse wheel has moved since the last funciton call. The value is positive if the wheel has moved forward and negative is the wheel has moved backwards. ExamineMouse() should be called before this function to update the mouse status.

Supported OS

Windows

104.8 MouseX

Syntax

```
Result = MouseX()
```

Description

Returns the actual mouse X position (in pixel) on the current screen. The position can be changed easily with the MouseLocate() function. ExamineMouse() should be called before this function to update the actual mouse position.

104.9 MouseY

Syntax

```
Result = MouseY()
```

Description

Returns the actual mouse Y position (in pixels) on the current screen. The position can be changed easily with the MouseLocate() function. ExamineMouse() should be called before this function to update the actual mouse position.

104.10 ReleaseMouse

Syntax

```
Result = ReleaseMouse(State)
```

Description

Lock or release the mouse to allow its use under standard OS. This is tipically called after checking the result of IsScreenActive() function. If 'State' = 1, the mouse is released, else the mouse is locked by the PureBasic program.

Chapter 105

Movie

PureBasic provides simple, yet powerful functions to integrate movie playback inside an application or game.

Windows: as it uses the DirectX 7 technology (DirectShow), any kind of media can be played with this library: AVI, MPG, DivX, Mp3 etc.

MacOS X: it uses QuickTime technology, therefore any kind of media (depending of the installed plugins) can be played with this library: AVI, MPG, DivX, Mp3 etc.

105.1 FreeMovie

Syntax

```
FreeMovie(#Movie)
```

Description

Free the specified #Movie and remove it from memory. This movie is no longer valid.

Note: all remaining movies are automatically freed when the program ends.

105.2 InitMovie

Syntax

```
Result = InitMovie()
```

Description

Initialize the movie environment for later use. You must call this function before any other functions in this library. This function tries to open DirectX (v3.0 with NT4.0 compatibility or v7.0 else); if it fails then DirectX is either not avaliable or is too old be used.

105.3 IsMovie

Syntax

```
Result = IsMovie(#Movie)
```

Description

Tests if the given #Movie number is a valid and correctly initialized movie. This function is bullet-proof and can be used with any value.If the 'Result' is not zero then the object is valid and initialized, otherwise it will equal zero. This is the correct way to ensure that a movie is ready to use.

105.4 LoadMovie

Syntax

```
Result = LoadMovie(#Movie, Filename$)
```

Description

Try to open, and prepare, a movie for later playback. If #PB_Any is used as '#Movie' parameter, the new movie number will be returned as 'Result'. If the Result is 0, the movie can't be opened (format not supported or file not found), otherwise its ok. PlayMovie() can be used to start playing it.

105.5 MovieAudio

Syntax

```
MovieAudio(#Movie, Volume, Balance)
```

Description

Control the audio stream of #Movie. Volume and balance can be modified during playback. Changes occur immediately. The volume range goes from 0 to 100 (100 being the loudest). Balance can range from -100 to 100 (-100 is full left, 0 is middle (normal mode) and 100 is full right).

Note: this command is not supported on rendered movies.

Supported OS

Windows, MacOS X

105.6 MovieHeight

Syntax

```
Height = MovieHeight(#Movie)
```

Description

Returns the height (in pixels) of the #Movie. If the result equals -1, then no video stream is found (or isn't compatiable), however the audio stream can still be played. This is especially useful from playing back MP3 files using the movie functions (!).

105.7 MovieInfo

Syntax

```
Result = MovieInfo(#Movie, Flags)
```

Description

Returns additional information about the #Movie.

Supported values for now:
0: return the number of frame per seconds (*1000).

Note: This function only works with real movies correctly, as no FPS value can be retrieved from a sound (e.g. MP3) since they don't contian any (picture) frames.

Supported OS

Windows, MacOS X

105.8 MovieLength

Syntax

```
Length = MovieLength(#Movie)
```

Description

Returns the length (in frames) of the #Movie.

Note: This function only works with real movies correctly, as sounds (e.g. MP3) don't contian any (picture) frames.

Supported OS

Windows, MacOS X

105.9 MovieSeek

Syntax

```
Result = MovieSeek(#Movie, Frame)
```

Description

Change the #Movie position to the given frame.

Note: this command is not supported on rendered movies.

Supported OS

Windows, MacOS X

105.10 MovieStatus

Syntax

```
Result = MovieStatus(#Movie)
```

Description

Get the status of the #Movie.

'Result' can be one of the following values:

-1: Movie is paused.
0: Movie is stopped
> 0: Movie is playing. The returned value is the current frame number displayed.

Note: Be careful with this function when using sounds (e.g. MP3) as no pictures (frames) exist.

105.11 MovieWidth

Syntax

```
Width = MovieWidth(#Movie)
```

Description

Returns the width (in pixels) of the #Movie. If the result is -1, then no video stream is found (or isn't compatiable). The audio stream can still be played.

This is especially useful from playing back MP3 files using the movie functions (!).

105.12 PauseMovie

Syntax

```
PauseMovie(#Movie)
```

Description

Pause the #Movie playback, it can be resumed using the ResumeMovie() function.

105.13 PlayMovie

Syntax

```
PlayMovie(#Movie, WindowID)
```

Description

Start to play a movie previously loaded with LoadMovie() on the specified window. The WindowID value can be easily obtained by using the WindowID() function from the Windows library.

The function ResizeMovie() can be used to resize and move the movie on this window (to not use the full window area, for example).

If the movie is intended to be rendered directly on video surfaces with RenderMovieFrame() then the 'WindowID' value has to be #PB_Movie_Rendered.

It's also possible to play a movie on a full screen, simply use the result of ScreenID() as 'WindowID'.

105.14 RenderMovieFrame

Syntax

```
RenderMovieFrame(#Movie, SpriteID)
```

Description

Render the #Movie frame directly on a sprite, instead of the video overlay. This is a very powerful function which can be used to achieve special effects in realtime (like video playback realtime transparency and zooming/rotating using the Sprite3D functions). Not all the codecs are compatible with this function and for now only MPEG files seem to work everytime. A valid SpriteID can be easily obtained with SpriteID() . To use this function, #PB_Movie_Rendered has to be specified in the PlayMovie() function.

Note: this command is only supported on DirectX7 subsystem.

Supported OS

Windows

105.15 ResizeMovie

Syntax

```
ResizeMovie(#Movie, x, y, Width, Height)
```

Description

Resize and move the #Movie display area on the movie window. This is especially useful before start of play with PlayMovie() .

105.16 ResumeMovie

Syntax

```
ResumeMovie(#Movie)
```

Description

Continue to play the #Movie, after a PauseMovie() function.

105.17 StopMovie

Syntax

```
StopMovie(#Movie)
```

Description

Stop playing the #Movie. If the movie is played again, it will restart from the beginning.

Chapter 106

Network

PureBasic supports the official protocol to exchange data via the internet: TCP/IP. This means that games and applications can be written using the well-established client/server model. With these functions, its possible to create any kind of internet-related application (browsers, web servers, ftp clienst, etc) or even multiplayer games.

106.1 CloseNetworkConnection

Syntax

```
CloseNetworkConnection(Connection)
```

Description

Close the specified connection. If a client calls this, the server will recieve a #PB_NetworkEvent_Disconnect event. If this is called from a server, then the connection will be closed without any notice to the client.

Note: all remaining opened connections are automatically closed when the program ends.

106.2 ConnectionID

Syntax

```
Result = ConnectionID(Connection)
```

Description

Returns the unique system identifier of the 'Connection'.

This result is sometimes also known as 'Handle'. Look at the extra chapter Handles and Numbers for more information.

106.3 ServerID

Syntax

```
Result = ServerID(#Server)
```

Description

Returns the unique system identifier of the '#Server'.

This result is sometimes also known as 'Handle'. Look at the extra chapter Handles and Numbers for more information.

106.4 CloseNetworkServer

Syntax

```
CloseNetworkServer(#Server)
```

Description

Shutdown the specified running #Server. All clients connected to this server are automatically removed. The port is freed and can be reused.

106.5 CreateNetworkServer

Syntax

```
Result = CreateNetworkServer(#Server, Port [, Mode])
```

Description

Create a new network #Server on the local computer using the specified port. If #PB_Any is used as '#Server' parameter, the new server number will be returned as 'Result'.

Port values can range from 6000 to 7000 (this is a recommended area space, but it can go from 0 to 65000 in reality).

Any number of servers can run simultaneously on the same computer, the only restriction being that no two servers can run using the same port.

If Result equals 0, the server can't be created (the port is in use), otherwise the server has been correctly created and is ready to use.

Port: Port number for this server

The optional 'Mode' parameter can be one of the following values:

```
#PB_Network_TCP: The server will use the TCP network protocol
 (default)
#PB_Network_UDP: The server will use the UDP network protocol
```

106.6 ExamineIPAddresses

Syntax

```
ExamineIPAddresses()
```

Description

Start examining the available IP addresses on the local computer. NextIPAddress() is used to retrieve each IP number.

106.7 HostName

Syntax

```
String\$ = HostName()
```

Description

Returns the computer's hostname.

106.8 InitNetwork

Syntax

```
Result = InitNetwork()
```

Description

This function must be called before any other function from the Network library.

If Result equals 0, then there is no TCP/IP stack available on the system.

106.9 IPString

Syntax

```
String\$ = IPString(IPAddress)
```

Description

Returns the string representation in dotted form ("127.0.0.1") of the specified numerical IPAddress.

106.10 IPAddressField

Syntax

```
Result = IPAddressField(IPAddress, Field)
```

Description

Returns the given field value of the specified IP address.

'IPAddress' can be a value created with MakeIPAddress() . Field can be a value between 0 and 3 (0 being the leftmost value, 3 being the rightmost).

For example 127.0.0.1: field 0 will be '127', and field 3 will be '1'.

This function is especially useful with:

- IPAddressGadget()
- MakeIPAddress()

106.11 MakeIPAddress

Syntax

```
Result = MakeIPAddress(Field0, Field1, Field2, Field3)
```

Description

Returns the equivalent numeric value of the specified IP address. Each field can only have a value between 0 and 255. The numeric number can be converted back into string in dotted notation with the IPString() function.

This function is especially useful with:

- IPAddressGadget()

106.12 EventServer

Syntax

```
Server = EventServer()
```

Description

This function is only needed on the server side. It returns the handle of the server which has recived data, allowing multiple servers to be managed on one computer.

106.13 EventClient

Syntax

```
Client = EventClient()
```

Description

This function is only needed on the server side. It returns the ID of the client which has sent data.

The functions GetClientIP() and GetClientPort() can be used to get more information on the client which send the data.

106.14 GetClientIP

Syntax

```
IP = GetClientIP(Client)
```

Description

This function should be called after EventClient() and will return the IP address of the client. GetClientPort() is available to get the client's port. IPString() can be used to convert easily the IP address into a string.

106.15 GetClientPort

Syntax

```
Port = GetClientPort(Client)
```

Description

This function should be called after EventClient() and will return the port of the client. The GetClientIP() function is available to get the client's IP address.

106.16 NetworkClientEvent

Syntax

```
Result = NetworkClientEvent(Connection)
```

Description

Result equals 0 if no data has been received on the specified client port, otherwise there is some data which should be processed.

The returned value indicates what has actually happened:

```
  0                     : Nothing has happened
  #PB_NetworkEvent_Data: Raw data has been received (to be read
   with ReceiveNetworkData()
)
  #PB_NetworkEvent_File: A file has been received (to be read
   with ReceiveNetworkFile()
)
```

106.17 NetworkServerEvent

Syntax

```
Result = NetworkServerEvent()
```

Description

Result equals 0 if no data has been received on the server port, otherwise there is some data which should be processed.

The returned value indicates what has actually happened:

```
  0                          : Nothing has happened
  #PB_NetworkEvent_Connect   : A new client has been connected
   to the server
  #PB_NetworkEvent_Data      : Raw data has been received (to
   be read with ReceiveNetworkData()
)
  #PB_NetworkEvent_File      : A file has been received (to be
   read with ReceiveNetworkFile()
)
  #PB_NetworkEvent_Disconnect: A client has quit the server
   (disconnection)
```

106.18 NextIPAddress

Syntax

```
Result = NextIPAddress()
```

Description

This function must be called after ExamineIPAddresses() . It will return the available IP addresses of the computer in numerical form. They can be easily converted into strings with IPString() .

If Result equals 0, then no more IP addresses can be found.

106.19 OpenNetworkConnection

Syntax

```
Connection = OpenNetworkConnection(ServerName$, Port [, Mode])
```

Description

Trys to open a connection to the specified server. 'ServerName$' can be an IP address or a full name (ie: "127.0.0.1" or "ftp.home.net").

If the connection has be granted by the server, then 'Connection' will not equal NULL, otherwise the connection has failed. 'Connection' is needed by all receive and send functions.

The optional 'Mode' parameter can be one of the following value:

```
  #PB_Network_TCP: The connection will use the TCP network
   protocol (default)
  #PB_Network_UDP: The connection will use the UDP network
   protocol. The connection will not be
                  explicitely created, as UDP is a
   connectionless protocol, but it will
                  add an entry in the PureBasic management
   stack and allow to send
                  data via UDP using the regular
   SendNetworkData()
 (and related) functions.
```

106.20 ReceiveNetworkData

Syntax

```
Result = ReceiveNetworkData(Connection, *DataBuffer,
   DataBufferLength)
```

Description

Receives a raw data from the specified client. This function can be used by both client and server applications.

On the server side 'Connection' is the client which has send the data (ClientID can be easily obtained with EventClient()).
On client side, 'Connection' is returned by OpenNetworkConnection() .

The data is read into the specified *DataBuffer. A memory area can be created using AllocateMemory() .

The 'Result' returns the number of bytes effectively read. If 'Result' is equal to DataBufferLength, then more data is available to be read. If an error occured on the connection (link broken, connection close by the server etc.), 'Result' will be -1.

On UDP connections, the maximum 'DataBufferLength' is 2048. On TCP connections, the maximum 'DataBufferLength' is 65536.

106.21 ReceiveNetworkFile

Syntax

```
ReceiveNetworkFile(Connection, Filename$)
```

Description

Receive an entire file from the specified client. This function can be used by both client and server applications.

On the server side, 'Connection' is the client which has sent the file.
On the client side, 'Connection' is returned by OpenNetworkConnection() .

The file must have been sent specifically by using SendNetworkFile() function.

106.22 SendNetworkData

Syntax

```
Result = SendNetworkData(Connection, *MemoryBuffer, Length)
```

Description

Send raw data to the specified client. This function can be used by both client and server applications.

On the server side, 'Connection' is the client which should receive this data.
On the client side, 'Connection' is returned by OpenNetworkConnection() .

The 'Result' is the number of bytes, that was actually sent. If it is not equal to the 'Length' parameter, the receiving buffer of the user is probably full. If nothing could be sent then 'Result' will equal -1.

On UDP connections, the maximum 'Length' is 2048. On TCP connections, the maximum 'Length' is 65536.

106.23 SendNetworkFile

Syntax

```
Result = SendNetworkFile(Connection, Filename$)
```

Description

Send a file to the specified client. This function can be used by both client and server applications.

On the server side, 'Connection' is the client which should receive this data.
On the client side, 'Connection' is returned by OpenNetworkConnection() .

The file is sent using very specific (and safe) methods. It must be received with the ReceiveNetworkFile() function. This functions locks the program execution until the whole file has been sent.

'Result' will equal 1 if the file was successfully sent, it'll equal 0 otherwise.

106.24 SendNetworkString

Syntax

```
SendNetworkString(Connection, String$)
```

Description

Send a string to the specified client. This function can be used by both client and server applications.

On the server side, 'Connection' is the client which should receive this data.
On the client side, 'Connection' is returned by OpenNetworkConnection() .

Note: There is no ReceiveNetworkString(), SendNetworkString() only provides a solution to quickly send strings. The string will be sent without the terminating NULL character, and can be received using ReceiveNetworkData() , after NetworkServerEvent() / NetworkClientEvent() returned #PB_NetworkEvent_Data). In unicode mode the string is sent as UTF-8, which is processor independant (unlike UTF-16 which is tied to processor endianness).

Chapter 107

Node

Nodes are containers which can be used to group objects like entities , sound , camera , billboard and even nodes together. Once an object is attached to a node, its position and movement is relative to the node position. It allows easy hierarchical object management.

InitEngine3D() should be called successfully before using the node functions.

107.1 AttachNodeObject

Syntax

```
Result = AttachNodeObject(#Node, ObjectID, Type)
```

Description

Attach an existing object to a #Node. The supported objects are the followings:

```
  - Entity : use EntityID()
 as 'ObjectID' and #PB_Node_Entity as 'Type'.
  - Sound3D: use Sound3D()
 as 'ObjectID' and #PB_Node_Sound3D as 'Type'.
  - Camera : use CameraID()
 as 'ObjectID' and #PB_Node_Camera as 'Type'.
  - Node   : use NodeID()
 as 'ObjectID' and #PB_Node_Node as 'Type'.
  - BillboardGroup : use BillboardGroupID()
 as 'ObjectID' and #PB_Node_BillboardGroup as 'Type'.
  - ParticleEmitter: use ParticleEmitterID()
 as 'ObjectID' and #PB_Node_ParticleEmitter as 'Type'.
```

An object can be detached from a node with DetachNodeObject() .

107.2 DetachNodeObject

Syntax

```
Result = DetachNodeObject(#Node, ObjectID, Type)
```

Description

Detach a previously attached object from a #Node. The supported objects are the followings:

```
 - Entity : use EntityID()
as 'ObjectID' and #PB_Node_Entity as 'Type'.
 - Sound3D: use SoundID3D()
as 'ObjectID' and #PB_Node_Sound3D as 'Type'.
 - Camera : use CameraID()
as 'ObjectID' and #PB_Node_Camera as 'Type'.
 - Node   : use NodeID()
as 'ObjectID' and #PB_Node_Node as 'Type'.
 - BillboardGroup : use BillboardGroupID()
as 'ObjectID' and #PB_Node_BillboardGroup as 'Type'.
 - ParticleEmitter: use ParticleEmitterID()
as 'ObjectID' and #PB_Node_ParticleEmitter as 'Type'.
```

An object can be attached to a node with AttachNodeObject() .

107.3 CreateNode

Syntax

```
Result = CreateNode(#Node [, x, y, z])
```

Description

Creates a new #Node at the specified position. If #PB_Any is used as '#Node' parameter, the new node number will be returned as 'Result'.

If the 'Result' is 0, the #Node creation has failed. If the #Node was already created, it is automatically freed and replaced by the new one.

107.4 NodeID

Syntax

```
NodeID = NodeID(#Node)
```

Description

Returns the unique ID which identifies the given '#Node' in the operating system. This function is very useful when another library needs a window reference.

107.5 NodeLocate

Syntax

```
NodeLocate(#Node, x, y, z)
```

Description

Changes the #Node absolute position relative to its parent node. If the node doesn't have a parent node, it changes the #Node absolute position in the world.

107.6 NodeLookAt

Syntax

```
NodeLookAt(#Node, x, y, z)
```

Description

Changes the #Node orientation in the world, to point towards the specified x,y,z point.

107.7 NodeX

Syntax

```
Result = NodeX(#Node)
```

Description

Returns the 'x' position of the #Node relative to its parent node. If the node doesn't have a parent node, it returns the 'x' position in the world.

107.8 NodeY

Syntax

```
Result = NodeY(#Node)
```

Description

Returns the 'y' position of the #Node relative to its parent node. If the node doesn't have a parent, it returns the 'y' position in the world.

107.9 NodeZ

Syntax

```
Result = NodeZ(#Node)
```

Description

Returns the 'z' position of the #Node relative to its parent node. If the node doesn't have a parent, it returns the 'z' position in the world.

107.10 FreeNode

Syntax

```
FreeNode(#Node)
```

Description

Free the specified #Node created with CreateNode() before. All its associated memory is released and this object can't be used anymore. The attached objects are not freed automatically, and can be re-used.

Note: All remaining nodes are automatically freed when the program ends.

107.11 IsNode

Syntax

```
Result = IsNode(#Node)
```

Description

Tests if the given #Node is a valid and correctly initialized entity.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that an entity is ready to use.

107.12 MoveNode

Syntax

```
MoveNode(#Node, x, y, z)
```

Description

Moves the #Node in the world according to the specified x,y,z values.

This is a relative move based on the previous node location. To perform an absolute move in the world, use NodeLocate() .

107.13 RotateNode

Syntax

```
RotateNode(#Node, x, y, z [, Mode])
```

Description

Rotates the #Node according to the specified x,y,z angle values. All angles are in degree, with values ranging from 0 to 360. The optional 'Mode' can be one of the following value:

```
PB_Absolute: absolute rotation (default).
PB_Relative: relative rotation based on the previous node
 rotation.
```

Chapter 108

OnError

This library provides a way to track program errors (program crashes) similar to the PureBasic debugger but without the drawbacks of a bigger filesize and lower execution speed when using the debugger. This way the final version of a program which is shipped to the end user can still intercept program errors and provide some information about the error to the user so he can report it back to the developper.

The PureBasic debugger is still the better tool for finding bugs during the development phase, as it provides much more detailed information about the state of the program (like actual variable values) and also has a number of interactive features to quickly locate bugs.

Note: If both this library and the PureBasic debugger are used, not all errors will be caught by the OnError library as some checks are made and reported by the debugger even before the actual code with the error is executed.

This library can provide information about the sourcecode file and line at which the error occured through the ErrorFile() and ErrorLine() commands, but only if this feature is enabled when compiling the program (it causes a very small slowdown in execution speed to track the line numbers). To enable this feature, enable the "Enable OnError lines support" checkbox in the compiler options or specify the /LINENUMBERING (Windows) or –linenumbering (Linux, Mac OSX) commandline switch when compiling from the commandline .

108.1 OnErrorExit

Syntax

```
OnErrorExit()
```

Description

Changes the action taken if an error occurs to directly exit the program, even if the default action on the system is not to exit the program on this kind of error. The system may display an error dialog or print an error message on the console on exit.

To exit the program silently and supress any system message, use OnErrorCall() and end the program from the error handler.

Example:

```
MessageRequester("OnError test", "Test start")

OnErrorExit()
```

```
Pokes(10, "Hello World") ; Cause a #PB_OnError_InvalidMemory error

MessageRequester("OnError test", "This should never be displayed")
```

108.2 OnErrorCall

Syntax

```
OnErrorCall(@ErrorHandler())
```

Description

Changes the action taken if an error occurs to call the specified handler procedure. The handler can display information about the error to the user using the commands of this library and do any needed cleanup to shutdown the application. The program will end as soon as the handler returns.

The address of a procedure of the following form:

```
Procedure ErrorHandler()

EndProcedure
```

Example:

```
Procedure ErrorHandler()
  MessageRequester("OnError test", "The following error happened: " + ErrorMessage())
EndProcedure

MessageRequester("OnError test", "Test start")

OnErrorCall(@ErrorHandler())
Pokes(10, "Hello World") ; Cause a #PB_OnError_InvalidMemory error

MessageRequester("OnError test", "This should never be displayed")
```

108.3 OnErrorGoto

Syntax

```
OnErrorGoto(?LabelAddress)
```

Description

Changes the action taken when an error occurs to jump to the specified label address and continue program execution there. After the jump to the label, the functions of this library can be used to get further information about the error.

Note: The program stack will not be adjusted before jumping to the label, so local variables should not be accessed as they may not be reachable anymore. It is also not safe to continue normal program execution after an error as things like the return address of a procedure may be wrong if the stack is no longer correct. The best practise is just to gather and display information about the error and then exit the program.

The address of a label in the program to jump to.

Example:

```
MessageRequester("OnError test", "Test start")

OnErrorGoto(?ErrorHandler)
Pokes(10, "Hello World") ; Cause a #PB_OnError_InvalidMemory error

MessageRequester("OnError test", "This should never be displayed")
End

ErrorHandler:
MessageRequester("OnError test", "The following error happened: " + ErrorMessage())
End
```

108.4 OnErrorDefault

Syntax

```
OnErrorDefault()
```

Description

Changes the action taken when an error occurs back to the system default. This usually means displaying an error dialog and exiting the program, but it may also mean to just ignore certain errors. To exit the program on every error, use OnErrorExit() .

Note: When using the OnError library inside a dll the best practise is to set the error handler at the beginning of every public dll function and resetting it back to the default with this command before returning to the caller to make sure there is no interference between the OnError library and any exception handling done by the calling program.

Example:

```
Procedure ErrorHandler()
  MessageRequester("OnError test", "The following error happened: " + ErrorMessage())
EndProcedure

MessageRequester("OnError test", "Test start")

OnErrorCall(@ErrorHandler())
OnErrorDefault()          ; Comment this to get the handler call instead of the system error handling
```

```
Pokes(10, "Hello World") ; Cause a #PB_OnError_InvalidMemory error

MessageRequester("OnError test", "This should never be displayed")
```

108.5 ErrorCode

Syntax

```
Result = ErrorCode()
```

Description

Returns the error code of the currently handled error. This command only returns a meaningful value if there was an error handled by OnErrorCall() or OnErrorGoto() .

Returns one of the following values:

```
#PB_OnError_InvalidMemory          : Read or write operation on an invalid location
#PB_OnError_Floatingpoint          : Floating point error
#PB_OnError_Breakpoint             : Debugger breakpoint reached (non-PureBasic breakpoints)
#PB_OnError_IllegalInstruction     : Attempt to execute an illegal instruction
#PB_OnError_PriviledgedInstruction: Attempt to execute a priviledged (system-) instruction
#PB_OnError_DivideByZero           : Division by zero (Windows only)
```

Linux and Mac OSX report #PB_OnError_Floatingpoint for division by zero errors.

In addition, every OS may have more possible error values. On Windows, custom errors can be raised with the RaiseError() command.

108.6 ErrorMessage

Syntax

```
Result\$ = ErrorMessage([ErrorCode])
```

Description

Returns an error message for the given error code in english.

With no parameter, this function returns the message for the currently handled error. 'ErrorCode' can be specified to get the message for a specific error code.

108.7 ErrorLine

Syntax

```
Result = ErrorLine()
```

Description

Returns the line number in the sourcecode where the current error occured. This command only returns a meaningful value if there was an error handled by OnErrorCall() or OnErrorGoto() .

The tracking of line numbers needs to be enabled on compilation for this command to return the actual line number. To enable this feature, enable the "Enable OnError lines support" checkbox in the compiler options or specify the /LINENUMBERING (Windows) or –linenumbering (Linux, Mac OSX) commandline switch when compiling from the commandline .

Returns the line number of the error, or -1 if the OnError lines support is disabled.

108.8 ErrorFile

Syntax

```
Result\$ = ErrorFile()
```

Description

Returns the filename of the sourcecode or includefile where the current error occured. This command only returns a meaningful value if there was an error handled by OnErrorCall() or OnErrorGoto() .

The tracking of line numbers needs to be enabled on compilation for this command to return the actual file name. To enable this feature, enable the "Enable OnError lines support" checkbox in the compiler options or specify the /LINENUMBERING (Windows) or –linenumbering (Linux, Mac OSX) commandline switch when compiling from the commandline .

Returns the filename of the error, or "OnError line support disabled" if the OnError lines support is disabled. (A simple way to check if the OnError lines support is enabled is to check if the result of ErrorLine() is not -1.)

108.9 ErrorAddress

Syntax

```
Result = ErrorAddress()
```

Description

Returns the memory address of the assembly instruction that caused the current error. This command only returns a meaningful value if there was an error handled by OnErrorCall() or OnErrorGoto() .

108.10 ErrorTargetAddress

Syntax

```
Result = ErrorTargetAddress()
```

Description

After an error with the code #PB_OnError_InvalidMemory, this command returns the memory address which was read/written when the error occured. This command has no meaning for other error codes.

108.11 ErrorRegister

Syntax

```
Result = ErrorRegister(Register)
```

Description

Returns the content of the specified CPU register at the time of the error. This command only returns a meaningful value if there was an error handled by OnErrorCall() or OnErrorGoto() .

The available register constants depend on the CPU type for which the program is compiled. The following values are available:

x86:

```
#PB_OnError_EAX
#PB_OnError_EBX
#PB_OnError_ECX
#PB_OnError_EDX
#PB_OnError_EBP
#PB_OnError_ESI
#PB_OnError_EDI
#PB_OnError_ESP
#PB_OnError_Flags
```

x64:

```
#PB_OnError_RAX
#PB_OnError_RCX
#PB_OnError_RDX
#PB_OnError_RBX
#PB_OnError_RSP
#PB_OnError_RBP
#PB_OnError_RSI
#PB_OnError_RDI
#PB_OnError_R8
#PB_OnError_R9
...
#PB_OnError_R15
#PB_OnError_Flags
```

PowerPC:

```
#PB_OnError_r0
#PB_OnError_r1
#PB_OnError_r2
...
#PB_OnError_r31
#PB_OnError_cr
#PB_OnError_xer
#PB_OnError_lr
#PB_OnError_ctx
```

108.12 RaiseError

Syntax

```
RaiseError(ErrorNumber)
```

Description

Artificially create the given error. The appropriate error action will be taken (call of the error handler or program termination by the system if no handler is set). The ErrorNumber will be available inside the error handler with the ErrorCode() command.

On Windows, any error number can be raised (even application defined ones). On Linux or Mac OSX, only the following errors can be raised:

```
#PB_OnError_InvalidMemory          : Read or write operation
 on an invalid location
#PB_OnError_Floatingpoint          : Floating point error
#PB_OnError_Breakpoint             : Debugger breakpoint
 reached (non-PureBasic breakpoints)
#PB_OnError_IllegalInstruction     : Attempt to execute an
 illegal instruction
#PB_OnError_PriviledgedInstruction: Attempt to execute a
 priviledged (system-) instruction
```

Example:

```
Procedure ErrorHandler()
  MessageRequester("OnError test", "The following error
 happened: " + ErrorMessage())
EndProcedure

MessageRequester("OnError test", "Test start")

OnErrorCall(@ErrorHandler())
RaiseError(#PB_OnError_InvalidMemory)

MessageRequester("OnError test", "This should never be
 displayed")
```

108.13 ExamineAssembly

Syntax

```
Result = ExamineAssembly(*Address [, *EndAddress])
```

Description

Initializes the disassembling at the given address or address range.

Important: The disassembly commands use the Udis86 disassembler library to decode the instructions. This library is released under the BSD license which can be viewed here . If ExamineAssembly() and the related commands are used in a program that is to be made public, the above linked licence text must be included with the software.

*Address specifies the address of the first instruction to disassemble. If *EndAddress is specified, the disassembling will end (NextInstruction() will return 0) as soon as *EndAddress is reached. If *EndAddress is not specified, the disassembling will run until NextInstruction() is no longer called.

Returns nonzero if disassembling is possible and zero otherwise.

Example:

```
DisableDebugger ; do not disassemble any debugger related instructions

Code_Start:
  ; Place code to be disassembled here
  a = (Random(100) * 5) + 2000
Code_End:

Text$ = "Disassembled code: " + Chr(13)
If ExamineAssembly(?Code_Start, ?Code_End)
  While NextInstruction()
    Text$ + RSet(Hex(InstructionAddress()), SizeOf(Integer)*2, "0")
    Text$ + " " + InstructionString() + Chr(13)
  Wend
EndIf

MessageRequester("Result", Text$)
```

108.14 NextInstruction

Syntax

```
Result = NextInstruction()
```

Description

Disassembles the next instruction after a call to ExamineAssembly() . Information about the disassembled instruction can be read with InstructionString() and InstructionAddress() .

Returns nonzero if the instruction was successfully disassembled. Returns zero if there was no instruction to disassemble anymore. (the *EndAddress that was specified in ExamineAssembly() was reached)

108.15 InstructionAddress

Syntax

```
Result = InstructionAddress()
```

Description

Returns the address of the instruction that was disassembled by a call to NextInstruction() .

108.16 InstructionString

Syntax

```
Result\$ = InstructionString()
```

Description

Returns a string representation of the instruction that was disassembled by a call to NextInstruction() .

Chapter 109

Packer

The packer library provides an efficient set of functions to pack and unpack data (compress/decompress). The pack/unpack routines are based on the excellent JCalG1 algorithm and are integrated to PureBasic (no external DLL needed).

109.1 AddPackFile

Syntax

```
Result = AddPackFile(Filename$ [, CompressionLevel])
```

Description

Add and compress the specified file to the current Pack previously created with CreatePack() . Adding a large file can take a long time with maximum compression level, but the unpack speed is very very fast (almost instantanious). 'CompressionLevel' is an optional value which can range from 0 (fastest, least compressed) to 9 (slowest, most compressed).

109.2 AddPackMemory

Syntax

```
Result = AddPackMemory(MemoryAddress, Length [,
    CompressionLevel])
```

Description

Add and compress the specified memory area and add it to the current Pack previously created with CreatePack() . Adding a large memory area can take a long time with maximum compression level, but the unpack speed is very very fast (almost instantanious). 'CompressionLevel' is an optional value which can range from 0 (fastest, least compressed) to 9 (slowest, most compressed).

Example:

```
Dim Tableau(10) ; Table of 11 elements

; Filling the table
For i=0 To 10
```

```
    Tableau(i) = i
  Next

  If CreatePack("test.pak")
    ; Add and compresses the data of the table
    AddPackMemory(@Tableau(), SizeOf(Long)*11, 9)
    ClosePack()
  EndIf

  ; Setting the table contents to zero
  For i=0 To 10
    Tableau(i) = 0
  Next

  ; Open the packed file
  If OpenPack("test.pak")
    *Pointeur=NextPackFile() ; Read and decompresses the memory zone
    Taille=PackFileSize()    ; Get the size of the memory zone
    ; Copy the decompressed memory zone to the table
    CopyMemory(*Pointeur, @Tableau(), Taille)
    ClosePack()
  EndIf

  ; Reading the table
  For i=0 To 10
    Debug Tableau(i)
  Next
```

109.3 ClosePack

Syntax

```
Result = ClosePack()
```

Description

Closes all pack files previously opened with OpenPack() or CreatePack() and release the used memory. It should only be called when you have finished all packing/unpacking.

Note: all remaining opened packs are automatically closed when the program ends.

109.4 CreatePack

Syntax

```
Result = CreatePack(Filename$)
```

Description

Create a new empty pack file. If 'Result' equals 0, the pack file can't be created. The following functions can be used to add data to the pack file:

AddPackFile() : Add, and compress, a whole file.
AddPackMemory() : Add, and compress, a memory area.

Once all the write operations have been completed ClosePack() should be called. Note: PureBasic automatically closes any opened Packs when the program quits.

```
; Create the packed file...
CreatePack("mycompressedfiles.pak")
  ; add your files
  AddPackFile("Image1.bmp")
  AddPackFile("Image2.bmp")
  AddPackFile("mywave1.wav")
  AddPackFile("mywave2.wav")
ClosePack()
```

109.5 NextPackFile

Syntax

```
MemoryAddress = NextPackFile()
```

Description

Read and uncompress the next file found in the current Pack, previously opened with OpenPack() . The memory address is shared and should never be freed by the programmer. To know the size the memory area (ie: the size of the uncompressed file), use PackFileSize() .

109.6 PackerCallback

Syntax

```
PackerCallback(@Procedure())
```

Description

Add a callback when compressing data. As the compression is (relatively) slow, it's possible to call a procedure at regular time to follow the packing progress. The time between two calls isn't user definable, it will be calculated automatically. This is for advanced users only.

The callback must have the following form:

Procedure CompressCallback(SourcePosition, DestinationPosition)

EndProcedure

SourcePosition: Actual position, in bytes, in the source buffer (uncompressed)
DestinationPosition: Actual position, in bytes, in the destination buffer (compressed)

To continue the compression, the procedure must return 1. To cancel it, just returns 0.

109.7 OpenPack

Syntax

```
Result = OpenPack(Filename$)
```

Description

Open a pack file previously created with CreatePack() . If 'Result' equals, the pack file can't be opened. NextPackFile() is used to retrieve and unpack data from the Pack.

Once all the read operations have been done ClosePack() should be called. Note: PureBasic automatically closes any opened Packs when the program quits.

```
; Loading the packed files back to memory...
OpenPack("mycompressedfiles.pak")
  CatchSprite(0,NextPackFile())   ; InitSprite must be called before in your program...
  CatchSprite(1,NextPackFile())   ; ...first file will be loaded as Sprite 0, second as Sprite 1
  CatchSound(0,NextPackFile())    ; InitSound must be called before in your program...
  CatchSound(1,NextPackFile())    ; ...first file will be loaded as Sound 0, second as Sound 1
ClosePack()
```

109.8 PackFileSize

Syntax

```
Size = PackFileSize()
```

Description

Returns the size of the current unpacked file, after calling NextPackFile() .

109.9 PackMemory

Syntax

```
Result = PackMemory(SourceMemoryID, DestinationMemoryID,
    SourceLength [, CompressionLevel])
```

Description

Pack the content of the SourceMemory area into the DestinationMemory area. The destinatation area must be at least the length of the Source area + 8. If Result equals 0, the compression has failed (or the file is not compressible with this alogrythm), else it's the compressed size of the Destination area. The compressed data can be unpacked with UnpackMemory() . For advanced users, a callback can be added to follow the packing progress with PackerCallback() . A memory area can be easily allocated with the AllocateMemory() function. 'CompressionLevel' is an optional value which can range from 0 (fastest, least compressed) to 9 (slowest, most compressed).

Note: The compression algorithm is relatively slow, however it produces excellent results (e.g. superior to .zip format) and decompression is extremely fast (faster than .zip).

109.10 UnpackMemory

Syntax

```
Result = UnpackMemory(SourceMemoryID, DestinationMemoryID)
```

Description

Unpack a previously packed memory area (with PackMemory()) into the DestinationMemory area. If 'Result' equals 0, the data can't be unpacked (corrupted or invalid format), otherwise the length of the unpacked data is returned as 'Result'.

UnpackMemory() expect a correctly packed input, as checks are not build in it (for performance reasons). If you unpack from a non-trusted source, you should do e.g. a MD5() check first.

Note: The decompression, unlike the compression, is very fast.

Chapter 110

Palette

Palettes can be used only on screens with 256 colors (or less) and allow to 'map' the colors displayed. It's possible to assign each color (indexed from 0 to 255) to a specified RGB color, choosen from a 24 bit palette (16 Millions colors). The advantage is huge, as all the pixels from a color can be changed to another color only by changing its associated color. It allows to do very fast palette cycling, fade-in, fade-out and other well know effects.

A color table with common colors and their accompanying RGB values you will find here .

Supported OS

Windows, Linux

110.1 CreatePalette

Syntax

```
Result = CreatePalette(#Palette)
```

Description

Create a new, empty 256 colors palette. If #PB_Any is used as '#Palette' parameter, the new palette number will be returned as 'Result'. The palette is initialized with the color 0. If 'Result' is 0 the #Palette creation has failed. The previous #Palette object is automatically freed.

Supported OS

Windows, Linux

110.2 DisplayPalette

Syntax

```
DisplayPalette(#Palette)
```

Description

Display the given #Palette previously created with CreatePalette() or LoadPalette() on the current screen.

Supported OS

Windows, Linux

110.3 GetPaletteColor

Syntax

```
Color = GetPaletteColor(#Palette, Index)
```

Description

Retrieves the RGB color at the given index of the #Palette. To get the Red, Green and Blue component of the color, use the Red() , Green() and Blue() functions.

Supported OS

Windows, Linux

110.4 FreePalette

Syntax

```
FreePalette(#Palette)
```

Description

Free the memory allocated to the specified #Palette.

Note: all remaining palettes are automatically freed when the program ends.

Supported OS

Windows, Linux

110.5 InitPalette

Syntax

```
Result = InitPalette()
```

Description

Init all the Palette environments for later use. You must put this function at the top of your source code if you want to use the Palette functions. If Result is equal to 0, it has failed and none of the palette functions can be used. On Windows, if it fails, it's probably because DirectX 7 (or DirectX 3 with NT4.0 compatibility) isn't found.

Supported OS

Windows, Linux

110.6 IsPalette

Syntax

```
Result = IsPalette(#Palette)
```

Description

Tests if the given #Palette number is a valid and correctly initialized palette. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a palette is ready to use.

Supported OS

Windows, Linux

110.7 LoadPalette

Syntax

```
Result = LoadPalette(#Palette, Filename$)
```

Description

Load an palette from a standard BMP file (256 colors pictures only). If #PB_Any is used as '#Palette' parameter, the new palette number will be returned as 'Result'. The palette is initialized with the value found inside the file. If the palette has been correctly loaded, the 'Result' value is not 0, else an error has happened. The previous #Palette object is automatically freed.

Supported OS

Windows, Linux

110.8 SetPaletteColor

Syntax

```
SetPaletteColor(#Palette, Index, Color)
```

Description

Changes the RGB color at the given index of the #Palette. To get a valid RGB color value, use the RGB() function or the following formula (faster):

RGBColorValue = Red + Green « 8 + Blue « 16

Supported OS

Windows, Linux

Chapter 111

Particle

Particles systems are widely used in the 3D scenes to simulate hard to predict events, like the rain, fire, explosions etc. PureBasic allows to create any number of particle emitters as needed which are running in an autonomous manner. Each emitter has its own shape, and its own properties like the velocity, speed, emission rate and more.

InitEngine3D() should be called successfully before using any of the Particle functions.

111.1 CreateParticleEmitter

Syntax

```
Result = CreateParticleEmitter(#ParticleEmitter, Width, Height,
    Depth, Type, [, x, y, z])
```

Description

Creates a new empty #ParticleEmitter of the given width, height and depth (respectively x,y and z dimension). If #PB_Any is used as '#ParticleEmitter' parameter, the new particle emitter number will be returned as 'Result'.

The default size of the future Particles can be specified with the ParticleSize() function. The optionals x,y,z parameters can be specified to set the position of the newly created emitter in the world. If the 'Result' is 0, the #ParticleEmitter can't be created. If the #ParticleEmitter was already created, it is automatically freed and replaced by the new one.

The 'Type' parameter can be one of the following value:

```
#PB_Particle_Point: The emitter is a single point
#PB_Particle_Box  : The emitter is a box, with an width,
 height and depth
```

111.2 IsParticleEmitter

Syntax

```
Result = IsParticleEmitter(#ParticleEmitter)
```

Description

Tests if the given #ParticleEmitter is a valid and correctly initialized particle emitter.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that a particle emitter is ready to use.

111.3 ParticleEmitterID

Syntax

```
ParticleEmitterID = ParticleEmitterID(#ParticleEmitter)
```

Description

Returns the unique ID which identifies the given '#ParticleEmitter' in the operating system. This function is very useful when another library needs a particle emitter reference.

111.4 ParticleEmitterLocate

Syntax

```
ParticleEmitterLocate(#ParticleEmitter, x, y, z)
```

Description

Changes the #ParticleEmitter absolute position in the world.

111.5 ParticleEmitterX

Syntax

```
Result = ParticleEmitterX(#ParticleEmitter)
```

Description

Returns the #ParticleEmitter X absolute position in the world.

111.6 ParticleEmitterY

Syntax

```
Result = ParticleEmitterY(#ParticleEmitter)
```

Description

Returns the #ParticleEmitter Y absolute position in the world.

111.7 ParticleEmitterZ

Syntax

```
Result = ParticleEmitterZ(#ParticleEmitter)
```

Description

Returns the #ParticleEmitter Z absolute position in the world.

111.8 ParticleEmissionRate

Syntax

```
ParticleEmissionRate(#ParticleEmitter, Rate)
```

Description

Change the #ParticleEmitter emission rate, in particles per second.

111.9 ParticleMaterial

Syntax

```
ParticleMaterial(#ParticleEmitter, MaterialID)
```

Description

Assigns a material to the specified #ParticleEmitter. This material will be used by all the Particles of this emitter. An emitter can only have one material assigned at once. A valid 'MaterialID' can be easily obtained with the MaterialID() function.

111.10 ParticleTimeToLive

Syntax

```
ParticleTimeToLive(#ParticleEmitter, MininumTime, MaximumTime)
```

Description

Change the particles time to live for the #ParticleEmitter. Every particle will live at least 'MinimumTime' and at most the 'MaximumTime' (in world time unit). A random value, within this range, will be used by every particle emitted.

111.11 ParticleVelocity

Syntax

```
ParticleVelocity(#ParticleEmitter, Minimum, Maximum)
```

Description

Changes the particles velocity. A random velocity value, picked in the 'Minimum' and 'Maximum' range, will be used by every particle emitted.

111.12 ParticleSize

Syntax

```
ParticleSize(#ParticleEmitter, Width, Height)
```

Description

Change the particles dimensions. Particles are 2D planes (billboards) which always face the camera. All particles in an emitter always have the same size.

111.13 ParticleColorRange

Syntax

```
ParticleColorRange(#ParticleEmitter, StartColor, EndColor)
```

Description

Changes the particles color range for the #ParticleEmitter. Every emitted particle will get a random value, within the 'StartColor' and 'EndColor' range. The 'StartColor' and 'EndColor' parameters are regular RGB values. RGB() can be used to get a valid color value.

111.14 ParticleColorFader

Syntax

```
ParticleColorFader(#ParticleEmitter, RedRate, GreenRate,
   BlueRate)
```

Description

Changes the particles color fader rate (In world time unit).

111.15 FreeParticleEmitter

Syntax

```
FreeParticleEmitter(#ParticleEmitter)
```

Description

Free the specified #ParticleEmitter. All its associated memory is released and this object can't be used anymore.

Note: all remaining particle emitters are automatically freed when the program ends.

111.16 HideParticleEmitter

Syntax

```
HideParticleEmitter(#ParticleEmitter, State)
```

Description

Hides or Shows the specified #ParticleEmitter. 'State' can take the following values:

```
  1: the #ParticleEmitter is hidden
  0: the #ParticleEmitter is shown
```

111.17 MoveParticleEmitter

Syntax

```
MoveParticleEmitter(#ParticleEmitter, x, y, z)
```

Description

Moves the #ParticleEmitter in the world according to the specified x,y,z values. This is a relative move based on the previous #ParticleEmitter location. To perform an absolute move in the world, use ParticleEmitterLocate() .

111.18 ParticleEmitterDirection

Syntax

```
ParticleEmitterDirection(#ParticleEmitter, x, y, z)
```

Description

Changes the direction of the #ParticleEmitter according to the specified x,y,z values.

111.19 ResizeParticleEmitter

Syntax

```
ResizeParticleEmitter(#Particle, #ParticleEmitter, Width,
    Height, Depth)
```

Description

Resizes the #Particle to the new specified 'Width', 'Height' and 'Depth' dimension, in world metrics.

Chapter 112

Preference

Preference file contains user defined program parameters stored on the disk and read back when the program start again (like '.INI' files under Windows for example). PureBasic offers the possibility to create heriarchical preference file easily useable on any computer plateforms. The file format is in ascii (standard) with one preference per line, using the 'Keyword = Value' syntax. Groups can be created for better reading.

Compiling in unicode modus forces PureBasic to save the strings in the preference file in UTF-8 format .

112.1 ClosePreferences

Syntax

```
ClosePreferences()
```

Description

Close a preference file previously opened with OpenPreferences() or created with CreatePreferences() .

112.2 CreatePreferences

Syntax

```
Result = CreatePreferences(Filename$)
```

Description

Creates a new empty preference file. If the file already exists, the file is erased. If 'Result' is 0 then the file can't be created. Once the file has been created successfully, the functions like WritePreferenceString() can be used to write data. Once some values are written in preferences, it's possible to read them back with the functions like ReadPreferenceString() . To remove a key or a group, the functions RemovePreferenceKey() and RemovePreferenceGroup() are available. To create or change the current group, use PreferenceGroup() .

Once all write operations have been done, ClosePreferences() is used to write effectively the preference file back to disk.

112.3 ExaminePreferenceGroups

Syntax

```
Result = ExaminePreferenceGroups()
```

Description

Starts the enumeration of all the groups found in the current preference file. The function NextPreferenceGroup() is used to list all the group found. If the function can't start the enumeration, the 'Result' is 0.

112.4 ExaminePreferenceKeys

Syntax

```
Result = ExaminePreferenceKeys()
```

Description

Starts the enumeration of all the keys found in the current group of the preference file. The current group can be selected with PreferenceGroup() or by examining all groups with ExaminePreferenceGroups() . The function NextPreferenceKey() is used to list all the keys found. If the function can't start the enumeration, the 'Result' is 0.

112.5 NextPreferenceGroup

Syntax

```
Result = NextPreferenceGroup()
```

Description

Retrieves information about the next group found in the enumeration started with ExaminePreferenceGroups() . To get the name of the group, use PreferenceGroupName() . The current examined preference group will also be used when values are read from the preferences or when the keys are examined with ExaminePreferenceKeys() . If 'Result' is 0, then no more groups are found.

112.6 NextPreferenceKey

Syntax

```
Result = NextPreferenceKey()
```

Description

Retrieves information about the next key found in the enumeration started with ExaminePreferenceKeys() . To get the name and the value of the key, use PreferenceKeyName() and PreferenceKeyValue() . If 'Result' is 0, then no more keys are found.

112.7 PreferenceGroupName

Syntax

```
Group\$ = PreferenceGroupName()
```

Description

Returns the name of the current group being enumerated with ExaminePreferenceGroups() .

112.8 PreferenceKeyName

Syntax

```
Key\$ = PreferenceKeyName()
```

Description

Returns the name of the current key being enumerated with ExaminePreferenceKeys() . To get the value of the key, use PreferenceKeyValue() .

112.9 PreferenceKeyValue

Syntax

```
Value\$ = PreferenceKeyValue()
```

Description

Returns the value, in string form, of the current key being enumerated with ExaminePreferenceKeys() . To get the name of the key, use PreferenceKeyName() .

112.10 OpenPreferences

Syntax

```
Result = OpenPreferences(Filename$)
```

Description

Open a previously existing preference file. If 'Result' is 0 then the file can't be opened or wasn't found. Nevertheless, if the file wasn't found, the programmer can still use the the read functions and it will return always the specified default value, very useful to initialize in one step the program variables. The functions like ReadPreferenceString() can be used to read the preference values stored in the file.

To remove a key or a group, the functions RemovePreferenceKey() and RemovePreferenceGroup() are available. To create or change the current group, use PreferenceGroup() . It is possible to change existing values with WritePreferenceString() and similar functions.

Once all operations have been done, ClosePreferences() is used to write effectively the preference file back to disk (if it has been modified in between).

112.11 PreferenceGroup

Syntax

```
Result = PreferenceGroup(Name$)
```

Description

Creates a new group (in the form: [Name$]) or change the current group in the preference file. All following read or write operations will be restricted to this group. To move outside of any groups, an empty 'Name$' can be used. The function returns 1 if the group allready existed and 0 if not.

If the group does not exist yet, it will not be created immediately. It will instead be created once the first key is written to it. This allows to use the PreferenceGroup() function to test if a group exists without creating many empty groups.

112.12 PreferenceComment

Syntax

```
PreferenceComment(Text$)
```

Description

Writes a new comment line in the current preference file.

112.13 ReadPreferenceDouble

Syntax

```
Result.d = ReadPreferenceDouble(Keyword$, DefaultValue)
```

Description

Try to read the specified associated 'Keyword$' value. If PreferenceGroup() has been used, then the search is restricted to the current group. If the 'Keyword$' isn't found or the preference file haven't been opened correctly (file missing for example) the specified default value is returned.

112.14 ReadPreferenceFloat

Syntax

```
Result.f = ReadPreferenceFloat(Keyword$, DefaultValue)
```

Description

Try to read the specified associated 'Keyword$' value. If PreferenceGroup() has been used, then the search is restricted to the current group. If the 'Keyword$' isn't found or the preference file haven't been opened correctly (file missing for example) the specified default value is returned.

112.15 ReadPreferenceInteger

Syntax

```
Result = ReadPreferenceInteger(Keyword$, DefaultValue)
```

Description

Try to read the specified associated 'Keyword$' value. If PreferenceGroup() has been used, then the search is restricted to the current group. If the 'Keyword$' isn't found or the preference file haven't been opened correctly (file missing for example) the specified default value is returned.

112.16 ReadPreferenceLong

Syntax

```
Result = ReadPreferenceLong(Keyword$, DefaultValue)
```

Description

Try to read the specified associated 'Keyword$' value. If PreferenceGroup() has been used, then the search is restricted to the current group. If the 'Keyword$' isn't found or the preference file haven't been opened correctly (file missing for example) the specified default value is returned.

112.17 ReadPreferenceQuad

Syntax

```
Result.q = ReadPreferenceQuad(Keyword$, DefaultValue)
```

Description

Try to read the specified associated 'Keyword$' value. If PreferenceGroup() has been used, then the search is restricted to the current group. If the 'Keyword$' isn't found or the preference file haven't been opened correctly (file missing for example) the specified default value is returned.

112.18 ReadPreferenceString

Syntax

```
Result\$ = ReadPreferenceString(Keyword$, DefaultValue)
```

Description

Try to read the specified associated 'Keyword$' value. If PreferenceGroup() has been used, then the search is restricted to the current group. If the 'Keyword$' isn't found or the preference file haven't been opened correctly (file missing for example) the specified default value is returned.

To read keyword values directly as variable types the following functions can be used:

ReadPreferenceDouble()
ReadPreferenceFloat()
ReadPreferenceLong()
ReadPreferenceQuad()

112.19 RemovePreferenceGroup

Syntax

```
RemovePreferenceGroup(Group$)
```

Description

Removes the specified 'Group$' and all its keys.

112.20 RemovePreferenceKey

Syntax

```
RemovePreferenceKey(Key$)
```

Description

Removes the specified 'Key$' and its value.

112.21 WritePreferenceFloat

Syntax

```
WritePreferenceFloat(Keyword$, Value.f)
```

Description

Writes the specified 'Keyword$' and its associated float value under the form: 'Keyword$ = Value' in the preference file, previously created with CreatePreferences() or opened with OpenPreferences() . If the key was already existing in the current group, its value is replaced.

112.22 WritePreferenceDouble

Syntax

```
WritePreferenceDouble(Keyword$, Value.d)
```

Description

Writes the specified 'Keyword$' and its associated double value under the form: 'Keyword$ = Value' in the preference file, previously created with CreatePreferences() or opened with OpenPreferences() . If the key was already existing in the current group, its value is replaced.

112.23 WritePreferenceInteger

Syntax

```
WritePreferenceInteger(Keyword$, Value)
```

Description

Writes the specified 'Keyword$' and its associated value under the form: 'Keyword$ = Value' in the preference file, previously created with CreatePreferences() or opened with OpenPreferences() . If the key was already existing in the current group, its value is replaced.

112.24 WritePreferenceLong

Syntax

```
WritePreferenceLong(Keyword$, Value)
```

Description

Writes the specified 'Keyword$' and its associated value under the form: 'Keyword$ = Value' in the preference file, previously created with CreatePreferences() or opened with OpenPreferences() . If the key was already existing in the current group, its value is replaced.

112.25 WritePreferenceQuad

Syntax

```
WritePreferenceQuad(Keyword$, Value.q)
```

Description

Writes the specified 'Keyword$' and its associated quad value under the form: 'Keyword$ = Value' in the preference file, previously created with CreatePreferences() or opened with OpenPreferences() . If the key was already existing in the current group, its value is replaced.

112.26 WritePreferenceString

Syntax

```
WritePreferenceString(Keyword$, Value$)
```

Description

Writes the specified 'Keyword$' and its associated string value under the form: 'Keyword$ = Value' in the preference file, previously created with CreatePreferences() or opened with OpenPreferences() . If the key was already existing in the current group, its value is replaced.

To write keyword values directly from numeric variable types the following functions can be used:

WritePreferenceDouble()
WritePreferenceFloat()
WritePreferenceLong()
WritePreferenceQuad()

Chapter 113

Printer

Printers are essential devices to transforms virtual, numeric data to sheet. Many softwares requiere a way to print data back to sheet to be really useful. Purebasic allows to print any kind of data from raw texts to pictures, in any resolution.

113.1 DefaultPrinter

Syntax

```
Result = DefaultPrinter()
```

Description

Selects the default printer as current printer for print operation. This function must be called before all other printer functions. If 'Result' is 0, then no default printer is available. Once DefaultPrinter() has been successfully called, StartPrinting() is used to actually start to print.

113.2 NewPrinterPage

Syntax

```
NewPrinterPage()
```

Description

Create a new empty page. The previous page is sent to the printer and can't be modified anymore. It has to be called inside a StartDrawing() /StopDrawing() block.

Example:

```
If PrintRequester()
  If StartPrinting("Two sheets")
    If StartDrawing(PrinterOutput())
      DrawingMode(#PB_2DDrawing_Transparent)
      DrawText(10, 10, "First page !", RGB(0, 0, 0))

      ; Tell the printer to start a new page
      NewPrinterPage()

```

```
        DrawText(10, 10, "Second page !", RGB(0, 0, 0))

        StopDrawing()
      EndIf

      StopPrinting()
    EndIf
  EndIf
```

113.3 PrinterOutput

Syntax

```
OutputID = PrinterOutput()
```

Description

Returns the OutputID of the current printer to be used with the StartDrawing() function. All the 2DDrawing functions are available and can be used for printing. The printer is saw like a empty sheet, exactly like a window or an image.

113.4 PrintRequester

Syntax

```
Result = PrintRequester()
```

Description

Open a regular print requester, to choose the printer and doing some adjustments. This function must be called before all other printer functions. If 'Result' is 0, the no printer are available or the user has cancelled the requester. Once PrintRequester() has been successfully called, StartPrinting() is used to actually start to print.

113.5 StartPrinting

Syntax

```
Result = StartPrinting(JobName$)
```

Description

Initialize the printer and start the print operation. The JobName$ is the name which will be shown in the printer spooler and which identify the printing. If 'Result' is 0, then the printing can't be achieved.

113.6 StopPrinting

Syntax

```
StopPrinting()
```

Description

Stop all the print operations and send the data to the printer.

113.7 PrinterPageWidth

Syntax

```
Result = PrinterPageWidth()
```

Description

Return the witdh of the drawing area, in pixel. The number changes with the DPI of the printer. This means than a document printed at 75 DPI will have a drawing area 4 times smaller than a document printed at 150 DPI.

Note: The return value refer to the client area, the one you can write on. It doesn't count the hardware printer margins, but doesn't add any 'soft' margins.

113.8 PrinterPageHeight

Syntax

```
Result = PrinterPageHeight()
```

Description

Return the height of the drawing area, in pixel. The number changes with the DPI of the printer. This means than a document printed at 75 DPI will have a drawing area 4 times smaller than a document printed at 150 DPI.

Note: The return value refer to the client area, the one you can write on. It doesn't count the hardware printer margins, but doesn't add any 'soft' margins.

Chapter 114

Process

The process library allows the programmer the ability to get information about the current program, as well as to execute other programs and communicate with them. This provides a platform for independent access to a programs environment block, the programs parameters, as well as standard input and output.

114.1 AvailableProgramOutput

Syntax

```
Result = AvailableProgramOutput(Program)
```

Description

Returns the number of bytes that are available to be read from the programs output. The program must have been started with RunProgram() using the #PB_Program_Read flag.

The output may be read with either ReadProgramString() or ReadProgramData() .

114.2 CloseProgram

Syntax

```
CloseProgram(Program)
```

Description

Closes the connection with the given program (which was started with RunProgram()) and frees all related data.

Note: This does not terminate the program, it only closes the connection with it. To terminate the program, call KillProgram() first. Also if the program terminated normally, this function must still be called to properly release all data.

If the program was started with the #PB_Program_Write flag, CloseProgram() will cause the program to receive an EOF (end of file) on its standard input. This condition can also be produced without directly closing the connection to the program by calling WriteProgramData() with the special #PB_Program_Eof value.

114.3 CountProgramParameters

Syntax

```
Result = CountProgramParameters()
```

Description

Returns the number of parameters that were passed to this program on the command line (or via RunProgram()).

ProgramParameter() may be used to read the individual parameters.

114.4 EnvironmentVariableName

Syntax

```
Result\$ = EnvironmentVariableName()
```

Description

This command must be called after ExamineEnvironmentVariables() and NextEnvironmentVariable() .

It returns the name of the current environment variable. EnvironmentVariableValue() may be used to return its value.

114.5 EnvironmentVariableValue

Syntax

```
Result\$ = EnvironmentVariableValue()
```

Description

This function must be called after ExamineEnvironmentVariables() and NextEnvironmentVariable() .

It returns the value of the current environment variable. EnvironmentVariableName() may be used to return its name.

114.6 ExamineEnvironmentVariables

Syntax

```
Result = ExamineEnvironmentVariables()
```

Description

Starts to examine the environment block of the program. If the environment block cannot be read, the result is zero.

NextEnvironmentVariable() , EnvironmentVariableName() and EnvironmentVariableValue() may be used to read the individual environment variables.

Example:

```
; Will output all environment variables of the program
;
OpenConsole()
If ExamineEnvironmentVariables()
  While NextEnvironmentVariable()
    PrintN(EnvironmentVariableName() + " = " +
 EnvironmentVariableValue())
  Wend
EndIf
PrintN("")
PrintN("Press Enter to quit.")
Input()
```

114.7 GetEnvironmentVariable

Syntax

```
Result\$ = GetEnvironmentVariable(Name$)
```

Description

Returns the content of the specified environment variable from the programs environment block. If the variable does not exist, an empty string is returned.

Example:

```
; Prints the content of the "PATH" environment variable
;
OpenConsole()
PrintN(GetEnvironmentVariable("PATH"))
PrintN("")
PrintN("Press Enter to quit.")
Input()
```

114.8 IsProgram

Syntax

```
Result = IsProgram(Program)
```

Description

Tests if the given 'Program' number is a program that was executed by the RunProgram() function. This function is bullet-proof and may be used with any value. If the 'Result' is not zero then the program is valid, otherwise zero is returned.

114.9 KillProgram

Syntax

```
KillProgram(Program)
```

Description

Immediately terminates the given program (which was previously started with RunProgram()).

Note: This terminates the program, but it does not close the connection with the program object of the terminated program. CloseProgram() must still be called to properly free all data associated with the program.

114.10 NextEnvironmentVariable

Syntax

```
Result = NextEnvironmentVariable()
```

Description

This function must be called after ExamineEnvironmentVariables() . It will go step by step through the environment variables of the program. It will return zero as soon as there are no more variables to read.

Use EnvironmentVariableName() and EnvironmentVariableValue() to get the name and content of the current variable.

114.11 ProgramExitCode

Syntax

```
Result = ProgramExitCode(Program)
```

Description

Returns the exitcode that was returned when the given program ended. The given 'Program' must have been stated before with the RunProgram() function.

This function should only be used after the given program has ended. Use ProgramRunning() or WaitProgram() to make sure of that.

The exitcode allows for a single value to be returned from the program, back to the program that executed it. This is usually used to indicate either the error or the success of the program.

To return an exitcode from a PB program, use the optional value with the End statement:

```
1   End 1    ; returns the exitcode 1
```

114.12 ProgramFilename

Syntax

```
Result\$ = ProgramFilename()
```

Description

Returns the full path and filename of this program. It may be used to find out where the program was installed to or how the executable was named. GetPathPart() or GetFilePart() may be used to get only the path or filename of the program from the return string.

If used inside a Dll, this function returns the path and filename of the dll, not the main program that loaded it.

114.13 ProgramID

Syntax

```
Result = ProgramID(Program)
```

Description

Returns the unique system identifier for the given Program (which was started earilier with RunProgram()). This is the so called "Process ID" or "PID".

If the identifier cannot be returned, -1 will be returned.
This will happen if RunProgram() is used to open a file in another program, like RunProgram("Source.pb").

Note: the value returned here is not a handle (unlike with most other ...ID() functions). Rather, it is the Process ID which may be viewed within the Taskmanager. To get a process handle, use the OpenProcess_() Api.

114.14 ProgramParameter

Syntax

```
Result\$ = ProgramParameter([Index])
```

Description

Gets the next parameter string that was passed to the executable when it was launched or, if the 'Index' parameter (starting with 0) is specified, the parameter with that index. When no more parameters are available, an empty string is returned.

This function is especially useful for console programs , where the user passes one or more parameter at the program start.

Note: Relying on the return of an empty string to detect the last parameter is not a good practice, since the function also returns an empty string if an empty string was passed in "" on the commandline. The prefered method which should be used to get all parameters, is to get the count with CountProgramParameters() and then to call ProgramParameter() as often as needed.

Example:

```
MyProgram.exe MyText.txt /FAST "Special Mode"
```

The first time that ProgramParameter() is called, it will return "MyText.txt", the second time "/FAST" and the third time "Special Mode".

114.15 ProgramRunning

Syntax

```
Result = ProgramRunning(Program)
```

Description

Tests if the given program (started previously with RunProgram()) is still running. It returns nonzero as long as the program has not yet ended.

114.16 ReadProgramData

Syntax

```
Result = ReadProgramData(Program, *Buffer, size)
```

Description

Reads data from the given programs output (stdout) and puts it in the specified buffer (a memory area will be reserved by using AllocateMemory()).

The program must have been started with RunProgram() and the #PB_Program_Read flag.

The function reads up to 'size' bytes, but also less if not that much is available (but it will not return zero bytes because it always waits for some data to read). Check the returnvalue to know how much data was actually read.

Note: This function waits until there is data available to read from the program. To prevent this wait, AvailableProgramOutput() may be used first to check if there is something to read.

Returns the number of bytes that were actually read.

114.17 ReadProgramError

Syntax

```
Result\$ = ReadProgramError(Program)
```

Description

Reads a line from the specified programs error output (stderr). The program must have been started with RunProgram() and the #PB_Program_Error flag.

This function does not wait for data to be available. If there is no error output to read, the function will return an empty string.

114.18 ReadProgramString

Syntax

```
Result\$ = ReadProgramString(Program)
```

Description

Reads a line from the output (stdout) of the given program. The program must have been started with RunProgram() and the #PB_Program_Read flag.

Note: This function waits until there is data available to read from the program. To prevent this wait, AvailableProgramOutput() may be used first to check if there is something to read. This function also waits until a full line of output is available. If not line-based or raw output is to be read, ReadProgramData() may be used.

114.19 RemoveEnvironmentVariable

Syntax

```
RemoveEnvironmentVariable(Name$)
```

Description

Removes the given environment variable from the programs environment block.

114.20 RunProgram

Syntax

```
Result = RunProgram(Filename$ [, Parameter$, WorkingDirectory$
    [, Flags [, SenderProgram]]])
```

Description

Launches an external program. The 'Filename$' should be the executable including its path. 'Parameters$' can be used to specify command line parameters that will be passed to the program. 'WorkingDirectory$' may be used to specify a directory that will then be the current directory for the external program.

Flags are optional and may be a combination (using '||' OR operator) of the following values:

```
#PB_Program_Wait    : Wait until the launched program quits
#PB_Program_Hide    : Launch the program in invisible mode
#PB_Program_Open    : Open the program to communicate with it
 or get information about it.
#PB_Program_Read    : Read the programs console output.
 (stdout)
#PB_Program_Write   : Write to the input of the program.
 (stdin)
#PB_Program_Error   : Read the error output of the program.
 (stderr)
#PB_Program_Connect: Connect another programs output to this
 programs input.
```

A program executed with #PB_Program_Open must be closed with CloseProgram() . The 'Read', 'Write', 'Error' and 'Connect' flags require the #PB_Program_Open flag to be set as well.

When using the #PB_Program_Connect flag, another program must have been started before with #PB_Program_Open and #PB_Program_Read. The number returned when running this program must be passed as the 'SenderProgram' parameter to RunProgram(). The output of the sender program will be sent directly to the input of the now executed program. Several programs may be connected in this way, to 'pipe' data through that group of connected programs.

The following functions may be used with a program that has been executed with the #PB_Program_Open flag:

- IsProgram() : check if the program number represents a valid program executed by RunProgram().
- ProgramID() : returns the OS ProcessID for the program.
- ProgramRunning() : check if the program is still running.
- WaitProgram() : wait for the program to end.
- KillProgram() : terminate the program.
- ProgramExitCode() : get the exitcode of the program.

- CloseProgram() : close the connection to the program.

The following functions may be used for programs executed with the 'Read', 'Write' and 'Error' flags:

- AvailableProgramOutput() : check if the programs output is available.
- ReadProgramString() : read a string from the programs output.
- ReadProgramData() : read data from the programs output.
- ReadProgramError() : read a string from the programs error output.
- WriteProgramString() : write a string to the programs input.
- WriteProgramData() : write data to the programs input.

If the program cannot be executed, the returnvalue is zero. If RunProgram() was successful, the returnvalue is nonzero.

If #PB_Program_Open was included in the Flags, the returnvalue is a number that identifies the program. It may be used in calls to get information about the program such as ReadProgramString() or ProgramExitCode() or other functions mentioned above.

Example:

```
; Executes the PB compiler with the /? option and displays the output (windows version)
; For Linux/MacOS change the "/?" to "-h".
;
Compiler = RunProgram(#PB_Compiler_Home+"/Compilers/pbcompiler", "/?", "", #PB_Program_Open|#PB_Program_Read)
Output$ = ""
If Compiler
  While ProgramRunning(Compiler)
    If AvailableProgramOutput(Compiler)
      Output$ + ReadProgramString(Compiler) + Chr(13)
    EndIf
  Wend
  Output$ + Chr(13) + Chr(13)
  Output$ + "Exitcode: " + Str(ProgramExitCode(Compiler))

  CloseProgram(Compiler) ; Close the connection to the program
EndIf

MessageRequester("Output", Output$)
```

Under Windows RunProgram() uses the default application associated with the file type of the given file. An example: RunProgram("Test.html") will open the web browser, which is generally used for displaying web sites on your system.

114.21 SetEnvironmentVariable

Syntax

```
SetEnvironmentVariable(Name$, Value$)
```

Description

Creates an environment variable in the environment block of this program with given name and value. If a variable with this name already existed, its content will be changed to the new value.

The environment block of the program is passed on to other programs executed with RunProgram() , so this method may be used to pass on information to a program executed by this program. (The executed program may use GetEnvironmentVariable() to read the variables.)

114.22 WaitProgram

Syntax

```
Result = WaitProgram(Program [, Timeout])
```

Description

Halts the execution of the code until the specified program (started before with RunProgram()) has ended or the optional timeout (in milliseconds) is reached.

Returns nonzero if the program has ended and zero if the timeout was reached.

114.23 WriteProgramData

Syntax

```
Result = WriteProgramData(Program, *Buffer, size)
```

Description

Writes the data from the buffer to the specified programs input (stdin). The program must have been started with RunProgram() and the #PB_Program_Write flag.

A special value of #PB_Program_Eof may be used for the *Buffer parameter to make the program receive an EOF (end of file) in its input. (Which tells the program that there will be no more input). The 'Size' parameter is ignored in this case. After calling WriteProgramData() with this special value, no more input may be written to the program.

Returns the number of bytes that were actually written.

114.24 WriteProgramString

Syntax

```
Result = WriteProgramString(Program, String$)
```

Description

Writes the given string to the specified programs input (stdin). The program must have been started with RunProgram() and the #PB_Program_Write flag.

There is no newline written to the program after the string. To include the newline, WriteProgramStringN() may be used. To write raw data to the program, WriteProgramData() may be used. This function may also be used to send an EOF (end of file) to the program which tells the program that there will be no more input.

114.25 WriteProgramStringN

Syntax

```
Result = WriteProgramStringN(Program, String$)
```

Description

Writes the given string to the specified programs input (stdin). The program must have been started with RunProgram() and the #PB_Program_Write flag.

For sending a string without newline WriteProgramString() may be used, for sending raw data WriteProgramData() .

Chapter 115

RegularExpression

Regular expressions allow to do advanced pattern matching to quickly match, extract or replace an arbitrary information in a string. These kind of expressions are often difficult to read an write, but once you masterize them it makes a lot of things easier. Therefore this library is not for beginners, and you need to have some solid basis with PureBasic and programming in general to be able to use this library efficiently.

This library uses PCRE, which is an opensource implementation of the Perl regular expression. All the regular expressions supported in PCRE will be supported in PureBasic. To have a complete list of supported pattern and arguments, please visit the PCRE page: http://www.pcre.org/pcre.txt
The PCRE license can be viewed here .

Important: The PCRE license requires that a copyright notice and the license text itself be included in any software that uses the library. So if the RegularExpression library is used in software that is to be made public, the above linked license must be included with the software.

115.1 CreateRegularExpression

Syntax

```
Result = CreateRegularExpression(#RegularExpression, Pattern$
    [, Flags])
```

Description

Create a new regular expression. If #PB_Any is used as '#RegularExpression' parameter, the new regular expression number will be returned as 'Result'. 'Pattern$' is the regular expression which will be applied to the string to match, extract or replace string information. If an error has been detected in the pattern, 'Result' will be 0. To get more information about the error, see RegularExpressionError() .

If a regular expression isn't used anymore, use FreeRegularExpression() .

'Flags' can be a combination of the following values:

```
#PB_RegularExpression_DotAll     : '.' matches anything
 including newlines.
#PB_RegularExpression_Extended   : whitespace and '#' comments
 will be ignored.
#PB_RegularExpression_MultiLine  : '^' and '$' match newlines
 within data.
#PB_RegularExpression_AnyNewLine: recognize 'CR', 'LF', and
 'CRLF' as newline sequences.
```

Example:

```
; This expression will match every word of 3 letter which begin by a lower case letter,
; followed with the character 'b' and which ends with an uppercase letter. ex: abC
;
If CreateRegularExpression(0, "[a-z]b[A-Z]")
  Debug MatchRegularExpression(0, "abC") ; Will be 1
  Debug MatchRegularExpression(0, "abc") ; Will be 0
Else
  Debug RegularExpressionError()
EndIf
```

115.2 ExtractRegularExpression

Syntax

```
Result = ExtractRegularExpression(#RegularExpression, String$, Array$())
```

Description

Extract strings according to the #RegularExpression into an array. The array is automatically resized to the number of element matching the expression found in the 'String$'. The number element found is return in 'Result'.

Example:

```
; This expression will match every word of 3 letter which begin by a lower case letter,
; followed with the character 'b' and which ends with an uppercase letter. ex: abC
;
If CreateRegularExpression(0, "[a-z]b[A-Z]")
  Dim Result$(0)
  NbFound = ExtractRegularExpression(0, "abC ABc zbA abc", Result$())
  For k = 0 To NbFound-1
    Debug Result$(k)
  Next
Else
  Debug RegularExpressionError()
EndIf
```

115.3 FreeRegularExpression

Syntax

```
FreeRegularExpression(#RegularExpression)
```

Description

Free the specified #RegularExpression and release its associated memory.

Note: all remaining regular expressions are automatically freed when the program ends.

115.4 IsRegularExpression

Syntax

```
Result = IsRegularExpression(#RegularExpression)
```

Description

Tests if the given #RegularExpression number is a valid and correctly initialized regular expression. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a regular expression is ready to use.

115.5 MatchRegularExpression

Syntax

```
Result = MatchRegularExpression(#RegularExpression, String$)
```

Description

Test the string against the #RegularExpression. 'Result' is will be 1 if there is one or more matches, or 0 if no pattern is found.

Example:

```
; This expression will match every word of 3 letter which begin by a lower case letter,
; followed with the character 'b' and which ends with an uppercase letter. ex: abC
;
If CreateRegularExpression(0, "[a-z]b[A-Z]")
  If MatchRegularExpression(0, "abC ABc zbA abc")
    Debug "The string match !"
  Else
    Debug "No pattern found in the string"
  EndIf
Else
  Debug RegularExpressionError()
EndIf
```

115.6 ReplaceRegularExpression

Syntax

```
Result\$ = ReplaceRegularExpression(#RegularExpression,
    String$, ReplaceString$)
```

Description

Replace some strings according to the #RegularExpression. Each pattern matching the regular expression will be replaced with 'ReplaceString$'.

Example:

```
; This expression will match every word of 3 letter which begin by a lower case letter,
; followed with the character 'b' and which ends with an uppercase letter. ex: abC
;
If CreateRegularExpression(0, "[a-z]b[A-Z]")
  Result$ = ReplaceRegularExpression(0, "abC ABc zbA abc", "---")
  Debug Result$ ; Will print "--- ABc --- abc"
Else
  Debug RegularExpressionError()
EndIf
```

115.7 RegularExpressionError

Syntax

```
Result\$ = RegularExpressionError()
```

Description

Return an human readable error (in english) about the lastest failure of CreateRegularExpression() .

Example:

```
; Here we put an extra bracked '[', so there is a syntax error in the regular expression
;
If CreateRegularExpression(0, "[a-z]b[[A-Z][]")
  Debug "Success"
Else
  Debug RegularExpressionError()
EndIf
```

Chapter 116

Requester

Computer users are well used to requesters as almost any graphical application use at least one of them. They are very handy as some basic tasks (like opening a file, choose a color, choose a font etc...) are all performed through standard windows called 'requesters'.

116.1 ColorRequester

Syntax

```
Color = ColorRequester([CurrentColor])
```

Description

Opens the standard requester to choose a color. The chosen color is returned under a 24 bit number containing the red, green and blue value, as usual. To easily get each RGB component value, Red() , Green() and Blue() can be used. If the user cancel the requester, -1 is returned. The optional 'CurrentColor' parameter can be used to set the default color when the requester is opened.

Example:

```
Color.l = ColorRequester()
If Color > -1
  a$ = "You have selected following color value:" + Chr(10)
 ; Chr(10) only needed
  a$ + "24 Bit value: " + Str(Color)              + Chr(10)
 ; for line-feed
  a$ + "Red value:    " + Str(Red(Color))         + Chr(10)
  a$ + "Green value:  " + Str(Green(Color))       + Chr(10)
  a$ + "Blue value:   " + Str(Blue(Color))
Else
  a$ = "The requester was canceled."
EndIf
MessageRequester("Information", a$, 0)
End
```

116.2 FontRequester

Syntax

```
Result = FontRequester(FontName$, FontSize, Flags [, Color [,
    Style]])
```

Description

Opens the standard requester to choose a font. If the user cancels the requester, the return value is 0, else it's 1. 'FontName$' and 'FontSize' can be used to set the default font.

The functions SelectedFontColor() , SelectedFontName() , SelectedFontSize() and SelectedFontStyle() can be used after a successful call to get the needed information about the selected font.

The optional 'Color' parameter can be used to set the default color when the requester is opened. A valid color value you can get with the function RGB() .

The optional 'Style' parameter can be used to set the default style when the requester is opened. See SelectedFontStyle() for available styles.

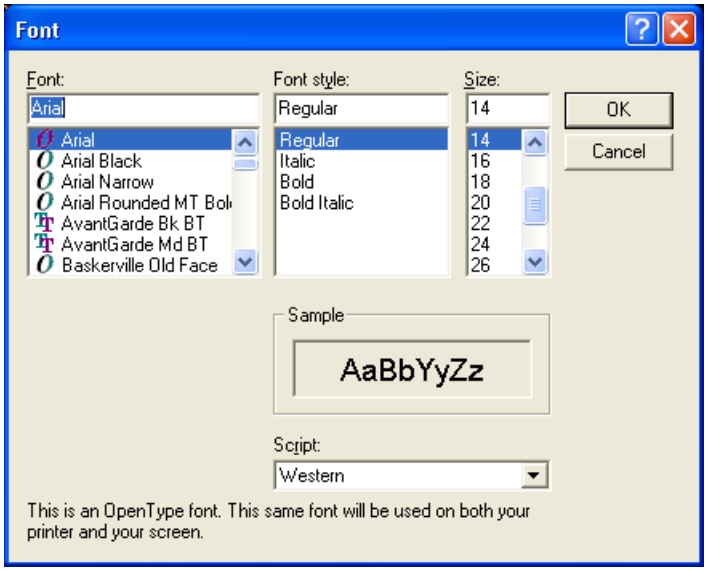
'Flags' can be one of the following values:

```
#PB_FontRequester_Effects : Enable the requesters effects,
 including color chooser.
```

Example:

```
FontName$ = "Arial"    ; set initial font  (could also be
 blank)
FontSize  = 14         ; set initial size  (could also be null)
Result = FontRequester(FontName$, FontSize,
 #PB_FontRequester_Effects)
If Result
  Message$ = "You have selected following font:"  + #LF$
  Message$ + "Name:   " + SelectedFontName()       + #LF$
  Message$ + "Size:   " + Str(SelectedFontSize())  + #LF$
  Message$ + "Color:  " + Str(SelectedFontColor()) + #LF$
  If SelectedFontStyle() & #PB_Font_Bold
    Message$ + "Bold" + #LF$
  EndIf
  If SelectedFontStyle() & #PB_Font_StrikeOut
    Message$ + "StrikeOut" + #LF$
```

```
    EndIf
    If SelectedFontStyle() & #PB_Font_Underline
      Message$ + "Underline" + #LF$
    EndIf
  Else
    Message$ = "The requester was canceled."
  EndIf

  MessageRequester("FontRequester", Message$,
   #PB_MessageRequester_Ok)
```



116.3 InputRequester

Syntax

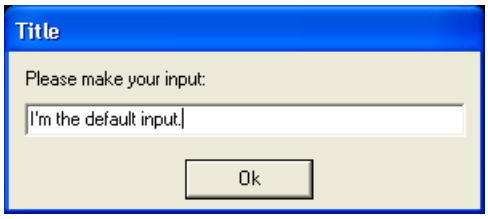
```
Text\$ = InputRequester(Title$, Message$, DefaultText$)
```

Description

Opens a blocking input requester to enter some text.

Example:

```
  Input$ = InputRequester("Title", "Please make your input:",
   "I'm the default input.")

  If Input$ > ""
    a$ = "You entered in the requester:" + Chr(10)   ; Chr(10)
   only needed
    a$ + Input$
                                                      ; for
   line-feed
  Else
    a$ = "The requester was cancelled or there was nothing
   entered."
  EndIf
  MessageRequester("Information", a$, 0)
  End
```



116.4 MessageRequester

Syntax

```
Result = MessageRequester(Title$, Text$ [, Flags])
```

Description

Opens a blocking requester to display some information. The program execution is totally stopped until the user close the requester.

The optional Flags parameter can be one of the following:

```
#PB_MessageRequester_YesNo        : to have 'yes' or 'no' buttons
#PB_MessageRequester_YesNoCancel  : to have 'yes', 'no' and 'cancel' buttons
#PB_MessageRequester_Ok           : to have the 'ok' only button
```

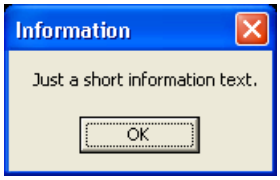
Possible return values are following constants:

```
#PB_MessageRequester_Yes    : the 'yes' button was pressed
#PB_MessageRequester_No     : the 'no' button was pressed
#PB_MessageRequester_Cancel : the 'Cancel' button was pressed
```

Example:

```
; Simple MessageRequester  (normally used for information purposes only)
; (result will be always the same, so we don't check it here)
MessageRequester("Information", "Just a short information text.", #PB_MessageRequester_Ok)

; MessageRequester with Yes / No buttons  (normally used for questions)
; (result will be shown in the following information requester)
Result = MessageRequester("Title", "Please make your input:", #PB_MessageRequester_YesNo)
a$ = "Result of the previously requester was: "
If Result = #PB_MessageRequester_Yes    ; pressed Yes button (Result = 6)
  a$ + "Yes"
Else                                    ; pressed No button (Result = 7)
  a$ + "No"
EndIf
MessageRequester("Information", a$, #PB_MessageRequester_Ok)

; MessageRequester with Yes / No / Cancel buttons  (normally used for questions)
```

```
; (result will be shown in the following information requester)
Result = MessageRequester("Title", "Please make your input:", #PB_MessageRequester_YesNoCancel)
a$ = "Result of the previously requester was: "
If Result = #PB_MessageRequester_Yes        ; pressed Yes button
  a$ + "Yes"
ElseIf Result = #PB_MessageRequester_No     ; pressed No button
  a$ + "No"
Else                                        ; pressed Cancel button or Esc
  a$ + "Cancel"
EndIf
MessageRequester("Information", a$, #PB_MessageRequester_Ok)
```



Note: In PureBasic you can also use many more constants of the Windows API. But be aware, that these constants are not cross-platform compatible! An overview you will find in the MSDN Library at the function MessageBox.

116.5 NextSelectedFilename

Syntax

```
Filename\$ = NextSelectedFilename()
```

Description

After OpenFileRequester() with #PB_Requester_MultiSelection, it returns the next selected file (if any). If the returned 'Filename$' is empty (""), then there are no more selected files.

Example:

```
Filename$ = OpenFileRequester("Choose some files","","",0, #PB_Requester_MultiSelection)

While Filename$
  Debug Filename$
  Filename$ = NextSelectedFilename()
Wend
```

116.6 OpenFileRequester

Syntax

```
Filename\$ = OpenFileRequester(Title$, DefaultFile$, Pattern$, PatternPosition [, Flags])
```

Description

Opens the standard requester for the user to choose a file. The title can be specified to replace the default one. The DefaultFile$ is useful to initialise the requester in the right directory and with the right filename.

The Pattern is a standard filter which allow to display only the files which end with such or such extension. It must be in the following form : "Text file || *.txt || Music file || *.mus;*.mod" The pattern always works in pairs: name (which really appears in the filter) and extention (ie: *.txt). Several extensions can be specified for a single type by using the ; (semi-colon) separator.

Finally 'PatternPosition' specifies which pattern must be the default. It begins from 0 up to the number of pattern. Once the requester has been closed, SelectedFilePattern() can be used to get back the selected pattern.

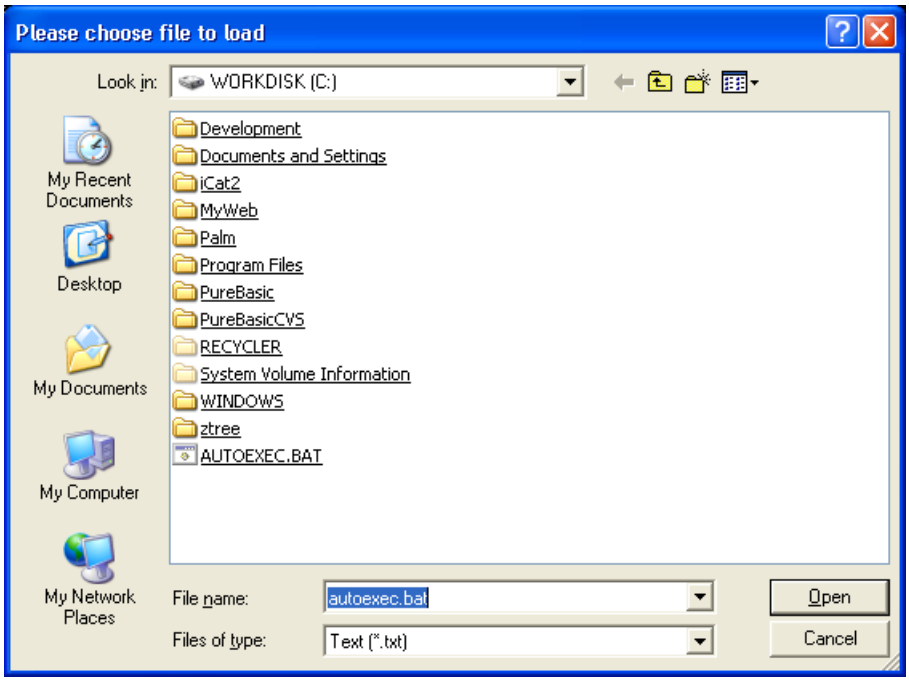
The returned 'Filename$' can be easily split into file, path and extension string with the following functions: GetFilePart() , GetPathPart() and GetExtensionPart() .

'Flags' is optional and can be a combination of the following values:

```
  #PB_Requester_MultiSelection: Enable the multiselection. (see
   NextSelectedFilename()
)
```

Example:

```
StandardFile$ = "C:\autoexec.bat"   ; set initial file+path to display
; With next string we will set the search patterns ("|" as separator) for file displaying:
;  1st: "Text (*.txt)" as name, ".txt" and ".bat" as allowed extension
;  2nd: "PureBasic (*.pb)" as name, ".pb" as allowed extension
;  3rd: "All files (*.*) as name, "*.*" as allowed extension, valid for all files
Pattern$ = "Text (*.txt)|*.txt;*.bat|PureBasic (*.pb)|*.pb|All files (*.*)|*.*"
Pattern = 0    ; use the first of the three possible patterns as standard
File$ = OpenFileRequester("Please choose file to load", StandardFile$, Pattern$, Pattern)
If File$
  MessageRequester("Information", "You have selected following file:" + Chr(10) + File$, 0)
Else
  MessageRequester("Information", "The requester was canceled.", 0)
EndIf
```



116.7 PathRequester

Syntax

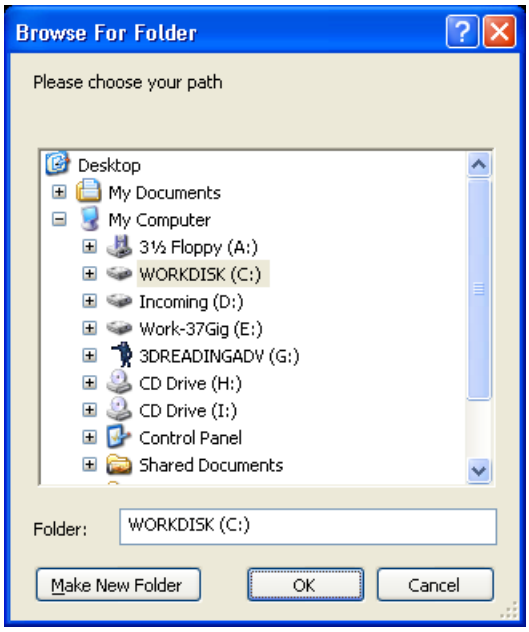
```
Filename\$ = PathRequester(Title$, InitialPath$)
```

Description

Opens the standard path requester for the user to select a path. The title can be specified to replace the default one. The InitialPath$ is useful to initialise the requester in the right directory. The path is returned with the terminating '\'

Example:

```
InitialPath$ = "C:\"    ; set initial path to display (could also be blank)
Path$ = PathRequester("Please choose your path", InitialPath$)
If Path$
  MessageRequester("Information", "You have selected following path:"+Chr(10)+Path$, 0)
Else
  MessageRequester("Information", "The requester was canceled.", 0)
EndIf
```



116.8 SaveFileRequester

Syntax

```
Filename\$ = SaveFileRequester(Title$, DefaultFile$, Pattern$,
    PatternPosition)
```

Description

Opens the standard requester for the user to save a file. The title can be specified to replace the default one. The DefaultFile$ is useful to initialise the requester in the right directory and with the right filename.

The Pattern is a standard filter which allow to display only the files which end with such or such extension. It must be in the following form : "Text file || *.txt || Music file || *.mus;*.mod" The pattern always work by pair: name (which really appears in the filter) and extention (ie: *.txt). Several extensions can be specified for a single type by using the ; (semi-colon) separator.

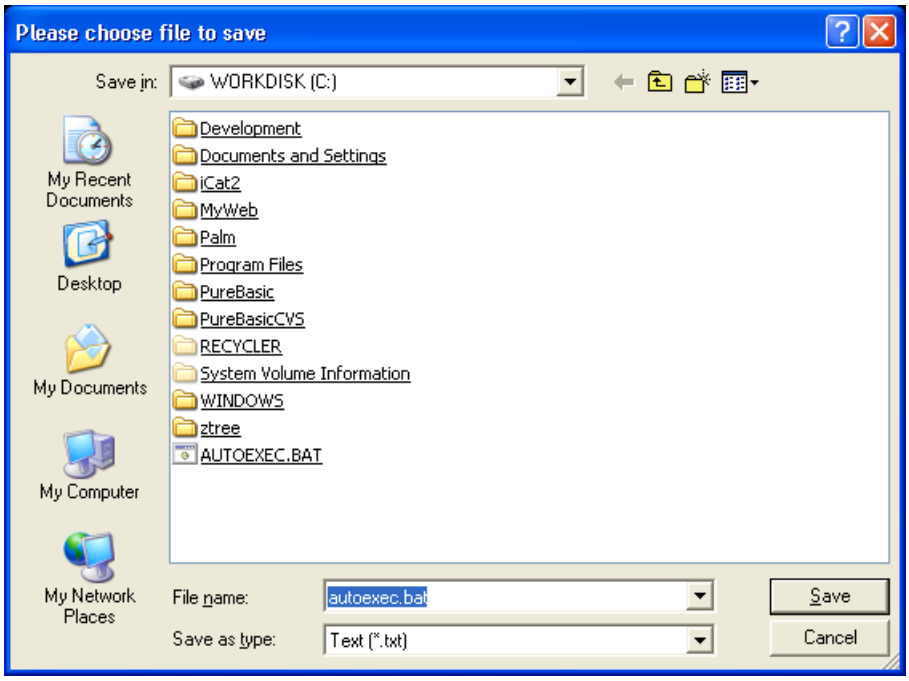
Finally 'PatternPosition' specifies which pattern must be the default. It begin from 0 up to the number of pattern. Once the requester has been closed, SelectedFilePattern() can be used to get back the selected pattern.

The returned 'Filename$' can be easily split into file, path and extension string with the following functions: GetFilePart() , GetPathPart() and GetExtensionPart() .

Example:

```
StandardFile$ = "C:\autoexec.bat"   ; set initial file+path to display
; With next string we will set the search patterns ("|" as separator) for file displaying:
;  1st: "Text (*.txt)" as name, ".txt" and ".bat" as allowed extension
;  2nd: "PureBasic (*.pb)" as name, ".pb" as allowed extension
;  3rd: "All files (*.*) as name, "*.*" as allowed extension, valid for all files
Pattern$ = "Text (*.txt)|*.txt;*.bat|PureBasic (*.pb)|*.pb|All files (*.*)|*.*"
```

```
Pattern = 0     ; use the first of the three possible patterns as standard
File$ = SaveFileRequester("Please choose file to save", StandardFile$, Pattern$, Pattern)
If File$
  MessageRequester("Information", "You have selected following file:"+Chr(10)+File$, 0)
Else
  MessageRequester("Information", "The requester was canceled.", 0)
EndIf
```



116.9 SelectedFilePattern

Syntax

```
Index = SelectedFilePattern()
```

Description

Returns the selected pattern index choosen with OpenFileRequester() or SaveFileRequester() . The first pattern is at position 0.

Example:

```
StandardFile$ = "C:\autoexec.bat"          ; initial path + file
Pattern$ = "Text (*.txt)|*.txt;*.bat|"     ; set first pattern (index = 0)
Pattern$ + "PureBasic (*.pb)|*.pb|"        ; set second pattern (index = 1)
Pattern$ + "Bmp (*.bmp)|*.bmp|"            ; set third pattern (index = 2)
Pattern$ + "Jpeg (*.jpg)|*.jpg|"           ; set fourth pattern (index = 3)
Pattern$ + "All files (*.*)|*.*"           ; set fifth pattern (index = 4)
```

```
Pattern = 1     ; use the second of the five possible patterns as standard

; Now we open a filerequester, you can change the pattern and will get the index after closing
File$ = OpenFileRequester("Please choose file to load", StandardFile$, Pattern$, Pattern)
Index = SelectedFilePattern()
If Index > -1
  MessageRequester("Information", "Following pattern index was selected: "+Str(Index), 0)
Else
  MessageRequester("Information", "The requester was canceled.", 0)
EndIf
```

116.10 SelectedFontColor

Syntax

```
Color = SelectedFontColor()
```

Description

Returns the color of the font chosen by the user with the FontRequester() . To get individual color RGB values Red() , Green() and Blue() can be used.

Supported OS

Windows

116.11 SelectedFontName

Syntax

```
Name\$ = SelectedFontName()
```

Description

Returns the name of the font chosen by the user with the FontRequester() . This name can be used directly by the LoadFont() function.

116.12 SelectedFontSize

Syntax

```
Size = SelectedFontSize()
```

Description

Returns the size of the font chosen by the user with the FontRequester() .

116.13 SelectedFontStyle

Syntax

```
Style = SelectedFontStyle()
```

Description

Returns the style of the font chosen by the user with the FontRequester() . 'Style' can contain one or several of the following values:

```
#PB_Font_Bold   : Font is bold.
#PB_Font_Italic: Font is italic.
```

To test if a value is present, use the '&' (bitwise AND) operator:

```
If SelectedFontStyle() & #PB_Font_Bold
  ; The font is bold
EndIf
```

Chapter 117

Scintilla

Scintilla is a free source code editing component. It comes with complete source code and a license that permits use in any project or product personal or commercial. The license may be viewed here . The sourcecode, as well as the library documentation may be found on the Scintilla Homepage.

From the Scintilla Homepage: As well as features found in standard text editing components, Scintilla includes features especially useful when editing and debugging source code. These include support for syntax styling, error indicators, code completion and call tips. The selection margin can contain markers like those used in debuggers to indicate breakpoints and the current line. Styling choices are more open than with many editors, allowing the use of proportional fonts, bold and italics, multiple foreground and background colours and multiple fonts.

Important: The scintilla license requires that a copyright notice be included in all software that uses it and the license text itself be included in the documentation for the software. So if this library is used in software that is to be made public, the above linked license MUST be included with the software.

PureBasic integrates the Scintilla library with the gadget library , so standard commands like ResizeGadget() or HideGadget() may be used with the scintilla control. Furthermore it provides the ScintillaSendMessage() function to communicate with the control to use its full potential. All structures and constants needed are allready defined in PureBasic.

Note: The MacOSX version of this library is a unofficial port of Scintilla done by the PureBasic Team and does not support all features and messages described in the Scintilla documentation.

117.1 InitScintilla

Syntax

```
Result = InitScintilla([LibraryName$])
```

Description

Loads the Scintilla dll which is needed to use the commands of this library. The optional 'LibraryName$' parameter specifies the path and name of the dll file (default is 'scintilla.dll'). The scintilla dll file is included in the "Compilers" folder of the PureBasic package.

Note: This command is only present on Windows, as on the other OS, the scintilla library is linked statically with the executable and therefore does not need to be loaded or distributed with the program.

Returns nonzero if the library was loaded successfully and zero if not.

Supported OS

Windows

117.2 ScintillaGadget

Syntax

```
Result = ScintillaGadget(#Gadget, x, y, Width, Height,
    @Callback())
```

Description

Creates a new scintilla editing control in the current GadgetList. If #PB_Any is used as '#Gadget' parameter, the new gadget number will be returned as 'Result'.

'@callback()'should be the address of a procedure to receive events from the control. It must be defined as follows, where 'Gadget' is the gadget number of the control and *scinotify points to a structure with information on the event:

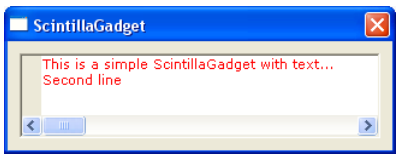
```

ProcedureDLL ScintillaCallBack(Gadget,
 *scinotify.SCNotification)
  ;
  ; The ProcedureDLL declaration is important for the
 callback to work correctly on MacOSX,
  ; on all other OS it has no effect.
  ;
EndProcedure
```

After creation, Scintilla specific commands may be sent to the control with the ScintillaSendMessage() command. In addition common gadget commands like ResizeGadget() or HideGadget() may be used with the control as well.

Example:

```
If OpenWindow(0, 0, 0, 320, 90, "ScintillaGadget",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)

  If InitScintilla()
     ScintillaGadget(0, 10, 10, 300, 70, #Null)

     ; Output set to red color
     ScintillaSendMessage(0, #SCI_STYLESETFORE, 0, RGB(255,
 0, 0))

     ; Set the initial text to the ScintillaGadget
     ScintillaSendMessage(0, #SCI_SETTEXT, 0, @"This is a
 simple ScintillaGadget with text...")

     ; Adding a second line of text with linebreak before
     Text$ = Chr(10) + "Second line"
     ScintillaSendMessage(0, #SCI_APPENDTEXT, Len(Text$),
 @Text$)
```

```
    EndIf

    Repeat : Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```



117.3 ScintillaSendMessage

Syntax

```
Result = ScintillaSendMessage(#Gadget, Message [, Param [,
    lParam]])
```

Description

Sends a message to the scintilla control to perform a specific task. More information about the possible messages may be found on the Scintilla Homepage. The #SCI_[...] constants representing the possible values for 'Message' are already defined in PB. The values for 'Param' and 'lParam' depend on the message. Not specifying them will pass a 0 value.

Returns the result of the message sent.

Chapter 118

SerialPort

The Serial port (also know as the RS-232 port) was first created in 1969, and despite its age, it is still widely used in the industry. The process of computer driven hardware prototype creation is simplified when the serial port is used. This library offers full access to the serial ports available on the computer. Some basic knowledge about the terms used in this documentation will be needed, check Wikipedia - RS232 for more information.

118.1 AvailableSerialPortInput

Syntax

```
Result = AvailableSerialPortInput(#SerialPort)
```

Description

Return the number of remaining bytes in the #SerialPort input buffer.

118.2 AvailableSerialPortOutput

Syntax

```
Result = AvailableSerialPortOutput(#SerialPort)
```

Description

Return the number of remaining bytes in the #SerialPort output buffer.

118.3 CloseSerialPort

Syntax

```
CloseSerialPort(#SerialPort)
```

Description

Close the specified #SerialPort previously opened with OpenSerialPort() .

Note: all remaining opened serial ports are automatically closed when the program ends.

118.4 GetSerialPortStatus

Syntax

```
Result = GetSerialPortStatus(#SerialPort, Attribute)
```

Description

Return the specified #SerialPort status. 'Attribute' may be one of the following values:

```
#PB_SerialPort_RI : Get RI signal status (0 or 1)
#PB_SerialPort_DCD: Get DCD signal status (0 or 1)
#PB_SerialPort_DSR: Get DSR signal status (0 or 1)
#PB_SerialPort_CTS: Get CTS signal status (0 or 1)
#PB_SerialPort_XonCharacter : Character used for Xon/Xoff
 handshaking sequence (between 1 and 255)
#PB_SerialPort_XoffCharacter: Character used for Xon/Xoff
 handshaking sequence (between 1 and 255)
```

118.5 IsSerialPort

Syntax

```
Result = IsSerialPort(#SerialPort)
```

Description

Tests if the given #SerialPort number indicates a valid and correctly initialized serial port. This function is bullet-proof and may be used with any value. If the 'Result' is not zero then the object is valid and initialized, otherwise it returns zero. This is the correct way to ensure than a serial port is ready to use.

118.6 SerialPortError

Syntax

```
Result = SerialPortError(#SerialPort)
```

Description

Return the error on the #SerialPort when ReadSerialPortData() , WriteSerialPortData() or WriteSerialPortString() fail.

'Result' can be a combination of the following values:

```
#PB_SerialPort_RxOver: An input buffer overflow has occurred.
                       There is either no room in the input
 buffer
                       or a character was received after the
 end-of-file (EOF)
#PB_SerialPort_OverRun: A character-buffer overrun has
 occurred.
                        The next character is lost.
#PB_SerialPort_RxParity : The hardware detected a parity
 error.
#PB_SerialPort_Frame    : The hardware detected a framing
 error.
```

```
#PB_SerialPort_Break     : The hardware detected a break
 condition.
#PB_SerialPort_TxFull    : The application tried to transmit a
 character but the output buffer was full.
#PB_SerialPort_IOE       :  An I/O error occurred during
 communications with the device.
#PB_SerialPort_WaitingCTS  : Specifies whether transmission
 is waiting for the CTS (clear-to-send) signal to be sent.
#PB_SerialPort_WaitingDSR  : Specifies whether transmission
 is waiting for the DSR (data-set-ready) signal to be sent.
#PB_SerialPort_WaitingRLSD : Specifies whether transmission
 is waiting for the RLSD (receive-line-signal-detect) signal
 to be sent.
#PB_SerialPort_XoffReceived: Specifies whether transmission
 is waiting because the XOFF character was received.
#PB_SerialPort_XoffSent    : Specifies whether transmission
 is waiting because the XOFF character was transmitted.
                             Transmission halts when the XOFF
 character is transmitted to a system that takes the next
                             character as XON, regardless of
 the actual character.
#PB_SerialPort_EOFSent: Specifies whether the end-of-file
 (EOF) character has been received.
```

118.7 SerialPortID

Syntax

```
SerialPortID = SerialPortID(#SerialPort)
```

Description

Returns the unique ID which identifies the given '#SerialPort' within the operating system. This function is very useful when another library requires a serial port reference.

118.8 OpenSerialPort

Syntax

```
Result = OpenSerialPort(#SerialPort, SerialPortName$, Bauds,
   Parity, Data, Stop, HandshakeMode, InputBufferSize,
   OutputBufferSize)
```

Description

Open a serial port for use. If #PB_Any is used as '#SerialPort' parameter, the new serial port number will be returned as 'Result'. 'SerialPortName$' is the text identifier for the serial port, for example "COM1" on Windows or "/dev/ttyS0" on Linux. 'Bauds' determines the speed of the serial connection and may be one of the following values:

```
50, 75, 110, 150, 300, 600, 1200, 1800, 2400
4800, 9600, 19200, 38400, 57600 or 115200
```

'Parity' defines how the parity will be handled on the connection. It may be one of the following values:

```
#PB_SerialPort_NoParity    : No parity
#PB_SerialPort_EvenParity  : Even parity
#PB_SerialPort_MarkParity  : Mark parity
#PB_SerialPort_OddParity   : Odd parity
#PB_SerialPort_SpaceParity: Space parity
```

'Data' defines the data length, in byte (usually 7 or 8). 'Stop' sets the number of stop bits (1, 1.5 or 2). 'HandshakeMode' may be one of the following values:

```
#PB_SerialPort_NoHandshake      : No handshaking
#PB_SerialPort_RtsHandshake     : No handshaking but RTS is
 set to 1
#PB_SerialPort_RtsCtsHandshake : RTS/CTS
#PB_SerialPort_XonXoffHandshake: Xon/Xoff
```

'InputBufferSize' defines the size in bytes of the input buffer. 'OutputBufferSize' defines the size in bytes of the output buffer.

If the serial port can't be opened (it may be already in use, or the parameters are wrong), 'Result' will be 0.

Example:

```
If OpenSerialPort(0, "COM1", 300, #PB_SerialPort_NoParity, 8,
 1, #PB_SerialPort_NoHandshake, 1024, 1024)
  Debug "Success"
Else
  Debug "Failed"
EndIf
```

118.9 ReadSerialPortData

Syntax

```
Result = ReadSerialPortData(#SerialPort, *Buffer, Length)
```

Description

Read an arbitrary amount of data from the #SerialPort. The number of bytes which are actually read are returned in 'Result'. If the input buffer was empty, this function will block until data is available. If a read error occurs, it will return 0. To check if data is available, use AvailableSerialPortInput() .

118.10 SerialPortTimeouts

Syntax

```
SerialPortTimeouts(#SerialPort, RIT, RTTC, RTTM, WTTC, WTTM)
```

Description

Change the default #SerialPort timeouts.

'RIT' (ReadIntervalTimeout): Specifies the maximum time, in milliseconds, which are allowed to elapse between the arrival of two characters on the communications line. Default value is 100.

'RTTC' (ReadTotalTimeoutConstant): Specifies the constant, in milliseconds, used to calculate the total time-out period for read operations. For each read operation, this value is

added to the product of the ReadTotalTimeoutMultiplier member and the requested number of bytes. Default value is 100.

'RTTM' (ReadTotalTimeoutMultiplier): Specifies the multiplier, in milliseconds, used to calculate the total time-out period for read operations. For each read operation, this value is multiplied by the requested number of bytes to be read. Default value is 10.

'WTTC' (WriteTotalTimeoutConstant): Specifies the constant, in milliseconds, used to calculate the total time-out period for write operations. For each write operation, this value is added to the product of the WriteTotalTimeoutMultiplier member and the number of bytes to be written. Default value is 10.

'WTTM' (WriteTotalTimeoutMultiplier): Specifies the multiplier, in milliseconds, used to calculate the total time-out period for write operations. For each write operation, this value is multiplied by the number of bytes to be written. Default value is 100.

118.11 SetSerialPortStatus

Syntax

```
SetSerialPortStatus(#SerialPort, Attribute, Value)
```

Description

Change the specified #SerialPort status. 'Attribute' may be one of the following values:

```
#PB_SerialPort_DTR: Set DTR signal status (0 or 1)
#PB_SerialPort_RTS: Set RTS signal status (0 or 1)
#PB_SerialPort_TXD: Set TXD signal status (0 or 1)
#PB_SerialPort_XonCharacter : Character used for Xon/Xoff
 handshaking sequence (between 1 and 255). Default value is
 $11.
#PB_SerialPort_XoffCharacter: Character used for Xon/Xoff
 handshaking sequence (between 1 and 255). Default value is
 $13.
```

Example:

```
If OpenSerialPort(0, "COM1", 300, #PB_SerialPort_NoParity, 8,
 1, #PB_SerialPort_XonXoffHandshake, 1024, 1024)
  SetSerialPortStatus(0, #PB_SerialPort_XonCharacter, 8)
  SetSerialPortStatus(0, #PB_SerialPort_XoffCharacter, 9)
  Debug "Success"
Else
  Debug "Failed"
EndIf
```

118.12 WriteSerialPortData

Syntax

```
Result = WriteSerialPortData(#SerialPort, *Buffer, Length)
```

Description

Write an arbitrary amount of data to the #SerialPort. The number of bytes actually written are returned in 'Result'. To check how much data is available in the output buffer, use AvailableSerialPortOutput() .

118.13 WriteSerialPortString

Syntax

```
Result = WriteSerialPortString(#SerialPort, String$ [, Format])
```

Description

Write a 'String$' to the #SerialPort. 'Format' may be PB_Ascii, PB_UTF8 or PB_UTF16 (default is PB_Ascii, see Unicode for further informations). To check how much data is available in the output buffer, use AvailableSerialPortOutput() .

Chapter 119

Sort

Often, elements must be sorted to be usable or more convenient. PureBasic offers highly optimized functions in order to sort arrays and linked lists , either in ascending or descending order.

119.1 SortArray

Syntax

```
SortArray(ArrayName(), Options [, Start, End])
```

Description

Sort the specified array , according to the given 'Options'. The array may be of one of basic type : byte, word, long, integer, string or float. An optional 'Start' and 'End' range index may be specified.

'Options' can take the following values:

#PB_Sort_Ascending : Sort the array in ascending order (low values first)
#PB_Sort_Descending: Sort the array in descending order (high values first)
#PB_Sort_NoCase : Sort the string array without case sensitive (a=A, b=B etc..)

Note: If an array is not fully filled then null elements will be sorted first in ascending order and last in descending order.

119.2 SortList

Syntax

```
SortList(ListName(), Options [, Start, End])
```

Description

Sort the specified list , according to the given 'Options'. The list may be of one of basic type : byte, word, long, integer, string or float. An optional 'Start' and 'End' range index may be specified.

'Options' can take the following values:

#PB_Sort_Ascending : Sort the list in ascending order (low values first)
#PB_Sort_Descending: Sort the list in descending order (high values first)
#PB_Sort_NoCase : Sort the string list without case sensitive (a=A, b=B etc..)

119.3 SortStructuredArray

Syntax

```
SortStructuredArray(ArrayName(), Options,
   OffsetOf(Structure\Field), Type [, Start, End])
```

Description

Sort the specified array , according to the given 'Options'. The array must have an associated structure . OffsetOf() may be used to retrieve the field offset in the structure associated to the array. The 'Type' parameter determine the type of the field in the structure. It must match the real structure field type. Available types are :

```
#PB_Sort_Byte      : The structure field to sort is a byte (.b)
#PB_Sort_Word      : The structure field to sort is a word (.w)
#PB_Sort_Long      : The structure field to sort is a long (.l)
#PB_Sort_String    : The structure field to sort is a string
 (.s or $, fixed strings are not supported)
#PB_Sort_Float     : The structure field to sort is a float
 (.f)
#PB_Sort_Double    : The structure field to sort is a double
 (.d)
#PB_Sort_Quad      : The structure field to sort is a quad (.q)
#PB_Sort_Character: The structure field to sort is a
 character (.c)
#PB_Sort_Integer   : The structure field to sort is an integer
 (.i)
#PB_Sort_Ascii     : The structure field to sort is an ascii
 character (.a)
#PB_Sort_Unicode   : The structure field to sort is a unicode
 character (.u)
```

An optional 'Start' and 'End' range index may be specified.

'Options' can take the following values:

#PB_Sort_Ascending : Sort the array in ascending order (low values first)
#PB_Sort_Descending: Sort the array in descending order (high values first)
#PB_Sort_NoCase : Sort the string array without case sensitive (a=A, b=B etc..)

Notes: If an array is not fully filled then null elements will be sorted first in ascending order and last in descending order.
Fixed strings are not supported by the sort routine.

Example:

```

Structure Animal
  Name$
  Speed.l
EndStructure

Dim Animals.Animal(2)

Animals(0)\Name$ = "Tiger"
Animals(0)\Speed = 10

Animals(1)\Name$ = "Jaguar"
Animals(1)\Speed = 40

```

```
Animals(2)\Name$ = "Zebre"
Animals(2)\Speed = 30

SortStructuredArray(Animals(), 0, OffsetOf(Animal\Name$),
 #PB_Sort_String)

For k=0 To 2
  Debug Animals(k)\Name$+" - Speed: "+Str(Animals(k)\Speed)
Next

SortStructuredArray(Animals(), 0, OffsetOf(Animal\Speed),
 #PB_Sort_Long)

For k=0 To 2
  Debug Animals(k)\Name$+" - Speed: "+Str(Animals(k)\Speed)
Next
```

119.4 SortStructuredList

Syntax

```
SortStructuredList(ListName(), Options,
   OffsetOf(Structure\Field), Type [, Start, End])
```

Description

Sort the specified list , according to the given 'Options'. The list must have an associated structure . OffsetOf() may be used to retrieve the field offset in the structure associated to the list. The 'Type' parameter determine the type of the field in the structure. It must match the real structure field type. Available types are :

```
#PB_Sort_Byte      : The structure field to sort is a byte (.b)
#PB_Sort_Word      : The structure field to sort is a word (.w)
#PB_Sort_Long      : The structure field to sort is a long (.l)
#PB_Sort_String    : The structure field to sort is a string
 (.s or $, fixed strings are not supported)
#PB_Sort_Float     : The structure field to sort is a float
 (.f)
#PB_Sort_Double    : The structure field to sort is a double
 (.d)
#PB_Sort_Quad      : The structure field to sort is a quad (.q)
#PB_Sort_Character: The structure field to sort is a
 character (.c)
#PB_Sort_Integer   : The structure field to sort is an integer
 (.i)
#PB_Sort_Ascii     : The structure field to sort is an ascii
 character (.a)
#PB_Sort_Unicode   : The structure field to sort is a unicode
 character (.u)
```

An optional 'Start' and 'End' range index may be specified.

'Options' can take the following values:

#PB_Sort_Ascending : Sort the list in ascending order (low values first)
#PB_Sort_Descending: Sort the list in descending order (high values first)
#PB_Sort_NoCase : Sorts the string list without case sensitive (a=A, b=B etc..)

Note: Fixed strings are not supported by the sort routine.

Example:

```

Structure Animal
  Name$
  Speed.l
EndStructure

NewList Animals.Animal()

AddElement(Animals())
Animals()\Name$ = "Tiger"
Animals()\Speed = 10

AddElement(Animals())
Animals()\Name$ = "Jaguar"
Animals()\Speed = 40

AddElement(Animals())
Animals()\Name$ = "Zebre"
Animals()\Speed = 30

SortStructuredList(Animals(), 0, OffsetOf(Animal\Name$),
 #PB_Sort_String)

ForEach Animals()
  Debug Animals()\Name$+" - Speed: "+Str(Animals()\Speed)
Next

SortStructuredList(Animals(), 0, OffsetOf(Animal\Speed),
 #PB_Sort_Long)

ForEach Animals()
  Debug Animals()\Name$+" - Speed: "+Str(Animals()\Speed)
Next
```

Chapter 120

Sound

The PureBasic sound system provides an easy way to have sound inside application or game. It uses special functions to get the maximum speed of available hardware. It requires DirectX 7 (Windows 9x/2000) or DirectX 3 when using the Windows NT4.0 compatibility mode.

120.1 CatchSound

Syntax

```
Result = CatchSound(#Sound, MemoryAddress [, Length [, Flags]])
```

Description

Load a WAV or any other format supported by the SoundPlugin library contained at the specified address. If #PB_Any is used as '#Sound' parameter, the new sound number will be returned as 'Result'. If the 'Result' is not 0, the sound has been correctly loaded and is ready to be played with the function PlaySound() , else the sound loading has failed. With WAV files, the 'Length' parameter doesn't need to be specified. The following functions can be used to enable automatically more sound formats:

UseFLACSoundDecoder() : 'Length' parameter must be specified
UseOGGSoundDecoder() : 'Length' parameter must be specified

The 'Flags' optional parameter can be a combination of the following values:

```
  #PB_Sound_Streaming: Enable streaming playback (only
   supported for FLAC
 and OGG
))
```

Example:

```
1   CatchSound(0, ?Music)
2   End
3
4   DataSection
5     Music: IncludeBinary "Sound.wav"
```

Note: The "?" is a pointer to a label. More informations about pointers and memory access can be found in the relating chapter here .

120.2 FreeSound

Syntax

```
FreeSound(#Sound)
```

Description

Stop and remove a sound previously loaded with LoadSound() or CatchSound() from memory. Once a sound has been freed, it can't be played anymore.

Note: all remaining sounds are automatically freed when the program ends.

120.3 InitSound

Syntax

```
Result = InitSound()
```

Description

Initialize the sound environment. If the 'Result' returns 0, then no sound can be played on this computer, or the sound device is busy. This function must be always called before any other sound function and should always check its result. If the sound environment fails, it's absolutely necessary to disable all the sound functions calls.

Possible causes of failure under MS Windows: DirectX 7 isn't available or no sound card are present. If the program need to run on NT4.0, be sure to enable the 'NT4.0 Compliant executable' in the compiler options.

120.4 IsSound

Syntax

```
Result = IsSound(#Sound)
```

Description

Tests if the given #Sound number is a valid and correctly initialized sound. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a sound is ready to use.

120.5 LoadSound

Syntax

```
Result = LoadSound(#Sound, Filename$ [, Flags])
```

Description

Load a WAV or any other format supported by the SoundPlugin library into memory. If #PB_Any is used as '#Sound' parameter, the new sound number will be returned as 'Result'. If the 'Result' is not 0, the sound has been correctly loaded and is ready to be played with the function PlaySound() , else the sound loading has failed.

The 'Flags' optional parameter can be a combination of the following values:

```
  #PB_Sound_Streaming: Enable sound streaming playback (only
   supported for FLAC
 and OGG
))
```

The following functions can be used to enable automatically more sound format:

UseFLACSoundDecoder()
UseOGGSoundDecoder()

120.6 PlaySound

Syntax

```
Result = PlaySound(#Sound [, Flags [, Volume]])
```

Description

Start to play the specified #Sound. 'Flags' is optional and can be a combination of the following values:

```
  #PB_Sound_Loop        : play the sound continuously (starts
   again when end is reached)
  #PB_Sound_MultiChannel: play the sound in a new channel
   instead of stopping the
        previously played sound. This allows to use the same
   sound and to play it on different
        channels at once. 'Result' will be new allocated
   channel, and can be used by the
        other sound commands: SoundVolume()
, SoundFrequency()
, SoundPan()
 and StopSound()
.
```

'Volume' is optional and is useful to set the initial volume of the #Sound. The valid values are from 0 (no volume) to 100 (full volume). The default value is 100.

120.7 SoundFrequency

Syntax

```
SoundFrequency(#Sound, Frequency [, Channel])
```

Description

Set the new frequency (in Hz), in realtime, for the #Sound. The frequency value is saved for the #Sound, so it's not needed to call it everytime. The valid values are from 1000 Hz to 100000 Hz.

'Channel' is optional and specify on which channel the sound frequency should be changed. A 'Channel' is available when using PlaySound() with #PB_Sound_MultiChannel.

Supported OS

Windows

120.8 SoundPan

Syntax

```
SoundPan(#Sound, Pan [, Channel])
```

Description

Set the new pan, in realtime, for the #Sound. The pan value is saved for the #Sound, so it's not needed to call it everytime. The panning is a way to play a sound on a stereo equipment. The valid values are from -100 (full left) to 100 (full right). If the pan is 0, then the sound is played on right and left speaker, equaly.

'Channel' is optional and specify on which channel the sound pan should be changed. A 'Channel' is available when using PlaySound() with #PB_Sound_MultiChannel.

Supported OS

Windows, MacOS X

120.9 SoundVolume

Syntax

```
SoundVolume(#Sound, Volume [, Channel])
```

Description

Set the new volume, in realtime, for the #Sound. The volume value is saved for the #Sound, so it's not needed to call it everytime. The valid values are from 0 (no volume) to 100 (full volume).

'Channel' is optional and specify on which channel the sound volume should be changed. A 'Channel' is available when using PlaySound() with #PB_Sound_MultiChannel.

120.10 StopSound

Syntax

```
StopSound(#Sound [, Channel])
```

Description

Stop the specified sound (if it was playing). If the #Sound value is -1, then all sounds currently playing are stopped.

'Channel' is optional and specify on which channel the sound should be stop. A 'Channel' is available when using PlaySound() with #PB_Sound_MultiChannel.

Chapter 121

Sound3D

The Sound3D library allows to handle sounds in 3D space. For example, it can handle automatic fading depending of the distance. It is based on the regular sound library syntax and behaviour.

It uses the 3D engine, so InitEngine3D() has to be called succesfully before using these functions.

121.1 FreeSound3D

Syntax

```
FreeSound3D(#Sound3D)
```

Description

Stop and remove a sound previously loaded with LoadSound3D() from memory. Once a sound has been freed, it can't be played anymore.

Note: all remaining sounds are automatically freed when the program ends.

121.2 IsSound3D

Syntax

```
Result = IsSound3D(#Sound3D)
```

Description

Tests if the given #Sound3D number is a valid and correctly initialized sound. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a sound is ready to use.

121.3 LoadSound3D

Syntax

```
Result = LoadSound3D(#Sound3D, Filename$ [, Flags])
```

Description

Load a WAV or an OGG sound file into memory. The sound has to be mono, as stereo sounds doesn't allow space positioning. If #PB_Any is used as '#Sound3D' parameter, the new sound number will be returned as 'Result'. If the 'Result' is not 0, the sound has been correctly loaded and is ready to be played with the function PlaySound3D() , else the sound loading has failed.

The 'Flags' optional parameter can be a combination of the following values:

```
#PB_Sound3D_Streaming: Enable sound streaming playback
```

Note: a sound doesn't hold its position. It can be attached to a node object to have its own position.

121.4 PlaySound3D

Syntax

```
Result = PlaySound3D(#Sound3D [, Flags])
```

Description

Start to play the specified sound. 'Flags' is optional and can be a combination of the following values:

```
#PB_Sound3D_Loop: play the sound continuously (starts again
 when end is reached)
```

121.5 SoundVolume3D

Syntax

```
SoundVolume3D(#Sound3D, Volume)
```

Description

Set the new volume, in realtime, for the #Sound3D. The volume value is saved for the #Sound3D, so it's not needed to call it everytime. The valid values are from 0 (no volume) to 100 (full volume).

121.6 StopSound3D

Syntax

```
StopSound3D(#Sound3D)
```

Description

Stop the specified sound (if it was playing). If the #Sound3D value is -1, then all sounds currently playing are stopped.

121.7 SoundID3D

Syntax

```
SoundID3D = SoundID3D(#Sound3D)
```

Description

Returns the unique ID which identifies the given '#Sound3D' in the operating system. This function is very useful when another library needs a window reference.

121.8 SoundRange3D

Syntax

```
SoundRange3D(#Sound3D, Minimum, Maximum)
```

Description

Set the range, in world units, for the sound emission. The 'Minimum' distance is the distance from the sound location when the sound volume starts fading. The 'Maximum' distance is the distance from the sound location when the sound can still be heard. Above this distance, the sound will stop playing. Between these two value, the sound will fade according to the current listener position. The first camera is the sound listener.

121.9 SoundCone3D

Syntax

```
SoundCone3D(#Sound3D, InnerCone.f, OuterCone.f, OuterConeVolume)
```

Description

Set the cone angle, to create a directional sound. The 'InnerCone' and 'OuterCone' value determine the cone angle (values are from 0 to 360). 'OuterConeVolume' is the sound volume outside of the cone (values are from 0 to 100).

Chapter 122

SoundPlugin

PureBasic supports external sound formats through a dynamic native plugin system. This means that only the needed encoder or decoder is added to the final executable, which decreases the final program size substantially. For example, if only the OGG decoder is needed, only the code dealing with the OGG decoder will be used. Another nice feature is the automatic sound format detection, whenever several different decoders are used. The following functions support the Sound plugins: LoadSound() , and CatchSound() .

122.1 UseFLACSoundDecoder

Syntax

```
UseFLACSoundDecoder()
```

Description

Enables the FLAC (Free Lossless Audio Codec) sound support for CatchSound() and LoadSound() .

Since FLAC is a not destructive compression codec, the file size will remain quite large, especially compared to OGG for example. This allows for a reduction in file size, while still retaining all the original sound information. For more information: Wikipedia - FLAC.

Sound streaming is supported for this plugin.

122.2 UseOGGSoundDecoder

Syntax

```
UseOGGSoundDecoder()
```

Description

Enables the OGG (OGG Vorbis) sound support for CatchSound() and LoadSound() .

The OGG format uses destructive compression (lossy) which entails the loss of some of the original information when it is compressed. This is actually one of the best lossy compression algorithms available, which can dramatically decrease the sound file size without the introduction of unnecessary audio artifacts. For more information: Wikipedia - OGG.

Sound streaming is supported for this plugin.

Chapter 123

Sprite

'Sprites' are well known by game players. These are small pictures, sometimes called 'brushes', which can be displayed at any position on a screen. The sprites can move over graphics using a transparent layer. Even better, PureBasic allows you to perform real time effects like real Shadow, AlphaBlending, RGBFilters, Alpha channel, ... all this in windowed or full screen mode !

As the sprites are very linked to the screen, the screen related functions have been included into the sprite library (it may change later). After initializing the screen and sprite enviroment via InitSprite() you can start opening a full-size or windowed screen.

Windows: DirectX 9 is used for sprite handling which allows your programs to use hardware acceleration if available. Three additional Subsystems are also available, depending of the needs: "OpenGL", "DirectX7" and "NT4"

Linux: SDL (Simple Directmedia Layer) is used to manage the sprites. This allows your programs to use hardware acceleration if available. An additional "OpenGL" Subsystem is also available, which use OpenGL as backend for all sprite and sprite3D functions.

MacOS X: OpenGL is used to manage the sprites which allows to use hardware acceleration.

123.1 CatchSprite

Syntax

```
Result = CatchSprite(#Sprite, MemoryAddress [, Mode])
```

Description

Load the specified sprite from the given memory area. If #PB_Any is used as '#Sprite' parameter, the new sprite number will be returned as 'Result'. A screen should be opened with OpenScreen() or OpenWindowedScreen() before loading a sprite. If an error occurs the value 0 is returned. Sprites can be in BMP format (1 bit, 4 bits, 8 bits, 16 bits, 24 bits or 32 bits color depth, in uncompressed (non RLE) form) or any other format supported by the ImagePlugin library. A catched sprite can be freed by using the FreeSprite() function. The 'CatchSprite' function is useful when using the 'IncludeBinary' PureBasic keyword. The images can be packed inside the executable. Nevertheless, use this option very carefully, as it will take more memory than storing the file in an external file (the file will be in both executable memory and loaded into physical memory). The following functions can be used to enable automatically more image formats:

UseJPEGImageDecoder()
UseJPEG2000ImageDecoder()
UsePNGImageDecoder()
UseTIFFImageDecoder()

UseTGAImageDecoder()

The 'Mode' parameter is optional and can take the following values (also combined with the '||' operator):

```
  0                        : Default mode. Sprite resides in
   video memory (if possible)
  #PB_Sprite_Memory        : Sprite is loaded into main PC
   memory (for SpecialFX)
  #PB_Sprite_Alpha         : Sprite is 8 bits grayscale and will
   be used by DisplayAlphaSprite()
 or DisplayShadowSprite()

  #PB_Sprite_Texture       : Sprite is created with 3D support -
   useful for the CreateSprite3D()
 function of the Sprite3D library.
  #PB_Sprite_AlphaBlending: Sprite is created with per pixel
   alpha-channel support, the image format has to support it
   (only
                            PNG and TIFF for now). The
   #PB_Sprite_Texture needs to be specified as well, and the
   sprite
                            has to be displayed with
   DisplaySprite3D()
.
```

Example:

```
1   CatchSprite(0, ?Pic)
2   End
3
4   Pic: IncludeBinary "Sprite.bmp"
```

Note: The "?" is a pointer to a label. More informations about pointers and memory access can be found in the relating chapter here .

When using 8 bits screenmode (256 colors, palettized), only 8bits BMP are accepted as sprite format. The ImagePlugins or other BMP color formats aren't supported.

123.2 ChangeAlphaIntensity

Syntax

```
ChangeAlphaIntensity(R, G, B)
```

Description

Changes the AlphaIntensity independently from each of the Red, Green and Blue channels. This is used with the DisplayAlphaSprite() function. This means than an AlphaSprite can be rendered in red, green or any color instead of transparent. This allows nice real time effects.

123.3 ClipSprite

Syntax

```
Result = ClipSprite(#Sprite, x, y, Width, Height)
```

Description

Add a clip zone to the specified sprite. For example, if a sprite is 100*100 (Width*Height) and a clipping zone is added as x=10, y=10, Width=20, Height=20 then when the sprite is displayed, only the rectangular area starting from x=10, y=10 with width=20 and height=20 will be displayed. To remove the clipping of the #Sprite, use #PB_Default constant for 'x', 'y', 'Width' and 'Height'.

Note: On some older gfx cards the clipping with ClipSprite() doesn't work properly, if the sprite is larger than the used screen. ClipSprite() has no effect when used with a Sprite3D .

123.4 CopySprite

Syntax

```
Result = CopySprite(#Sprite1, #Sprite2 [, Mode])
```

Description

Copy the #Sprite1 to #Sprite2. If #PB_Any is used as '#Sprite2' parameter, the new sprite number will be returned as 'Result'. #Sprite2 is created if it does not exist.

The 'Mode' parameter is optional and can take the following values (also combined with the '||' operator):

```
  0                        : Default mode. Sprite resides in
   video memory (if possible)
  #PB_Sprite_Memory        : Sprite is loaded into main PC
   memory (for SpecialFX)
  #PB_Sprite_Alpha         : Sprite is 8 bits grayscale and will
   be used by DisplayAlphaSprite()
 or DisplayShadowSprite()

  #PB_Sprite_Texture       : Sprite is created with 3D support -
   useful for the CreateSprite3D()
 function of the Sprite3D library.
  #PB_Sprite_AlphaBlending: Sprite is created with per pixel
   alpha-channel support, the image format has to support it
   (only
                            PNG and TIFF for now). The
   #PB_Sprite_Texture needs to be specified as well, and the
   sprite
                            has to be displayed with
   DisplaySprite3D()
.
```

123.5 CreateSprite

Syntax

```
Result = CreateSprite(#Sprite, Width, Height [, Mode])
```

Description

Creates an empty sprite with given dimensions. If #PB_Any is used as '#Sprite' parameter, the new sprite number will be returned as 'Result'.

The 'Mode' parameter is optional and can take the following values:

```
  0                  : Default mode. Sprite resides in video
   memory (if possible)
  #PB_Sprite_Memory : Sprite is loaded into main PC memory (for
   SpecialFX)
  #PB_Sprite_Alpha  : Sprite is 8 bits grayscale and will be
   used by DisplayAlphaSprite()
 or DisplayShadowSprite()

  #PB_Sprite_Texture: Sprite is created with 3D support -
   useful for the CreateSprite3D()
 function of the Sprite3D library.
  #PB_Sprite_AlphaBlending: Sprite is created with per pixel
   alpha-channel support, the image format has to support it
   (only
                            PNG and TIFF for now). The
   #PB_Sprite_Texture needs to be specified as well, and the
   sprite
                            has to be displayed with
   DisplaySprite3D()
.
```

123.6 DisplayAlphaSprite

Syntax

```
DisplayAlphaSprite(#Sprite, x, y)
```

Description

Display the #Sprite at the specified position on the current screen. The color 128,128,128 (gray) is considered as a transparent color (this color will not be displayed). The sprite emulates an 'AlphaChannel' so any tricks possible with such a channel are available with it. The sprite must be an 8 bit (256 colors) sprite, with only gray colors. On this sprite, each gray intensity less than 128,128,128 makes the area progressively darker and each intensity above 128,128,128 meakes the area progressively lighter. This allows your program to do very cool effects in real time like soft shadows etc.. the possibilities are huge. ChangeAlphaIntensity() can be used to control each R,G,B component of the Alpha Channel. The sprite must be loaded with LoadSprite() using the #PB_Sprite_Alpha option. This function is clipped, so it's perfectly legal to display the sprite outside of the screen.

This function works much faster if StartSpecialFX() is used.

123.7 DisplayRGBFilter

Syntax

```
DisplayRGBFilter(x, y, Width, Height, R, G, B)
```

Description

Displays a (translucent) color filter over the screen. This allows you to display rectangular areas in any color very quickly.

This function works much faster if StartSpecialFX() is used.

123.8 DisplayShadowSprite

Syntax

```
DisplayShadowSprite(#Sprite, x, y)
```

Description

Display the #Sprite at the specified position on the current screen. The color 0 is considered transparent (this color will not be displayed). The sprite is displayed like a shadow, using a faster algorithm than DisplayAlphaSprite() , but with only one intensity. The sprite must be an 8 bit (256 colors) sprite. It must be loaded with LoadSprite() using the #PB_Sprite_Alpha option. This function is clipped, so it's perfectly legal to display the sprite outside of the screen.

This function works much faster if StartSpecialFX() is used.

123.9 DisplaySolidSprite

Syntax

```
DisplaySolidSprite(#Sprite, x, y, Color)
```

Description

Display the #Sprite at the specified position on the current screen. The color 0 is considered transparent (this color will not be displayed).

The sprite is displayed in a single 'Color'. RGB() can be used to get a valid color value. A table with common colors is available here .

The sprite must be an 8 bit (256 colors) sprite. It must be loaded with LoadSprite() using the #PB_Sprite_Alpha option. This function is clipped, so it's perfectly legal to display the sprite outside of the screen.

This function works much faster if StartSpecialFX() is used.

123.10 DisplaySprite

Syntax

```
DisplaySprite(#Sprite, x, y)
```

Description

Display the #Sprite at the specified position on the current screen. There is no transparent color. This function is clipped, so it's perfectly legal to display the sprite outside of the screen.

123.11 DisplayTranslucentSprite

Syntax

```
DisplayTranslucentSprite(#Sprite, x, y, Intensity)
```

Description

Display the #Sprite at the specified position on the current screen. The color 0 is considered transparent (this color will not be displayed). The intensity controls the transparency of the sprite over the background. Intensity values can vary from 0 (fully transparent) to 255 (opaque). This effect is also known as 'AlphaBlending'. This function is clipped, so it's perfectly legal to display the sprite outside of the screen.

This function works much faster if StartSpecialFX() is used.

123.12 DisplayTransparentSprite

Syntax

```
DisplayTransparentSprite(#Sprite, x, y)
```

Description

Display the #Sprite at the specified position on the current screen. The default transparent color is 0 (black - this color will not be displayed). It's possible to change the transparent color with TransparentSpriteColor() . This function is clipped, so it's perfectly legal to display the sprite outside of the screen.

123.13 FreeSprite

Syntax

```
FreeSprite(#Sprite)
```

Description

Remove the specified sprite from memory. It's not possible to the sprite anymore after calling this function.

Note: all remaining sprites are automatically freed when the program ends.

123.14 GrabSprite

Syntax

```
Result = GrabSprite(#Sprite, x, y, Width, Height [, Mode])
```

Description

Grabs the screen content from the area x,y,Width,Height, and creates a new #Sprite. If #PB_Any is used as '#Sprite' parameter, the new sprite number will be returned as 'Result'.

The 'Mode' parameter is optional and can take the following values:

```
  0                  : Default mode. Sprite resides in video
   memory (if possible)
  #PB_Sprite_Memory : Sprite is loaded into main PC memory (for
   SpecialFX)
  #PB_Sprite_Alpha  : Sprite is 8 bits grayscale and will be
   used by DisplayAlphaSprite()
 or DisplayShadowSprite()

  #PB_Sprite_Texture: Sprite is created with 3D support -
   useful for the CreateSprite3D()
 function of the Sprite3D library.
```

Note: GrabSprite() should be always called outside a StartDrawing() : StopDrawing() block.

123.15 InitSprite

Syntax

```
Result = InitSprite()
```

Description

Initialises the sprite environment for later use. You must put this function at the top of your source code if you want to use the sprite functions.

Result will be 0, if the initialization of the sprite inviroment fails. You should always test this result to see if the sprite environment has been correctly initialised. If not you must quit the program or disable all the calls to the sprite related functions. This function tries to initialise DirectX 7, so if it fails, it's probably because DirectX is not found or is a previous version.

123.16 IsSprite

Syntax

```
Result = IsSprite(#Sprite)
```

Description

Tests if the given #Sprite number is a valid and correctly initialized sprite. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a sprite is ready to use.

123.17 LoadSprite

Syntax

```
Result = LoadSprite(#Sprite, Filename$ [, Mode])
```

Description

Load the specified sprite into memory for immediate use. If #PB_Any is used as '#Sprite' parameter, the new sprite number will be returned as 'Result'.

The sprite can be in BMP format (1 bit, 4 bits, 8 bits, 16 bits, 24 bits or 32 bits color depth, in uncompressed (non RLE) form) or any other format supported by the ImagePlugin library.

A screen should be opened with OpenScreen() or OpenWindowedScreen() before loading a sprite.

If an error occurs the value 0 is returned, otherwise the sprite was loaded successfully. A loaded sprite can be freed by using the FreeSprite() function. The following functions can be used to enable automatically more image formats:

UseJPEGImageDecoder()
UseJPEGImage2000Decoder()
UsePNGImageDecoder()
UseTIFFImageDecoder()
UseTGAImageDecoder()

The 'Mode' parameter is optional and can take the following values (also combined with the '||' operator):

```
  0                        : Default mode. Sprite resides in
   video memory (if possible)
  #PB_Sprite_Memory        : Sprite is loaded into main PC
   memory (for SpecialFX)
  #PB_Sprite_Alpha         : Sprite is 8 bits grayscale and will
   be used by DisplayAlphaSprite()
 or DisplayShadowSprite()

  #PB_Sprite_Texture       : Sprite is created with 3D support -
   useful for the CreateSprite3D()
 function of the Sprite3D library.
  #PB_Sprite_AlphaBlending: Sprite is created with per pixel
   alpha-channel support, the image format has to support it
   (only
                            PNG and TIFF for now). The
   #PB_Sprite_Texture needs to be specified as well, and the
   sprite
                            has to be displayed with
   DisplaySprite3D()
.
```

Note: Sprites shouldn't be larger than the used screenmode. Using larger sprites possibly works on some hardware, on others not. Better split your large sprite to several smaller ones.

When using 8 bits screenmode (256 colors, palettized), only 8bits BMP are accepted as sprite format. The ImagePlugins or other BMP color formats aren't supported.

123.18 SaveSprite

Syntax

```
Result = SaveSprite(#Sprite, Filename$ [, ImagePlugin [,
   Flags]])
```

Description

Saves the specified #Sprite to a file. By default, the saved image will be in 24bit BMP format. Very useful for screenshots when used with the GrabSprite() function. If the 'Result' is 0, the sprite hasn't be saved (disk full, bad filename, etc.). 'ImagePlugin' is an optional parameter and can be one of the following constant:

```
  #PB_ImagePlugin_BMP      : Save the image in BMP  (format by
   default)
  #PB_ImagePlugin_JPEG     : Save the image in JPEG (the
   function UseJPEGImageEncoder()
 has to be used)
  #PB_ImagePlugin_JPEG2000: Save the image in JPEG2000 (the
   function UseJPEGImage2000Encoder()
 has to be used)
  #PB_ImagePlugin_PNG      : Save the image in PNG  (the
   function UsePNGImageEncoder()
 has to be used)
```

'Flags' is an optional parameter depending of the plugin used. For now, only the quality setting is supported: a number from 0 (worste quality) to 10 (maximum quality). Only the JPEG and JPEG 2000 plugins currently support it (default quality is set to '7' if no flags are specified).

123.19 SpriteCollision

Syntax

```
Result = SpriteCollision(#Sprite1, x1, y1, #Sprite2, x2, y2)
```

Description

Tests to see if the two sprites are overlapping. If not, the result is 0. As x and y parameters the coordinates of the two sprites are needed, the width and height of each sprite does PureBasic know itself.

This routine compares the rectangular areas around the sprites, giving a very fast but not very accurate function (depending on the shapes of your sprites). Very useful for fast arcade games.

For a more exact collision check use SpritePixelCollision() .

123.20 SpriteDepth

Syntax

```
Result = SpriteDepth(#Sprite)
```

Description

Return the color depth of the specified sprite.

123.21 SpriteHeight

Syntax

```
Result = SpriteHeight(#Sprite)
```

Description

Return the height in pixels of the specified sprite.

123.22 SpriteID

Syntax

```
SpriteID = SpriteID(#Sprite)
```

Description

Returns the unique system identifier of the #Sprite.

This result is sometimes also known as 'Handle'. Look at the extra chapter Handles and Numbers for more information.

123.23 SpritePixelCollision

Syntax

```
Result = SpritePixelCollision(#Sprite1, x1, y1, #Sprite2, x2,
    y2)
```

Description

Tests to see if the two sprites are overlapping. If not, the result is 0. As x and y parameters the coordinates of the two sprites are needed, the width and height of each sprite does PureBasic know itself.

This routine performs a transparent pixel exact collision check, giving a slower but very accurate result. To optimize the comparison you should always remove as many transparent pixels as possible so that the sprite size is fully used by the sprite (i.e. do not have large transparent borders around the actual image of the sprite).

For a less exactly but very fast collision check use SpriteCollision() .

Note: SpritePixelCollision() currently doesn't work with alpha channel (PNG, TIFF), it will be ignored.

Supported OS

Windows (DirectX)

123.24 SpriteWidth

Syntax

```
Result = SpriteWidth(#Sprite)
```

Description

Returns the width in pixels of the specified sprite.

123.25 StartSpecialFX

Syntax

```
StartSpecialFX()
```

Description

PureBasic allows you to perform real time effects like alphablending, shadow, color filter etc. These rendering effects are very CPU intensive (even if they are very optimized) and must use a special trick to be performed quickly. StartSpecialFX() creates a memory buffer to perform all the rendering and the result will be copied back to video memory when StopSpecialFX() is called. This means that hardware acceleration (i.e. graphics card chips) cannot be used during this period. All operations are performed by the main processor. This is important to understand otherwise game performance will drop a lot.

All SpecialFX operations should be grouped and rendered at one time (Shadow, translucent sprite etc). Standard sprites should be displayed after StopSpecialFX() to use the hardware acceleration, as usual. If you are using ClearScreen() , place it inside the StartSpecialFX() : ... : StopSpecialFX() block. The current display is not copied into the buffer (would be too slow), so be sure to don't call any display functions before calling StartSpecialFX() as they will be ignored.

As all the operations are done in the system memory (not on the video memory), the sprites should be loaded with the #PB_Sprite_Memory flag, to increase a lot the sprite display speed.

functions which use the SpecialFX mode are: DisplayAlphaSprite() , DisplaySolidSprite() , DisplayShadowSprite() , DisplayRGBFilter() and DisplayTranslucentSprite()

Supported OS

Windows (DirectX), Linux (SDL)

123.26 StopSpecialFX

Syntax

```
StopSpecialFX()
```

Description

Once all rendering operations are complete (see StartSpecialFX()), this function copies the buffer back to video memory and hardware acceleration is available again.

Supported OS

Windows (DirectX), Linux (SDL)

123.27 AvailableScreenMemory

Syntax

```
Result = AvailableScreenMemory()
```

Description

Returns the quantity of video memory available for the sprites and other video data (in bytes). This function must be called after OpenScreen() or OpenWindowedScreen() .

Supported OS

Windows (DirectX), Linux (SDL)

123.28 ChangeGamma

Syntax

```
ChangeGamma(RedIntensity, GreenIntensity, BlueIntensity)
```

Description

Changes the Gamma for the current screen. This only works in full screen mode (not in windowed mode). Red, Green and Blue channels intensity can be changed individually. Valid values for each channels are between 0 and 255. This function can be used to do full screen fadein/fadeout, color splashing etc.. If it does not do anything, then the hardware does not support it (no emulation is provided, due to the high number of operations required to perform it in software).

```
ChangeGamma(255, 255, 0)    ; Will remove completely the blue channel from the screen
```

Supported OS

Windows (DirectX)

123.29 ClearScreen

Syntax

```
ClearScreen(Color)
```

Description

Clear the whole screen with the specified 'color'. RGB() can be used to get a valid color value. A table with common colors is available here .

Note: ClearScreen() should be always called outside a StartDrawing() : StopDrawing() block.

123.30 CloseScreen

Syntax

```
CloseScreen()
```

Description

Close the current screen (either windowed or full screen mode). After closing a screen, all the sprites must be reloaded as the screen format has been lost and the video memory released. An application or game can switch from full screen to windowed mode on the fly without any problem.

123.31 FlipBuffers

Syntax

```
FlipBuffers()
```

Description

Flip the back and front buffers of the current screen. The invisible area is now visible and vice versa, which allows you to do a 'double-buffering' effect (flicker free graphical displays). A screen must have been opened with OpenScreen() or OpenWindowedScreen() . The way the buffer are flipped (with or without synchronization) is set by OpenScreen() or OpenWindowedScreen() .

The FlipBuffers() function must always be called outside a StartDrawing() : ... : StopDrawing() program block.

123.32 IsScreenActive

Syntax

```
Result = IsScreenActive()
```

Description

Games and full screen applications using PureBasic functions run under a multitasking environment (AmigaOS, Windows or Linux). This means that the user can switch back from full screen to the normal desktop. This change can be detected with this function and appriopriate actions should be taken, such as ReleaseMouse() , pause the game, stop the sounds etc. If the 'Result' is 0, then the screen is no longer active, otherwise the screen is active. This function must be called after FlipBuffers() , as the events are handled inside FlipBuffers() .

123.33 ScreenID

Syntax

```
Result = ScreenID()
```

Description

Returns the OS ScreenID. On Windows this is a normal WindowID so any function which needs such an ID can use it, like PlayMovie() for example.

Supported OS

Windows (DirectX), Linux (SDL)

123.34 SetFrameRate

Syntax

```
SetFrameRate(FrameRate)
```

Description

Set the frame rate (in frames per second) for the current screen. This is especially useful for windowed screen mode where there is no refresh rate for the screen. This function sets the maximum number of times per second that the FlipBuffers() function is called.

123.35 OpenScreen

Syntax

```
Result = OpenScreen(Width, Height, Depth, Title$ [, FlipMode [,
    RefreshRate])
```

Description

Opens a new screen according to the specified 'Width', 'Height' and 'Depth'. The opened screen is created with 2 video buffers to allows you to do double buffering, especially useful

for games. The buffers can be manipulated with the FlipBuffers() function. If the screen fails to open the result is '0'.

Width and Height should be standard combinations: 640*480, 800*600, 1024*768...

Depth can take the following values:

```
4 :    16 colors, independant palette available
8 :   256 colors, independant palette available
16: 65000 colors, fixed palette
24: 16 Mo colors, fixed palette
32: 16 Mo colors, faster than 24 bit mode
```

The optional 'FlipMode' parameter allows to wait or not for the screen synchronization before flipping the buffers with FlipBuffers() (also known as 'Vertical blank synchronization'). It can have one of the following values:

```
#PB_Screen_NoSynchronization   : disable synchronization
#PB_Screen_WaitSynchronization : enable synchronization
(default value)
#PB_Screen_SmartSynchronization: enable synchronization,
with a CPU saver routine when the program doesn't consume
                                all the CPU time (full
screen mode only)
```

Waiting for the screen synchronization allows the flip to be perfect (no 'tearing' or other visible artifacts) because the flip is performed when the screen has been fully drawed (and when the screen spot is outside of visible screen area). This also link the flip frequence to the actual screen refresh, ie: for 60Hz screen it could have at most 60 flip per seconds, etc.

An optional 'RefreshRate' (in Hz) can be set for the new screen. If it can't be handled, then OpenScreen() will fail.
Note: on Windows, the refresh rate could be locked or forced by the video card drivers, so it could be inaccurate.

Note: on Windows, the 'Title$' parameter will be displayed in the taskbar, so it's recommended to use a title related to the application name. Also, the Requester functions cannot be used on screens created with OpenScreen.

To open a screen area on a regular window, see OpenWindowedScreen() .

123.36 OpenWindowedScreen

Syntax

```
Result = OpenWindowedScreen(WindowID, x, y, Width, Height,
    AutoStretch, RightOffset, BottomOffset [, FlipMode])
```

Description

Open a new screen area according to given parameters on the given Window, which must be opened before using OpenWindow() . The "windowed screen" is able to use the hardware acceleration the same way than full-size OpenScreen() function.

If AutoStretch is 1, then the screen area will automatically resize (and its contents will be zoomed to new screen size) when the window size changes.

RightOffset and BottomOffset are used to define a margin on the right and bottom of the window (for statusbar for example).

The optional 'FlipMode' parameter allows to wait or not for the screen synchronization before flipping the buffers with FlipBuffers() (also known as 'Vertical blank synchronization'). It can have one of the following values:

```
#PB_Screen_NoSynchronization    : disable synchronization
#PB_Screen_WaitSynchronization : enable synchronization
(default value)
#PB_Screen_SmartSynchronization: enable synchronization,
with a CPU saver routine when the program doesn't consume
    all the CPU time (full screen mode only)
```

Waiting for the screen synchronization allows the flip to be perfect (no 'tearing' or other visible artifacts) because the flip is performed when the screen has been fully drawed (and when the screen spot is outside of visible screen area). This also link the flip frequence to the actual screen refresh, ie: for 60Hz screen it could have at most 60 flip per seconds, etc.

The opened screen is created with 2 video buffers to allow you to do double buffering, especially useful for games. The buffers can be manipulated with the FlipBuffers() function.

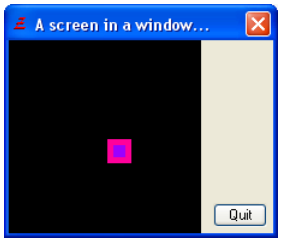
If the screen fails to open, the result is '0'.

Note: Only one windowed screen can be opened at one time.

Example 1: (fixed screen size with gadgets)

```
If InitSprite() = 0
  MessageRequester("Error", "Can't open screen & sprite enviroment!", 0)
  End
EndIf

If OpenWindow(0, 0, 0, 220, 160, "A screen in a window...", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  ButtonGadget(0, 170, 135, 45, 20, "Quit")

  If OpenWindowedScreen(WindowID(0), 0, 0, 160, 160, 0, 0, 0)
    CreateSprite(0, 20, 20)
    If StartDrawing(SpriteOutput(0))
      Box(0, 0, 20, 20, RGB(255, 0, 155))
      Box(5, 5, 10, 10, RGB(155, 0, 255))
      StopDrawing()
    EndIf
  Else
    MessageRequester("Error", "Can't open windowed screen!", 0)
    End
  EndIf
EndIf

direction = 2
Repeat
  ; It's very important to process all the events remaining in the queue at each frame
  ;
  Repeat
    Event = WindowEvent()

    Select Event
      Case #PB_Event_Gadget
        If EventGadget() = 0
          End
        EndIf

      Case #PB_Event_CloseWindow
        End
    EndSelect
  Until Event = 0
```

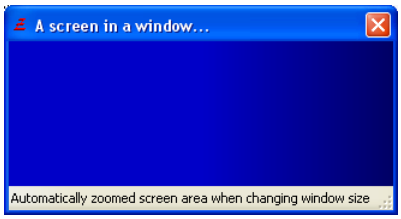
```

    FlipBuffers()
    ClearScreen(RGB(0, 0, 0))
    DisplaySprite(0, x, x)
    x + direction
    If x > 140 : direction = -2 : EndIf
    If x < 0   : direction =  2 : EndIf
    Delay(1)
  ForEver
```



Example 2: (screen with enabled auto-stretch and bottom-offset feature)

```
If InitSprite() = 0
  MessageRequester("Error", "Can't open screen & sprite enviroment!", 0)
  End
EndIf

If OpenWindow(0, 0, 0, 320, 200, "A screen in a window...", #PB_Window_SystemMenu | #PB_Window_SizeGadget | #PB_Window_ScreenCentered)
  CreateStatusBar(0, WindowID(0))
    AddStatusBarField(320)

  StatusBarText(0, 0, "Automatically zoomed screen area when changing window size...")

  If OpenWindowedScreen(WindowID(0), 0, 0, 320, 200, 1, 0, 20)

    CreateSprite(0, 50, 50) ; Create an empty sprite, will be whole black

    Repeat
      ; It's very important to process all the events remaining in the queue at each frame
      ;
      Repeat
        Event = WaitWindowEvent(10)

        If Event = #PB_Event_CloseWindow
          End
        EndIf
      Until Event = 0

      FlipBuffers()
      ClearScreen(RGB(0, 0, 200)) ; A blue background

      DisplaySprite(0, 10, 10)  ; Display our black box at the left-top corner
      DisplaySprite(0, 260, 10) ; Display our black box at the right-top corner
```

```
      ForEver

    Else
      MessageRequester("Error", "Can't open windowed screen!", 0)
    EndIf
  EndIf
```



For a more detailed example look at

123.37 ScreenOutput

Syntax

```
OutputID = ScreenOutput()
```

Description

Returns the OutputID of the currently used screen to perform 2D rendering operations on it. It will use the PureBasic 2DDrawing library (see StartDrawing())

123.38 SpriteOutput

Syntax

```
OutputID = SpriteOutput(#Sprite)
```

Description

Returns the OutputID of the specified sprite to perform 2D rendering operations on it. It will use the PureBasic 2DDrawing library (see StartDrawing()).

Note: It's not possible to use SpriteOutput() (and therefore the 2DDrawing functions) with a '#PB_Sprite_Alpha' sprite.

This command is not supported for the OpenGL subsystem .

123.39 TransparentSpriteColor

Syntax

```
TransparentSpriteColor(#Sprite, Color)
```

Description

Change the sprite transparent color (when displayed with DisplayTransparentSprite()). RGB() can be used to get a valid color value. A table with common colors is available here .

If #PB_Default is used as #Sprite value, then the default color (normally black (0,0,0)) is changed to the new given one and all future loaded sprites (with LoadSprite()) will use this color as the transparent one.

123.40 UseBuffer

Syntax

```
UseBuffer(#Sprite)
```

Description

Changes the default rendering buffer (usually the screen) to the given #Sprite. This means than all following DisplaySprite() like functions will be rendered on the #Sprite.

To get back on the main screen buffer, use #PB_Default as #Sprite value.

Supported OS

Windows (DirectX), Linux (SDL)

123.41 ExamineScreenModes

Syntax

```
Result = ExamineScreenModes()
```

Description

Starts to examine the available screen modes on the local computer. The screen modes list can be retrieved with the help of the NextScreenMode() function. If the Result is 0, the screen modes can't be listed.

123.42 NextScreenMode

Syntax

```
Result = NextScreenMode()
```

Description

This function should be called after ExamineScreenModes() . It will go step by step into the screen modes list. The current screen mode informations can be retieved with the following functions: ScreenModeWidth() , ScreenModeHeight() , ScreenModeDepth() and ScreenModeRefreshRate() . If the 'Result' is 0, the screen mode listing is finished.

Example:

```
InitSprite()

If ExamineScreenModes()
  While NextScreenMode()
    Debug
 Str(ScreenModeWidth())+"x"+Str(ScreenModeHeight())+"x"+Str(ScreenModeDepth())
  Wend
EndIf
```

123.43 ScreenModeDepth

Syntax

```
Depth = ScreenModeDepth()
```

Description

Returns the depth of the current screenmode listed with ExamineScreenModes() and NextScreenMode() functions. Depth can be either 8, 15, 16, 24 or 32 bits depending on the graphic card.

123.44 ScreenModeHeight

Syntax

```
Height = ScreenModeHeight()
```

Description

Returns the height (in pixels) of the current screenmode listed with ExamineScreenModes() and NextScreenMode() functions.

123.45 ScreenModeRefreshRate

Syntax

```
RefreshRate = ScreenModeRefreshRate()
```

Description

Returns the refreshrate (in hertz) of the current screenmode listed with ExamineScreenModes() and NextScreenMode() functions.

Note: On linux, with the SDL subsystem, this command always returns 0.

123.46 ScreenModeWidth

Syntax

```
Width = ScreenModeWidth()
```

Description

Returns the width (in pixels) of the current screenmode listed with ExamineScreenModes() and NextScreenMode() functions.

Chapter 124

Sprite3D

The Sprite3D library is a small 3D engine used to display 2D sprites while using the new 3D hardware capability like real-time zooming, transform, blending (transparency) and more. To know more about sprites, just read the 'Sprite' documentation. A new graphics card is highly recommended to get the full power of these functions (with 3D acceleration). 3D Sprites are provided to allow uncommon effects to be achieved easily but remember than they could be more restrictive than regular sprite (regarding the size for example).

Windows: DirectX 9 is used for sprite handle which allow to use the hardware blitter chip if available. These library functions don't use the OGRE 3D-Engine , so their relating DLL's are not needed. There is two alternative subsystems available on Windows: 'OpenGL' and 'DirectX7', which could be useful for better compatibility with older Windows. Note: DirectX7 is not supported on the 64 bit edition of Windows.

MacOS X: OpenGL is used to manage the sprites. This makes it possible to use hardware acceleration if available.

124.1 CreateSprite3D

Syntax

```
Result = CreateSprite3D(#Sprite3D, #Sprite)
```

Description

Create a 3D sprite with the specified 2D sprite. If #PB_Any is used as '#Sprite3D' parameter, the new sprite3d number will be returned as 'Result'. A 3D sprite is only 4 points (a rectangle) where a texture will be mapped on it. You can't draw directly on a 3D sprite.

The texture is in fact a regular sprite loaded by LoadSprite() or CatchSprite() or created by CreateSprite() using the #PB_Sprite_Texture mode. The texture must be a square of one of the following dimensions: 16*16, 32*32, 64*64, 128*128 or 256*256. Other sizes may work on some specific cards, but not all.

If the 'Result' is 0, then the 3D Sprite can't be created, else it's ok.

The 2D sprite used to create the 3D sprite must not be deleted when a 3D sprite use it. A single 2D Sprite can be used by any number of 3D Sprites.

124.2 FreeSprite3D

Syntax

```
FreeSprite3D(#Sprite3D)
```

Description

Free the specified sprite from memory. The associated 2D Sprite isn't freed.

Note: all remaining 3D sprites are automatically freed when the program ends.

124.3 DisplaySprite3D

Syntax

```
DisplaySprite3D(#Sprite3D, x, y [, Transparency])
```

Description

Display the specified #Sprite3D at the given (x,y) coordinates. A Transparency value is used and should be between 0 and 255. 0 means invisible, 255 means opaque. Any values in this range can be used. For better rendering, a 32-bit screen mode is strongly recommended for smooth fade-in and fade-out. With Sprite3DQuality() you can adjust the rendering mode of the 3D sprite if needed.

To display a sprite with alpha-channel support, the sprite which has been chosen to create the 3D sprite should have the #PB_Sprite_AlphaBlending flag specified.

Start3D() must be called before using this function. If several 3D sprites should be displayed, it's much better to display them all in one time. Example:

```
Start3D()
  DisplaySprite3D(1, 100, 100, 128)
  DisplaySprite3D(2, 100, 150, 100)
; ...
Stop3D()
```

124.4 InitSprite3D

Syntax

```
Result = InitSprite3D()
```

Description

Try to initialize the Sprite3D environment. If the 'Result' is 0, then a 3D hardware (or emulation) is not present. InitSprite() must be called before this function.

124.5 IsSprite3D

Syntax

```
Result = IsSprite3D(#Sprite3D)
```

Description

Tests if the given #Sprite3D number is a valid and correctly initialized sprite3D. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a sprite3D is ready to use.

124.6 RotateSprite3D

Syntax

```
RotateSprite3D(#Sprite3D, Angle.f, Mode)
```

Description

Rotates the specified #Sprite3D to the given 'Angle'. The 'Angle' value can be from 0 to 360. 'Mode' value take the following values:

```
#PB_Absolute: the angle is set to the new angle.
#PB_Relative: the angle is added to the previous #Sprite3D
 angle.
```

124.7 Sprite3DBlendingMode

Syntax

```
Sprite3DBlendingMode(SourceMode, DestinationMode)
```

Description

Changes the way of Sprite3D are blended with the background (when using DisplaySprite3D() and a transparency level). Both source and destination mode can take a value from 1 to 13. This function is for advanced users only. Here is the DirectX list of the corresponding values :

```
#D3DBLEND_ZERO            = 1
#D3DBLEND_ONE             = 2
#D3DBLEND_SRCCOLOR        = 3
#D3DBLEND_INVSRCCOLOR     = 4
#D3DBLEND_SRCALPHA        = 5
#D3DBLEND_INVSRCALPHA     = 6
#D3DBLEND_DESTALPHA       = 7
#D3DBLEND_INVDESTALPHA    = 8
#D3DBLEND_DESTCOLOR       = 9
#D3DBLEND_INVDESTCOLOR    = 10
#D3DBLEND_SRCALPHASAT     = 11
#D3DBLEND_BOTHSRCALPHA    = 12
#D3DBLEND_BOTHINVSRCALPHA = 13
```

Note: the default values are Sprite3DBlendingMode(#D3DBLEND_SRCALPHA, #D3DBLEND_INVSRCALPHA).

124.8 Sprite3DQuality

Syntax

```
Sprite3DQuality(Quality)
```

Description

Changes the way of Sprite3D are rendered. 'Quality' can take the following values:

```
#PB_Sprite3D_NoFiltering      : No filtering (faster but ugly
 when zooming/rotating)
#PB_Sprite3D_BilinearFiltering: Bilinear filtering (slower
 but very clean when zooming/rotating)
```

The default 'Quality' value is #PB_Sprite3D_NoFiltering.

124.9 Start3D

Syntax

```
Result = Start3D()
```

Description

Start the 3D engine and setup it to display the 3D Sprites. This function must be called before using DisplaySprite3D() and Stop3D() must be called when all the rendering are done. If the 'Result' is 0, the initialize has failed and 3D sprites functions can't be used.

Note: Inside a Start3D() : Stop3D() block there should be no functions of the standard Sprite library (e.g. like DisplaySprite()), only functions of the Sprite3D library. The code inside the block should also be finished with a FlipBuffers() .

Example:

```
1  For a = 0 To 255
2    DisplaySprite(10, 0, 0)        ; display background image
3    Start3D()
4      DisplaySprite3D(0, 0, 0, a)
5    Stop3D()
6    FlipBuffers()
7  Next
```

124.10 Stop3D

Syntax

```
Stop3D()
```

Description

Stop the 3D engine. It's typically called after rendering the last 3D sprite. Start3D() must be called before using this function.

124.11 TransformSprite3D

Syntax

```
TransformSprite3D(#Sprite3D, x1, y1, [z1], x2, y2, [z2], x3,
    y3, [z3], x4, y4, [z4])
```

Description

Transforms the #Sprite3D to the new given coordinates. This is typically used to perform real time transformations. Warning, as a Sprite3D is 2 vertex (2 triangles), the transformation could looks very strange... If one of the optional 'z' parameter is specified, all need to be specified.

```
;
; x1          x2
;    ---------
;    |       //|
;    |     /  |
;    |  /     |
```

```
;    //        /
;    ---------
; x4           x3
;
```

124.12 ZoomSprite3D

Syntax

```
ZoomSprite3D(#Sprite3D, Width, Height)
```

Description

Zooms the specified #Sprite3D from the given dimension.

Chapter 125

StatusBar

A 'StatusBar' is the bottom bar of a window where some informations are displayed on it. This bar is always visible and can be split in several parts.

125.1 AddStatusBarField

Syntax

```
AddStatusBarField(Width)
```

Description

Add a field to the current statusbar previously created with CreateStatusBar() . Each new field is created after the old one.

'Width' is the width of the new field in pixels. If 'Width' is set to #PB_Ignore then the field will be autosized to fill the remaining free space on the statusbar. Multiple fields can have a width of #PB_Ignore, in which case the free space will be devided evenly among these fields.

The following commands can be used to set or change the content of the statusbar field:
- StatusBarText()
- StatusBarImage()
- StatusBarProgress()

An example of using AddStatusBarField() you can see at CreateStatusBar() .

125.2 CreateStatusBar

Syntax

```
Result = CreateStatusBar(#StatusBar, WindowID)
```

Description

Create and add an empty #StatusBar to the specified WindowID. If #PB_Any is used as '#StatusBar' parameter, the new statusbar number will be returned as 'Result'. If the Result is 0, the StatusBar creation has failed, else it's fine. Once the bar is created, AddStatusBarField() can be used to setup the different parts of the bar.

Example:

```
If OpenWindow(0, 0, 0, 440, 50, "StatusBar",
 #PB_Window_SystemMenu|#PB_Window_ScreenCentered|#PB_Window_SizeGadget)
  If CreateStatusBar(0, WindowID(0))
    AddStatusBarField(90)
    AddStatusBarField(100)
    AddStatusBarField(#PB_Ignore) ; autosize this field
    AddStatusBarField(100)
  EndIf

  StatusBarText(0, 0, "Area normal")
  StatusBarText(0, 1, "Area borderless",
 #PB_StatusBar_BorderLess)
  StatusBarText(0, 2, "Area right & raised",
 #PB_StatusBar_Right | #PB_StatusBar_Raised)
  StatusBarText(0, 3, "Area centered", #PB_StatusBar_Center)

  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

125.3 FreeStatusBar

Syntax

```
FreeStatusBar(#StatusBar)
```

Description

Free the given status bar.

Note: all remaining status bars are automatically freed when the program ends.

125.4 IsStatusBar

Syntax

```
Result = IsStatusBar(#StatusBar)
```

Description

Tests if the given #StatusBar number is a valid and correctly initialized status bar. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a status bar is ready to use.

125.5 StatusBarImage

Syntax

```
StatusBarImage(#StatusBar, Field, ImageID [, Appearance])
```

Description

Set the specified Field to display an image. ImageID can be easily obtained by using LoadImage() or ImageID() . 'Appearance' is optional and can be used to alter the look of the field with the following values:

```
#PB_StatusBar_Raised    : raised borders
#PB_StatusBar_BorderLess: without border
#PB_StatusBar_Center    : center the icon in the field
#PB_StatusBar_Right     : align the icon to the right of the
 field
```

Example:

```
If OpenWindow(0, 0, 0, 340, 50, "StatusBarImage",
 #PB_Window_SystemMenu|#PB_Window_ScreenCentered|#PB_Window_SizeGadget)
  If CreateStatusBar(0, WindowID(0))
    AddStatusBarField(120)
    AddStatusBarField(170)
  EndIf

  If LoadImage(0, "cube16.ico")     ; change path/filename to
 your own image (or icon)
    StatusBarImage(0, 0, ImageID(0))
    StatusBarImage(0, 1, ImageID(0),
 #PB_StatusBar_Right|#PB_StatusBar_Raised)
  EndIf

  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

125.6 StatusBarID

Syntax

```
StatusBarID = StatusBarID(#StatusBar)
```

Description

Returns the unique system identifier of the #StatusBar.

This result is sometimes also known as 'Handle'. Look at the extra chapter Handles and IDs for more information.

125.7 StatusBarText

Syntax

```
StatusBarText(#StatusBar, Field, Text$ [, Appearance])
```

Description

Set the text for the specified 'Field' in the given '#StatusBar'. 'Appearance' is optional and can be used to alter the look of the field with the following values:

```
#PB_StatusBar_Raised     : raised borders
#PB_StatusBar_BorderLess: without border
#PB_StatusBar_Center     : center the text in the field
#PB_StatusBar_Right      : align the text to the right of the
 field
```

The options can be combined together with the '||' (OR) operator :

```
#PB_StatusBar_BorderLess | #PB_StatusBar_Right : Borderless
 right aligned field.
```

For a working example look at CreateStatusBar() .

125.8 StatusBarProgress

Syntax

```
StatusBarProgress(#StatusBar, Field, Value [, Appearance [,
   Min, Max]])
```

Description

Display a progress bar in the specified 'Field' in the given '#StatusBar'. The 'Value' parameter specifies the current progress (relative to the current minimum and maximum). To update this value simply call this command again with a different value.

'Appearance' is optional and can be used to alter the look of the field with the following values:

```
#PB_StatusBar_Raised     : raised borders
#PB_StatusBar_BorderLess: without border
```

The 'Min' and 'Max' parameter specify the boundaries of the progress bar. If they are not specified or have the value #PB_Ignore then the boundaries from a previous call to this command will be used. The default 'Min' and 'Max' for newly created statusbar fields are 0 and 100.

Example:

```
1   If OpenWindow(0, 0, 0, 340, 50, "StatusBarProgress",
     #PB_Window_SystemMenu | #PB_Window_ScreenCentered |
     #PB_Window_SizeGadget)
2    If CreateStatusBar(0, WindowID(0))
3      AddStatusBarField(120)
4      AddStatusBarField(170)
5    EndIf
6
7    StatusBarText(0, 0, "ProgressBar !")
8    StatusBarProgress(0, 1, 25)
9
10   Repeat
11   Until WaitWindowEvent() = #PB_Event_CloseWindow
12  EndIf
```

125.9 StatusBarHeight

Syntax

```
Result = StatusBarHeight(#StatusBar)
```

Description

Returns the height in pixel of the #StatusBar. This is useful for correct calculation on window height when using a statusbar.

Chapter 126

String

Strings are the method used in order to store a list of characters. With the functions supplied in this library, many essential actions may be preformed upon strings.

126.1 Asc

Syntax

```
Ascii = Asc(String$)
```

Description

Return the Ascii value of the first character in the string.

Example:

```

ValueASCII = Asc("!") ; 'ValueASCII' will get the value '33'
```

Note: It is also possible to obtain the ASCII value of a character (and not of a chain) while placing it between apostrophes.

Example:

```

ValueASCII = '!' ; 'ValueASCII' will get the value '33'
```

Note: A table with all Ascii values and their relating figures may be found here .

126.2 Bin

Syntax

```
Result\$ = Bin(Value.q [, Type])
```

Description

Convert a quad numeric number into a string, in binary format.

Example:

```
Debug Bin(32) ; Will display "100000"
Debug Bin(1024*1024*1024*1024) ; Will display
 "1000000000000000000000000000000000000000000"
```

Note: If leading zero are needed in the output string, use the RSet() function as follows:

```
Debug RSet(Bin(32), 16, "0") ; Will display "0000000000100000"
```

The optional parameter 'Type' may be used, if the 'Value' should be treated as another type. 'Type' may be:

```
#PB_Byte    : The value is a byte number, ranging from 0 to 255
#PB_Ascii   : The value is a ascii character, ranging from 0
 to 255
#PB_Word    : The value is a word number, ranging from 0 to
 65536
#PB_Unicode: The value is a unicode character, ranging from 0
 to 65536
#PB_Long    : The value is a long number, ranging from 0 to
 4294967296
#PB_Quad    : The value is a quad number, ranging from 0 to
 18446744073709551615
```

See also Str() , Val() and Hex() .

126.3 Chr

Syntax

```
Text\$ = Chr(ASCII-Value)
```

Description

Returns the character associated with the given ASCII value.

Example:

```
Debug Chr(33) ; Will display "!"
```

Note: A table with all Ascii values and their relating figures may be found here .

126.4 CountString

Syntax

```
Result = CountString(String$, StringToCount$)
```

Description

Returns the number of occurences of StringToCount$ found in String$. This is not word based, which means than if the 'StringToCount$' is a part of a word, it will be counted as well, as shown in the following example.

```
Debug CountString("How many 'ow' contains Bow ?", "ow") ; will display 3
```

126.5 FindString

Syntax

```
Position = FindString(String$, StringToFind$, StartPosition)
```

Description

Try to find the 'StringToFind$' within the given 'String$' starting from the given 'StartPosition'. If the string is found, its position is returned (in its offset, starting from 1). If the string is not found, it returns 0.

```
Debug FindString("PureBasic", "Bas", 1) ; will display 5
```

126.6 Hex

Syntax

```
Result\$ = Hex(Value.q [, Type])
```

Description

Convert a quad numeric number into a string, in hexadecimal format.

Example:

```
Debug Hex(12) ; Will dipslay "C"
Debug Hex(1234567890) ; Will display "0499602D2"
```

Note: If leading zero are needed in the output string, use the RSet() function as follows:

```
Debug RSet(Hex(12), 4, "0") ; Will display "000C"
```

The optional parameter 'Type' may be used, if the 'Value' should be treated as another type.

'Type' may be:

```
#PB_Byte    : The value is a byte number, ranging from 0 to 255
#PB_Ascii   : The value is a ascii character, ranging from 0 to 255
#PB_Word    : The value is a word number, ranging from 0 to 65536
#PB_Unicode: The value is a unicode character, ranging from 0 to 65536
#PB_Long    : The value is a long number, ranging from 0 to 4294967296
#PB_Quad    : The value is a quad number, ranging from 0 to 18446744073709551615
```

Example:

```

Debug Hex(-1)
Debug Hex(-1, #PB_Byte)
Debug Hex(-1, #PB_Word)
Debug Hex(-1, #PB_Long)
Debug Hex(-1, #PB_Quad)     ; quad value is the standard
```

See also Str() , Val() and Bin() .

126.7 InsertString

Syntax

```
Result\$ = InsertString(String$, StringToInsert$, Position)
```

Description

Insert 'StringToInsert$' into 'String$' at the specified 'Position'. The first 'Position' index is 1.

Example:

```

Debug InsertString("Hello !", "World", 7)  ; Will display "Hello World!"
Debug InsertString("Hello !", "World", 1)  ; Will display "WorldHello !"
```

126.8 LCase

Syntax

```
Result\$ = LCase(String$)
```

Description

Returns the original string converted into lower case characters (if possible). This function also supports accent letters, so if a upper 'É' is found, it will be transformed into 'é'.

Example:

```

Debug LCase("This is Art") ; Will display "this is art"
```

126.9 Left

Syntax

```
Result\$ = Left(String$, Length)
```

Description

Returns the first 'Length' number of characters from the left side of the string. This function is safe to use even if out-of-bound values are given for the 'Length' parameter, in that case, it will simply return the result which matches best.

Example:

```

Debug Left("This is Art",4) ; Will display "This"
```

126.10 Len

Syntax

```
Length = Len(String$)
```

Description

Returns the Character length of the string.

Example:

```

Debug Len("This is Art") ; will display 11
```

126.11 LSet

Syntax

```
Result\$ = LSet(String$, Length [, Character$])
```

Description

Pads a string to the left, and adds 'space' character to fit the specified length. The optional 'Character$' parameter may be used to replace 'space' with another character. If the string is longer than the specified 'Length', it will be truncated to 'Length' characters, starting from the left side of the string.

Example:

```

Debug LSet("L", 8)           ; will display "L       "
Debug LSet("L", 8, "-")      ; will display "L-------"
Debug LSet("LongString", 4) ; will display "Long"
```

126.12 LTrim

Syntax

```
Result\$ = LTrim(String$ [, Character$])
```

Description

Removes all the 'space' characters located in the front of a string. If the optional 'Character$' parameter is specified, it removes all the 'Character$' located in the front of a string. 'Character$' has to be a one character length string.

Example:

```

Debug LTrim("     This is Art")  ; Will display "This is Art"
Debug LTrim("!!Hello Word", "!") ; Will display "Hello World"
```

126.13 Mid

Syntax

```
Result\$ = Mid(String$, StartPosition [, Length])
```

Description

Extract a string of specified 'Length' from the given 'String$'. 'StartPosition' (minimum value = 1) specifies which character position to start extracting from. 'Length' is optional and specifies how many characters from that position you want to extract. If 'Length' is omitted, characters are extracted until the end of string.

Example:

```

Debug Mid("Hello", 2) ; Will display "ello"
Debug Mid("Hello", 2, 1) ; Will display "e"
```

126.14 RemoveString

Syntax

```
String\$ = RemoveString(String$, StringToRemove$ [, Mode [,
    StartPosition [, NbOccurences]]])
```

Description

Finds all occurrences of 'StringToRemove$' within the given 'String$' and removes them. 'StartPosition' is the first character of the string to start the removing. First character position index is 1. If the optional 'NbOccurences' parameters is specified, it only removes the given 'StringToRemove$' up to the 'NbOccurences' number.

'Mode' is an optional parameter and may be a combination of the following values:

```
#PB_String_NoCase: Case insensitive remove (A=a). By default
 the remove is case sensitive.
```

Example:

```

Debug RemoveString("This is Art", "is") ; Will display "Th Art"
Debug RemoveString("This is Art", "is", 0, 1, 1) ; Will display "Th is Art"
```

126.15 ReplaceString

Syntax

```
String\$ = ReplaceString(String$, StringToFind$, StringToReplace$ [, Mode [, StartPosition]])
```

Description

Try to find any occurrences of 'StringToFind$' in the given 'String$' and replace them with 'StringToReplace$'. An optional 'StartPosition' parameter may be specified to tell at which position the replace should start. First 'StartPosition' index is 1.

'Mode' is optional and may be a combination of the following values:

```
#PB_String_NoCase : Case insensitive search (A=a). By default the search is case sensitive.
#PB_String_InPlace: In place replacing. This means that the string is replaced directly in the memory.
The 'StringToFind$' and 'StringToReplace$' parameter must have the same length ! This is
a dangerous option, for advanced users only. The advantage is the very high speed of the
replacement. When using this option, the result of ReplaceString() has to be ignored (and
not affected to something, as this is the string passed in parameter which is changed).
```

Example:

```

Debug ReplaceString("This is Art", " is", " was") ; Will display "This was Art"
Debug ReplaceString("Hello again, hello again", "HELLO", "oh no...", 1, 10) ; Will display "Hello again, oh no... again"

test$ = "Bundy, Barbie, Buddy"
ReplaceString(test$, "B", "Z", 2, 1)  ; all B gets changed to Z  (directly in memory, no valid return value here)
Debug test$   ; Output of the changed string
```

126.16 Right

Syntax

```
Result\$ = Right(String$, Length)
```

Description

Returns the last 'Length' number of characters from the right side of the string. This function is safe to use even if out-of-bound values are given for the 'Length' parameter, in that case, it will simply return the result which matches best.

Example:

```

Debug Right("This is Art", 3) ; Will display "Art"
```

126.17 RSet

Syntax

```
Result\$ = RSet(String$, Length [, Character$])
```

Description

Pads a string to the right, and adds 'space' characters to fit the specified length. The optional 'Character$' parameter may be used to replace 'space' with another character. If the string is longer than the specified 'Length', it will be truncated to 'Length' characters, starting from the left side of the string.

Example:

```

Debug RSet("R", 8)           ; will display "       R"
Debug RSet("R", 8, "-")      ; will display "-------R"
Debug RSet("LongString", 4) ; will display "Long"
```

126.18 RTrim

Syntax

```
Result\$ = RTrim(String$ [, Character$])
```

Description

Removes all the 'space' characters located at the end of a string. If the optional 'Character$' parameter is specified, it removes all the 'Character$' located at the end of a string. 'Character$' has to be a one character length string.

Example:

```

Debug RTrim("This is Art      ") ; Will display "This is Art"
Debug RTrim("Hello Word!!", "!") ; Will display "Hello World"
```

126.19 StringByteLength

Syntax

```
Result = StringByteLength(String$ [, Format])
```

Description

Returns the number of bytes required to store the string in memory in the given format in bytes. 'Format' may be #PB_Ascii, #PB_UTF8 or #PB_Unicode. If 'Format' is not specified, the mode of the executable (unicode or ascii) is used.

Note: The number of bytes returned does not include the terminating Null-Character of the string. The size of the Null-Character is 1 byte for Ascii and UTF8 mode and 2 bytes for Unicode mode.

Example:

```

Debug StringByteLength("ä", #PB_UTF8) ; will display 2
```

126.20 StringField

Syntax

```
Result\$ = StringField(String$, Index, Delimiter$)
```

Description

Returns the string field at the specified index. A one character 'Delimiter$' is used to split the string. The first 'Index' position is 1.

Example:

```

For k=1 To 6
  Debug StringField("Hello I am a split string", k, " ")
Next
```

126.21 StrF

Syntax

```
Result\$ = StrF(Value.f [, NbDecimal])
```

Description

Convert a float number into a string. A maximum number of decimal places may be specified. The number will be rounded, if 'NbDecimal' is smaller than existing decimal places of 'Value.f'.

Note: Signed integer numbers must be converted with Str() and unsigned numbers with StrU() .

Example:

```

value.f = 10.54
Debug "Result: " + StrF(value)     ; we do not use the 2nd parameter, so we get a full float number
Debug "Result: " + StrF(value,2)   ; we want a result with two decimal places, no rounding needed as we have only two
Debug "Result: " + StrF(value,0)   ; we want a result with no decimal places, so the value is rounded
```

126.22 StrD

Syntax

```
Result\$ = StrD(Value.d [, NbDecimal])
```

Description

Convert a double number into a string. A maximum number of decimal places may be specified. The number will be rounded, if 'NbDecimal' is smaller than existing decimal places of 'Value.d'.

Note: Signed integer numbers must be converted with Str() and unsigned numbers with StrU() .

Example:

```

Value.d = 10.54
Debug "Result: " + StrD(Value, 2)   ; we want a result with two decimal places, no rounding needed as we have only two

Debug "Result: " + StrD(Value, 0)   ; we want a result with no decimal places, so the value is rounded
```

126.23 Str

Syntax

```
Result\$ = Str(Value.q)
```

Description

Convert a signed quad number into a string.

Note: Floats must be converted with StrF() , doubles with StrD() and unsigned numbers with StrU() .

Example:

```

Value.q = 100000000000000001
Debug "Result: " + Str(Value)
```

See also Val() , Hex() and Bin() .

126.24 StrU

Syntax

```
Result\$ = StrU(Value, Type)
```

Description

Convert a unsigned numeric number into a string.

'Type' may be:

```
#PB_Byte    : The value is a byte number, ranging from 0 to 255
#PB_Ascii   : The value is a ascii character, ranging from 0
 to 255
#PB_Word    : The value is a word number, ranging from 0 to
 65536
#PB_Unicode: The value is a unicode character, ranging from 0
 to 65536
#PB_Long    : The value is a long number, ranging from 0 to
 4294967296
#PB_Quad    : The value is a quad number, ranging from 0 to
 18446744073709551615
```

Note: Signed integer numbers must be converted with Str() and float numbers with StrF() or StrD() .

Example:

```

byte.b = 255
Debug Str(byte) ; Will display -1
Debug StrU(byte, #PB_Byte) ; Will display 255
```

126.25 ReverseString

Syntax

```
Result\$ = ReverseString(String$)
```

Description

Reverse all the characters in the 'String$'. The last characters becomes the first characters, and vice-versa.

Example:

```

Debug ReverseString("Hello")  ; Will display "olleH"
```

126.26 Space

Syntax

```
Result\$ = Space(Length)
```

Description

Create an empty string of the given 'Length' filled with 'space' characters.

Example:

```

  Debug Space(5) ; Will display "     "
```

126.27 Trim

Syntax

```
Result\$ = Trim(String$ [, Character$])
```

Description

Removes all the 'space' characters located at the beginning and at the end of a string. If the optional 'Character$' parameter is specified, it removes all the 'Character$' located at the beginning and at the end of a string. 'Character$' has to be a one character length string.

Example:

```

  Debug Trim("   Hello     ")  ; Will display "Hello"
  Debug Trim("!!Hello!!", "!") ; Will display "Hello"
```

126.28 UCase

Syntax

```
Result\$ = UCase(String$)
```

Description

Returns the original string converted into upper case characters (if possible). This function also supports accent letters, so if a lower 'é' is found, it will be transformed into 'É'.

Example:

```

  Debug UCase("This is Art") ; Will display "THIS IS ART"
```

126.29 ValD

Syntax

```
Result.d = ValD(String$)
```

Description

Transform a string into a double value. The string must be a double in decimal format.

Note: Strings containing an integer can also be converted with Val() , and 32 bit floats with ValF() (with less accuracy than ValD()).

Example:

```

Debug ValD("10.000024") ; will display 10.000024.
```

126.30 ValF

Syntax

```
Result.f = ValF(String$)
```

Description

Transform a string into a float value. The string must be a float in decimal format.

Note: Strings containing an integer can also be converted with Val() and 64 bit floats with ValD() (with more accuracy than ValF()).

Example:

```

Debug ValF("10.24") ; Will display 10.24.
```

126.31 Val

Syntax

```
Result.q = Val(String$)
```

Description

Transform a string into a quad numeric value. The string may be an integer in decimal, hexadecimal (with '$' prefix) or binary (with '%' prefix) format.

Note: Strings containing a 32 bit floats may be converted with ValF() and 64 bit floats with ValD() .

Example:

```

Debug Val("1024102410241024") ; will print '1024102410241024'.
Debug Val("$10FFFFFFFF")      ; will print '73014444031'.
Debug Val("%1000")            ; will print '8'.
```

See also Str() , Hex() and Bin() .

Chapter 127

SysTray

SysTray is the right part of the taskbar which contains some icons and the date/hour. PureBasic provides full access to this area and the capability to add any number of images to it.

127.1 AddSysTrayIcon

Syntax

```
Result = AddSysTrayIcon(#SysTrayIcon, WindowID, ImageID)
```

Description

Add an icon in the SysTray area. If #PB_Any is used as the '#SysTrayIcon' parameter, the new systray icon number will be returned as 'Result'. WindowID must be a valid WindowID() . ImageID must be a previously loaded image (using LoadImage() or using CatchImage()). Icon image type (.ico) are only supported on Windows, for Linux it is advised to use a PNG image (in order to have a transparency layer). ImageID() may be used to get this ID easily.

When an event occurs on any of the SysTray icons the #PB_Event_SysTray event is returned. EventGadget() can be used to know which SysTrayIcon has been used. EventType() functions is also updated by this function.

Note: All SysTray icons are automatically removed when the program ends.

127.2 ChangeSysTrayIcon

Syntax

```
ChangeSysTrayIcon(#SysTrayIcon, ImageID)
```

Description

Change the specified icon in the SysTray area. ImageID must be a previously loaded image (using LoadImage() or using CatchImage()). Icon image type (.ico) are only supported on Windows, for Linux it is advised to use a PNG image (in order to have a transparency layer). ImageID() may be used to get this ID easily.

127.3 IsSysTrayIcon

Syntax

```
Result = IsSysTrayIcon(#SysTrayIcon)
```

Description

Tests if the given #SysTrayIcon number is a valid and correctly initialized systray icon. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a systray icon is ready to use.

127.4 SysTrayIconToolTip

Syntax

```
SysTrayIconToolTip(#SysTrayIcon, Text$)
```

Description

Associate the specified Text$ with the SysTray icon. Tooltip text is that text which is displayed when the mouse cursor hovers over the icon for a period of time (yellow floating box).

127.5 RemoveSysTrayIcon

Syntax

```
RemoveSysTrayIcon(#SysTrayIcon)
```

Description

Remove the specified SysTray icon.

Note: all remaining SysTray icons are automatically removed when the program ends.

Chapter 128

Terrain

Terrains are outdoor 3D scenes which simulate a realistic natural environment based on pre-calculated 2D maps. They are useful in many cases, for example 3D representations of land, simulation games and more. InitEngine3D() should be called successfully before using the Terrain functions.

128.1 CreateTerrain

Syntax

```
CreateTerrain(Filename$, MaterialID, [ScaleX, ScaleY, ScaleZ,
    [Precision]])
```

Description

Creates a new 3D terrain from the given 2D picture file. The picture can be in either JPEG, PNG or TGA format and has to be in 256 color mode (grayscale). Each pixel of the picture will be interpreted as the height of the terrain at this point. A fully black (R,G,B=0.0,0) pixel will be at a height of 0 and a full-scale white (R,G,B=255,255,255) pixel will be at a height of 255. Therefore it is possible to have up to 256 levels of height. If required, the terrain can be scaled when it is created using the ScaleX, ScaleY and ScaleZ optional parameters. For performance reasons, it is not possible to resize a terrain in realtime, only when it is created. The used picture must be a sqare, where x and y sizes are powers of 2 +1, with a minimum size of 65. Possible sizes can be 65x65, 129x129, 257x257 and so on. E.g. for a terrain with 64x64 grid of squares you need 65x65 pixels.

The specified material will be used by the terrain for rendering and can be easily got using the MaterialID() function. The material usually contains two texture layers: the first one is the texture which is applied globally to the terrain and the second layer is the detail texture, which is applied several times over the terrain (to get a more precise render). The example below shows the usual material setup for a terrain.

The last optional parameter controls the precision of rendering the terrain. As there are many calculations involved, this parameter has been introduced to tune the terrain display depending on the target computer. For example, you can set the rendering to be slower but more accurate on faster computers or faster but not as good looking for slower computers.

The terrain has to be created before any cameras .

Filename$ - A string containing the name of the picture file to create the terrain from.

MaterialID - The ID of the material to render the terrain in, which can be got using the MaterialID() function.

ScaleX - The factor to scale the terrain by in the X direction. A value of 1 means to keep the same size. A value of 2 means to double the size, 3 to triple the size and so on. Fractional values between 0 and 1 will cause the terrain to be decreased in size.

ScaleY - The factor to scale the terrain by in the Y direction. Values are the same as the ScaleX parameter.

ScaleZ - The factor to scale the terrain by in the Z direction. Values are the same as the ScaleX parameter.

Precision - This is optional, but can only be set if the Scale parameters have been used. Precision for the rendering of the terrain. Possible values range from 1 (best, but slowest) to 20 (fastest but suffers from polygon displacement when close up).

Example:

```
CreateMaterial  (0, LoadTexture(0, "Global.jpg"))     ; First layer is global
AddMaterialLayer(0, LoadTexture(1, "Detail.jpg"), 1) ; Second layer is detail
```

128.2 TerrainHeight

Syntax

```
Height = TerrainHeight(x, z)
```

Description

Returns the height value at the specified position in the terrain. The origin for terrain coordinates is (0,0) and is located at the top/left of the 2D picture. This function is mainly used to handle collision with the ground.

x - The horizontal position of the point from which to read the height

z - The depth of the point from which to read the height

Height - The height value at the specified position of the terrain.

Chapter 129

Texture

Textures are useful to have realistic looking meshes . It's possible to create a texture on fly (with regular 2DDrawing functions) or load it from disk. InitEngine3D() should be called successfully before using the Texture functions.

129.1 CreateTexture

Syntax

```
Result = CreateTexture(#Texture, Width, Height)
```

Description

Creates a new blank #Texture with the specified dimension (in pixels). If #PB_Any is used as '#Texture' parameter, the new texture number will be returned as 'Result'.

It's strongly recommended to make the texture square and with power of 2 width/height: 64x64, 128x128, 256x256... Old GFX cards can have limitation, so if possible limit the texture size to 256x256.

If the 'Result' is 0, the texture creation has failed. If the #Texture was already created, it is automatically freed and replaced by the new one.

Example:

```
1  CreateTexture(0, 256, 256) ; Creates a new 256x256 texture
```

129.2 FreeTexture

Syntax

```
FreeTexture(#Texture)
```

Description

Frees the specified #Texture. All its associated memory is released and this object can't be used anymore.

Note: all remaining textures are automatically freed when the program ends.

129.3 IsTexture

Syntax

```
Result = IsTexture(#Texture)
```

Description

Tests if the given #Texture is a valid and correctly initialized texture.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that a texture is ready to use.

129.4 LoadTexture

Syntax

```
Result = LoadTexture(#Texture, Filename$)
```

Description

Loads a new #Texture from the disk. If #PB_Any is used as '#Texture' parameter, the new texture number will be returned as 'Result'. Before loading a texture, an archive has to be specified with Add3DArchive() .

Texture format can be in PNG, TGA or JPG. It's strongly recommended to make the texture dimension square and with power of 2 width/height: 64x64, 128x128, 256x256... Old GFX cards can have limitation, so if possible limit the texture size to 256x256.

If the 'Result' is 0, the texture can't be loaded. If the #Texture was already created, it is automatically freed and replaced by the new one.

129.5 TextureID

Syntax

```
TextureID = TextureID(#Texture)
```

Description

Returns the TextureID of the specifiec #Texture. This is the unique texture identifier needed by other functions: AddMaterialLayer() , CreateMaterial() .

129.6 TextureHeight

Syntax

```
Height = TextureHeight(#Texture)
```

Description

Returns the height (in pixels) of the specified #Texture.

129.7 TextureWidth

Syntax

```
Width = TextureWidth(#Texture)
```

Description

Returns the width (in pixels) of the specified #Texture.

Chapter 130

Thread

A thread is a part of a program which runs asynchronously, in the background of this program. This means than it's possible to perform long operations (compression, image processing, etc) without halting the whole program and let the user continue to do other things. A thread runs within your program, it's not another process. When the main program exits, all the threads are destroyed. Under PureBasic, threads are simply a procedure which is called asynchronously. The thread runs until the procedure exits.

Examples of places of programs where threads are useful are when you need to be able to handle multiple situations with different response times or which occur at different intervals. In the above paragraph, the response times of image processing and the user interface are quite different (you would want to wait for the image to be processed but always have the user interface to respond).

PureBasic has a special compiler setting to create threadsafe executables. (/THREAD commandline switch or "create threadsafe executable" in the IDE compiler options). Without this mode, certain functions (and also the string access) are faster, but not save to use in threads. It is still possible to create threads without this mode but it is not recommended, as even something simple like a local string access can be dangerous and needs to be protected. Enabling this option makes these things save inside threads, but comes at the price of some speed. The decision on wether or not to use threads should therefore done with care, and the threadmode should only be used when there is a real need for it.

Note: Threads need to be used carefully because it is possible that you can have multiple access to shared resources (memory, variables, files, etc) and you need to manually ensure that you do run into trouble because of this. The Mutex functions in this library can be used to synchronize access to such shared resources.

Using the Threaded keyword it's possible to create thread-based persistent objects (variables, arrays, linkedlists, maps).

Note: Don't use DirectX inside threads (MS Windows limitation)! If you need to display graphics in threads use Images and 2DDrawing instead.

130.1 IsThread

Syntax

```
Result = IsThread(Thread)
```

Description

Tests if the given thread number is a valid thread created with the CreateThread() function, and if it is still running. It will return 0 if it is not a valid thread, or if it has allready ended.

130.2 ThreadID

Syntax

```
ThreadID = ThreadID(Thread)
```

Description

Returns the unique ID which identifies the given 'Thread' in the operating system. This function is very useful when another library needs a thread reference.

This result is sometimes also known as 'Handle'. Look at the extra chapter Handles and Numbers for more information.

130.3 CreateMutex

Syntax

```
Mutex = CreateMutex()
```

Description

Creates a new mutex object. The mutex is intially unlocked.

The main objective of the mutex functions is thread syncronisation. They do not create too much overhead, but they only work within one program, not system-wide. A mutex is an object wich can only be "owned" or locked by one thread at a time, so it is used to protect shared resources. Only the thread that has the mutex locked may access a certain file, memory area, ...
See LockMutex() and UnlockMutex() for locking/unlocking a mutex.

Example:

```
; Run this code once as it is. You will see that the lines printed are
; mixed from between the threads. Now uncomment the Mutex functions and
; the strings are printed in order, because only one thread at a time
; has the right to execute the printing functions.
;
Procedure WithoutMutex(*Number)
  Shared Mutex

  For a = 1 To 5
    ;LockMutex(Mutex)    ; uncomment this to see the difference

    PrintN("Thread "+Str(*Number)+": Trying to print 5x in a row:")
    For b = 1 To 5
      Delay(50)
      PrintN("Thread "+Str(*Number)+" Line "+Str(b))
    Next b

    ;UnlockMutex(Mutex) ; uncomment this to see the difference
  Next a
EndProcedure
```

```

OpenConsole()
Mutex = CreateMutex()

thread1 = CreateThread(@WithoutMutex(), 1)
Delay(25)
thread2 = CreateThread(@WithoutMutex(), 2)
Delay(25)
thread3 = CreateThread(@WithoutMutex(), 3)

WaitThread(thread1)
WaitThread(thread2)
WaitThread(thread3)

Input()
```

130.4 CreateThread

Syntax

```
Thread = CreateThread(@ProcedureName(), *Value)
```

Description

Creates a new thread running in the application background. If the thread is correctly created, it returns the Thread number which is used with the other thread functions, such as KillThread() , PauseThread() , etc. The procedure which you use as a thread must take one parameter and cannot return anything. The '*Value' argument of CreateThread() is passed as the parameter to the procedure. If you do try to return a value from your thread it will simply be lost.

@procedurename()- The address of the procedure you want to use as the code for the new thread. Remember to put the @ in front to get the name and the () afterwards so it gets the address of the procedure.

*Value - This is passed to the thread code as the parameter to your function. It is up to you to decide what this is used for.

The number for the newly created thread, or zero if a new thread could not be created. This number is required if you want to control the thread using the other functions in this library.

Example:

The example below shows the basic way to create a thread, although in this case it does not do anything.

```
Procedure YourProcedure(*Value)
  ; The variable '*Value' will contain 23
EndProcedure

CreateThread(@YourProcedure(), 23)
```

Example: Passing multiple parameters to a thread

```
Structure Person
  Name$
  Age.b
  Phone.l
EndStructure

Procedure Thread(*Parameters.Person)

  ; Display the parameters
  ;
  Debug *Parameters\Name$
  Debug *Parameters\Age
  Debug *Parameters\Phone

  ; Once we don't need them anymore, use ClearStructure() to ensure than every dynamic
  ; objects (if any) are correctly cleared, and release the dynamic memory block
  ClearStructure(*Parameters, Person)
  FreeMemory(*Parameters)

EndProcedure

; We use a dynamically allocated block, so even if we call it from a procedure, it will
; still work. The memory block will be freed by the thread, when
;
*Parameters.Person = AllocateMemory(SizeOf(Person))
*Parameters\Name$ = "John"
*Parameters\Age   = 30
*Parameters\Phone = 10203040

CreateThread(@Thread(), *Parameters) ; Send the thread a pointer to our structure

Delay(2000)
```

130.5 FreeMutex

Syntax

```
FreeMutex(Mutex)
```

Description

Frees a mutex object and the memory it requires.

The mutex object should be unlocked by the time it is freed and it may no longer be used after it was freed. To ensure this a mutex should only be freed after all threads that use it have either ended, or are otherwise sure not to use this mutex again.

130.6 KillThread

Syntax

```
KillThread(Thread)
```

Description

Imediately kills the specified thread, which had previously been created with CreateThread() . This is a very dangerous function, and should only be used rarely. The problem is that the thread is killed immediately and has no chance to perform any cleanup code (for example, freeing memory, releasing items, de-allocating its own stack).

If possible, a flag like a global variable should be used to tell the thread to quit itself (which does the needed cleanup) and this function should only be used if this is not possible for some reason.

Thread - The number of the thread that you want to kill. This value is returned by CreateThread() .

Example:

```
; A procedure/thread which will never exit. Not good, but it
; shows how KillThread works
Procedure PrintStuff(*Interval)
  Repeat
    PrintN(".")
    Delay(*Interval)
  ForEver
EndProcedure

If OpenConsole()
  thread = CreateThread(@PrintStuff(), 500)
  If thread
    For i=0 To 10
      PrintN("A")
      Delay(999)

      If i=5
        KillThread(thread)
      EndIf
    Next
  EndIf
EndIf
```

130.7 LockMutex

Syntax

```
LockMutex(Mutex)
```

Description

Waits until the mutex object is available (not locked by another thread) and then locks the object so no other thread can get a lock on the object.

After this function returns, it is assured that this thread is the only one with a locked state on the mutex. The thread can now freely access the shared resource that is protected by this mutex, as it is the only one with exclusive access to the mutex.

If another thread calls LockMutex() while this one has the lock, it will wait inside the LockMutex() function until this thread calls UnlockMutex() to release its lock on the mutex.

Note: Since the LockMutex() function waits until the mutex is available, it can easily lead to lockup situations if a UnlockMutex() call is forgotten.

See CreateMutex() for a code example.

130.8 PauseThread

Syntax

```
PauseThread(Thread)
```

Description

Pauses the execution of the specified thread, previously created with CreateThread() . The thread can be resumed with ResumeThread() .

Thread - The number of the thread you want to pause. This value is returned by CreateThread() .

Example:

```
Procedure PrintStuff(*Dummy)
  For i = 0 To 10
    PrintN(".")
    Delay(200)
  Next
EndProcedure

If OpenConsole()
  thread = CreateThread(@PrintStuff(), 0)
  If thread
    Delay(100)
    PauseThread(thread)
    For i = 0 To 10
      PrintN("A")
      Delay(50)
    Next

    ; Resume thread and give it enough time to complete
    ResumeThread(thread)
    Delay(3000)
  EndIf
EndIf
```

130.9 ResumeThread

Syntax

```
ResumeThread(Thread)
```

Description

Resumes execution of the specified thread, previously paused with PauseThread() .

Thread - The number of the thread you want to resume. This value is returned by CreateThread() .

See the PauseThread() function for an example code.

130.10 ThreadPriority

Syntax

```
OldPriority = ThreadPriority(Thread, Priority)
```

Description

Change the priority of the specified thread and returns the old priority.

The priority value can go from 1 to 32. 1 is the lowest priority available, 16 is the normal priority and 32 is the time critical priority (highest, please don't use it unless you know what you're doing).

If the priority is 0, then the priority isn't changed (useful to only retrieve the thread priority without change it). All threads are created with a normal priority.

Typically you would give a thread which is always running (for example, an image processing thread) a priority which is not greater than any other thread in your system. The reason for this is that if it is always running, and it has a high priority, then no other thread will get a chance to run. If a thread needs to be highly responsive but spends most of its time waiting for some event you might consider giving it a higher than normal priority.

Windows schedules threads (chooses which one to run for a short length of time) using a pre-emptive priority based scheduling strategy, with round robin scheduling within the same priorirty level. This means that if it is time for execution of a different thread (known as a context switch) then the thread with the highest priority that is available to run will be the next one to be executed. If there is more than one thread with the highest priority, and they are available to run, then each thread will be cycled through on successive context switches.

Thread - The number of the thread you want to change the priority of. This value is returned by CreateThread() .
Priority - The new priority you want to assign to the thread. The priority can be 0 (meaning not to change the priority) or range from 1 (lowest priority) to 32 (highest priority). 16 is normal priority. Windows doesn't support 32 different level of priority, here is the corresponding table:

```
- 1: lowest
- between 2 and 15: below normal
- 16:  normal
- between 17 and 30: above normal
- 31: highest
- 32: time critical
```

This function returns the priority of the thread before this function was called. This can be useful if you only want to boost the priority of the thread for a short time and then return it to its previous level. The returned value is not necessary the same than the one set with ThreadPriority(), as it depends of the granularity of the priority tuning offered by the system.

Example:

```
; Procedure which always runs (note, no Delay function as
; this would cause the thread to stop running while it was
; being delayed
Procedure PrintStuff(*Interval)
  For i = 0 To 1000000000
    ; Nasty busy wait
  Next
EndProcedure

If OpenConsole()
  thread = CreateThread(@PrintStuff(), 500)
  If thread
    ; Increase the priority above the main thread
    ; You should notice a delay before the print function
    ; is executed. Now change the 17 to 15 (lower than normal priority)
    ; and see that the print executes immediately
    ThreadPriority(thread, 17)
    PrintN("Waits for higher priority thread to finish")
  EndIf

  PrintN("Press return to exit")
  Input()
EndIf
```

Supported OS

Windows

130.11 TryLockMutex

Syntax

```
Result = TryLockMutex(Mutex)
```

Description

Tries to lock the specified mutex. Unlike LockMutex() , this function does not stop execution until the mutex is available. It returns immediately and the returnvalue indicates if the lock was successful or not. This is useful in situations were the thread should not wait for the mutex to be available but rather do other things in the meantime.

Returns nonzero if the mutex was successfully locked and zero if the lock failed.

Note: If the lock was successful, the UnlockMutex() function must be called to make the mutex available to other threads again. If this is not done, this could easily lead to a lockup situation.

Example:

```
Procedure ThreadProcedure(*Value)
  Shared Mutex

  Repeat
    If TryLockMutex(Mutex)
      PrintN("Mutex successfully locked.")

      UnlockMutex(Mutex)
      Break ; quit the loop and thread
    Else
      PrintN("Still waiting for mutex access...")
      Delay(200)
    EndIf
  ForEver
EndProcedure

OpenConsole()

Mutex = CreateMutex()
LockMutex(Mutex) ; main program has the mutex locked at first
Thread = CreateThread(@ThreadProcedure(), 0)

Delay(4000)
UnlockMutex(Mutex) ; now release the mutex, so the thread can get it

Input()
```

130.12 UnlockMutex

Syntax

```
UnlockMutex(Mutex)
```

Description

Unlocks a mutex previously locked by LockMutex() . The mutex is then available again for other threads to lock it.

Note: A mutex can only be unlocked by the thread that also locked it.

See CreateMutex() for a code example.

130.13 WaitThread

Syntax

```
Result = WaitThread(Thread [, Timeout])
```

Description

Stop the program execution until the specified 'Thread' exits, or the optional timeout (in milliseconds) is reached. If the thread is already finished, it returns immediately.

Thread - The number of the thread you want to wait for. This value is returned by CreateThread() .
Timeout - optional timeout to wait in miliseconds.

If the thread has ended, the returnvalue is nonzero. Zero is returned if the timeout was reached.

Example:

```
Procedure PrintStuff(*Interval)
  For i = 0 To 10
    PrintN(".")
    Delay(*Interval)
  Next
EndProcedure

If OpenConsole()
  thread = CreateThread(@PrintStuff(), 500)
  If thread
    ; Wait for thread to finish before we continue
    ; Try commenting the WaitThread function out and seeing what happens
    WaitThread(thread)

    For i = 0 To 10
      PrintN("A")
      Delay(1000)
    Next
  EndIf
EndIf
```

130.14 CreateSemaphore

Syntax

```
Semaphore = CreateSemaphore([InitialCount])
```

Description

Creates a new semaphore object. The optional 'InitialCount' must be a positive value that specifies the initial count of the semaphore. If it is not specified, the initial count is 0.

A semaphore is a thread synchronisation object that keeps an internal count. It has two kinds of operations: signal and wait . A wait operation decreases the count of the semaphore by one. If the count would drop below 0, the wait operation will block until a signal call is made. A signal operation increases the count one, releasing a blocking thread if there is one. A semaphore allows to enforce minimum or maximum counts across threads for example to prevent a queue from running out or getting too many items.

Unlike a mutex , a semaphore object is not "owned" by a particular thread, which means that signal/wait calls do not need to be done from the same thread as it is the case with LockMutex() and UnlockMutex() . In fact a common use for a semaphore objects is for one thread to do the SignalSemaphore() calls and for another one to do all WaitSemaphore() calls to implement a producer/consumer pattern.

Example:

This example shows a "producer" thread populating a queue with elements and the main thread reading them. The semaphore is used to make sure the queue never runs out of elements. Note that this could also be achieved with only a mutex and waiting/polling for queue elements with a Delay() inbetween, but the semaphore commands do a more efficient wait (returning immediately at the SignalSemaphore() call and not just on the next time a polling loop would check the queue).

```
Global Semaphore = CreateSemaphore()
Global Mutex     = CreateMutex()
Global NewList Queue()

Procedure Producer(Total)

  For i = 1 To Total
    Delay(Random(750) + 250)

    ; The queue access still needs a normal mutex lock to be threadsafe
    LockMutex(Mutex)
      LastElement(Queue())
      AddElement(Queue())
      Queue() = i
    UnlockMutex(Mutex)

    ; Signal that there is a new queue element
    SignalSemaphore(Semaphore)
  Next i

EndProcedure

If CreateThread(@Producer(), 30)

  For i = 1 To 30
    ; wait for one element to be available
    WaitSemaphore(Semaphore)

    ; display the queue state
    LockMutex(Mutex)
      Queue$ = "Queue:"
      ForEach Queue()
        Queue$ + " " + Str(Queue())
      Next Queue()
      Debug Queue$

      ; remove head element from the queue
      FirstElement(Queue())
      DeleteElement(Queue())
    UnlockMutex(Mutex)

  Next i

EndIf
```

130.15 FreeSemaphore

Syntax

```
FreeSemaphore(Semaphore)
```

Description

Destroys the given Semaphore object and frees all resources used by it.

130.16 SignalSemaphore

Syntax

```
SignalSemaphore(Semaphore)
```

Description

Increases the internal count of the semaphore by one, releasing a waiting thread if there is one.

Note: The semaphore count is limited to a signed 32bit value , so a maximum of 2147483647 SignalSemaphore() calls can be made without being balanced by WaitSemaphore() calls inbetween.

See CreateSemaphore() for a code example.

130.17 WaitSemaphore

Syntax

```
WaitSemaphore(Semaphore)
```

Description

Decreases the internal count of the semaphore by one, blocking thread execution if the count would fall below zero. A blocked thread is resumed as soon as another thread calls SignalSemaphore() .

Note: The semaphore count is limited to a signed 32bit value , so a maximum of 2147483647 WaitSemaphore() calls can be made without being balanced by SignalSemaphore() calls inbetween.

See CreateSemaphore() for a code example.

130.18 TrySemaphore

Syntax

```
Result = TrySemaphore(Semaphore)
```

Description

Decreases the internal count of the semaphore by one only if the count is above 0. This is the same as a WaitSemaphore() operation, but without blocking if the count would fall below 0.

Returns nonzero if the semaphore count was decreased, or 0 if the count could not be decreased because it was already zero.

Chapter 131

ToolBar

The toolbars are very useful to access some functions of the application quickly, with the help of small icons. It's often the shortcuts of menus items. PureBasic allows to create any number of toolbar and to handle them as if it was a menu.

131.1 CreateToolBar

Syntax

```
Result = CreateToolBar(#ToolBar, WindowID)
```

Description

Create an empty toolbar identified by the number '#ToolBar' on the given window. If #PB_Any is used as '#Toolbar' parameter, the new toolbar number will be returned as 'Result'. WindowID can be easily obtained with the WindowID() function.

This toolbar become the default toolbar for creation and it's possible to use ToolBarStandardButton() , ToolBarImageButton() and ToolBarSeparator() to add some elements to this toolbar.

If the 'Result' is 0, the toolbar can't be created, else all is fine.

The later checking of an event, means if the user clicked on an item of the toolbar, will done the same way like checking for a menu event with the function EventMenu() .

Example:

```
If OpenWindow(0, 0, 0, 150, 25, "ToolBar",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateToolBar(0, WindowID(0))
    ToolBarStandardButton(0, #PB_ToolBarIcon_New)
    ToolBarStandardButton(1, #PB_ToolBarIcon_Open)
    ToolBarStandardButton(2, #PB_ToolBarIcon_Save)
  EndIf
  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Menu
      Debug "ToolBar ID: "+Str(EventMenu())
    EndIf
  Until Event = #PB_Event_CloseWindow
EndIf
```

131.2 FreeToolBar

Syntax

```
FreeToolBar(#ToolBar)
```

Description

Free the specified #Toolbar.

Note: all remaining toolbars are automatically freed when the program ends.

131.3 DisableToolBarButton

Syntax

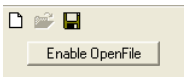
```
DisableToolBarButton(#ToolBar, ButtonID, State)
```

Description

Disable or enable the given #ToolBar button: if 'State' is 0, the button is enabled, if State is 1, the button is disabled.

Example:

```
If OpenWindow(0, 0, 0, 150, 60, "ToolBar",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateToolBar(0, WindowID(0))
    ToolBarStandardButton(0, #PB_ToolBarIcon_New)
    ToolBarStandardButton(1, #PB_ToolBarIcon_Open)
    ToolBarStandardButton(2, #PB_ToolBarIcon_Save)
    DisableToolBarButton(0, 1, 1) : Disabled = #True
  EndIf

  ButtonGadget(0, 20, 30, 110, 20, "Enable OpenFile")

  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Gadget
      If EventGadget() = 0
        If Disabled = #True
          DisableToolBarButton(0, 1, 0)
          SetGadgetText(0,"Disable OpenFile")
          Disabled = #False
        Else
          DisableToolBarButton(0, 1, 1)
          SetGadgetText(0,"Enable OpenFile")
          Disabled = #True
        EndIf
      EndIf
    EndIf
  Until Event = #PB_Event_CloseWindow
EndIf
```



131.4 GetToolBarButtonState

Syntax

```
State = GetToolBarButtonState(#ToolBar, ButtonID)
```

Description

Get the state of the specified #ToolBar button: if 'State' is 1, the button is toggled (pushed), if 'State' is 0, the button is not toggled.

Supported OS

Windows, Linux

131.5 IsToolBar

Syntax

```
Result = IsToolBar(#ToolBar)
```

Description

Tests if the given #ToolBar number is a valid and correctly initialized toolbar. This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure than a toolbar is ready to use.

131.6 SetToolBarButtonState

Syntax

```
SetToolBarButtonState(#ToolBar, ButtonID, State)
```

Description

Set the state of the specified #ToolBar button: if 'State' is 1, the button will be pushed, if 'State' is 0, the button will be released. It allows to use toggle like buttons on a toolbar.

Supported OS

Windows, Linux

131.7 ToolBarHeight

Syntax

```
Result = ToolBarHeight(#ToolBar)
```

Description

Returns the height in pixel of the #ToolBar. This is useful for correct calculation on window height when using a toolbar.

131.8 ToolBarImageButton

Syntax

```
ToolBarImageButton(#ButtonID, ImageID [, Mode])
```

Description

Add an image button to the toolbar being constructed. CreateToolBar() must be called before to use this function.

The 'ImageID' can be easily obtained by using ImageID() from the Image library. It can be an image loaded with LoadImage() or created in memory with CreateImage() . To have a real transparent background, use the 'icon' (.ico) file format on Windows, or the PNG file format on Linux/MacOS X.

With the optional 'Mode' parameter can be adjusted, if the ToolBarImageButton will act as a toggle button or not. One of the following constants can be used:

```
#PB_ToolBar_Normal: the button will act as standard button
 (default setting)
#PB_ToolBar_Toggle: the button will act as toggle button
```

Later the GetToolBarButtonState() and SetToolBarButtonState() functions can be used on it.

The later checking of an event, means if the user clicked on an item of the toolbar, will done the same way like checking for a menu event with the function EventMenu() .

Example:

```
If OpenWindow(0, 0, 0, 150, 25, "ToolBar",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  CreateImage(0,16,16)
  StartDrawing(ImageOutput(0))
    Box(0,0,16,16,RGB(255,255,255))
    Box(4,4,8,8,RGB(255,0,0))
  StopDrawing()
  CreateImage(1,16,16)
  StartDrawing(ImageOutput(1))
    Box(0,0,16,16,RGB(255,0,0))
    Box(4,4,8,8,RGB(255,255,255))
  StopDrawing()
  If CreateToolBar(0, WindowID(0))
    ToolBarImageButton(0,ImageID(0))
    ToolBarImageButton(1,ImageID(1))
  EndIf
  Repeat
  Until WaitWindowEvent() = #PB_Event_CloseWindow
EndIf
```

131.9 ToolBarSeparator

Syntax

```
ToolBarSeparator()
```

Description

Add a vertical separator to toolbar being constructed. CreateToolBar() must be called before to use this function.

Example:

```
If OpenWindow(0, 0, 0, 150, 25, "ToolBar",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateToolBar(0, WindowID(0))
    ToolBarStandardButton(0, #PB_ToolBarIcon_New)
    ToolBarSeparator()
    ToolBarStandardButton(1, #PB_ToolBarIcon_Open)
    ToolBarSeparator()
    ToolBarStandardButton(2, #PB_ToolBarIcon_Save)
    ToolBarSeparator()
    ToolBarSeparator()
  EndIf
  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Menu
      Debug "ToolBar ID: "+Str(EventMenu())
    EndIf
  Until Event = #PB_Event_CloseWindow
EndIf
```

131.10 ToolBarStandardButton

Syntax

```
ToolBarStandardButton(#ButtonID, #ButtonIcon [,Mode])
```

Description

Add a standard button to the toolbar being constructed. CreateToolBar() must be called before to use this function. A standard button is an icon which is available directly under the OS. The '#ButtonIcon' parameter must be one of the following constants :

```
#PB_ToolBarIcon_New
#PB_ToolBarIcon_Open
#PB_ToolBarIcon_Save
#PB_ToolBarIcon_Print
#PB_ToolBarIcon_PrintPreview
#PB_ToolBarIcon_Find
#PB_ToolBarIcon_Replace

#PB_ToolBarIcon_Cut
#PB_ToolBarIcon_Copy
#PB_ToolBarIcon_Paste
#PB_ToolBarIcon_Undo
#PB_ToolBarIcon_Redo

#PB_ToolBarIcon_Delete
#PB_ToolBarIcon_Properties
#PB_ToolBarIcon_Help
```

With the optional 'Mode' parameter can be adjusted, if the ToolBar button will act as a toggle button or not. One of the following constants can be used:

```
#PB_ToolBar_Normal: the button will act as standard button
 (default setting)
#PB_ToolBar_Toggle: the button will act as toggle button
```

Later the GetToolBarButtonState() and SetToolBarButtonState() functions can be used on it.

The later checking of an event, means if the user clicked on an item of the toolbar, will done the same way like checking for a menu event with the function EventMenu() .

Example:

```
If OpenWindow(0, 0, 0, 150, 25, "ToolBar",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateToolBar(0, WindowID(0))
    ToolBarStandardButton(0, #PB_ToolBarIcon_New)
    ToolBarStandardButton(1, #PB_ToolBarIcon_Open)
    ToolBarStandardButton(2, #PB_ToolBarIcon_Save)
  EndIf
  Repeat
    Event = WaitWindowEvent()
    If Event = #PB_Event_Menu
      Debug "ToolBar ID: "+Str(EventMenu())
    EndIf
  Until Event = #PB_Event_CloseWindow
EndIf
```

Supported OS

Windows, Linux

131.11 ToolBarToolTip

Syntax

```
ToolBarToolTip(#ToolBar, ButtonID, Text$)
```

Description

Associates the specified Text$ to the #ToolBar button. A Tooltip text is a text which displayed when the mouse cursor is over the button since a few time (yellow floating box).

Example:

```
If OpenWindow(0, 0, 0, 150, 60, "ToolBar",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  If CreateToolBar(0, WindowID(0))
    ToolBarStandardButton(0, #PB_ToolBarIcon_New)
    ToolBarStandardButton(1, #PB_ToolBarIcon_Open)
    ToolBarStandardButton(2, #PB_ToolBarIcon_Save)
    ToolBarToolTip(0, 0, "New document")
    ToolBarToolTip(0, 1, "Open file")
    ToolBarToolTip(0, 2, "Save file")
  EndIf
  Repeat
```

```
    Until WaitWindowEvent() = #PB_Event_CloseWindow
  EndIf
```

131.12 ToolBarID

Syntax

```
ToolBarID = ToolBarID(#ToolBar)
```

Description

Returns the unique system identifier of the #ToolBar.

This result is sometimes also known as 'Handle'. Look at the extra chapter Handles and Numbers for more information.

Chapter 132

Window

Windows are essential components of modern interfaces. PureBasic provides full access to them.

132.1 AddKeyboardShortcut

Syntax

```
AddKeyboardShortcut(#Window, Shortcut, Event)
```

Description

Add or replace a keyboard shortcut to the specified #Window. A shortcut generates a menu event (like a menu item) as most of them are used in conjunction with menus. 'Event' is the number which will be returned by the EventMenu() function. The 'Event' value has a limited range, from 0 to 64000. By default, a window already has the #PB_Shortcut_Tab and #PB_Shortcut_Tab||#PB_Shortcut_Shift shortcuts to handle tab and shift-tab correctly trough the gadgets. A shortcut can be removed with RemoveKeyboardShortcut() .

The 'Shortcut' parameter can be one of the following constants:

```
#PB_Shortcut_Back
#PB_Shortcut_Tab
#PB_Shortcut_Clear
#PB_Shortcut_Return
#PB_Shortcut_Menu
#PB_Shortcut_Pause
#PB_Shortcut_Print
#PB_Shortcut_Capital
#PB_Shortcut_Escape
#PB_Shortcut_Space
#PB_Shortcut_Prior
#PB_Shortcut_Next
#PB_Shortcut_End
#PB_Shortcut_Home
#PB_Shortcut_Left
#PB_Shortcut_Up
#PB_Shortcut_Right
#PB_Shortcut_Down
#PB_Shortcut_Select
#PB_Shortcut_Execute
#PB_Shortcut_Snapshot
```

```
#PB_Shortcut_Insert
#PB_Shortcut_Delete
#PB_Shortcut_Help
#PB_Shortcut_0
#PB_Shortcut_1
#PB_Shortcut_2
#PB_Shortcut_3
#PB_Shortcut_4
#PB_Shortcut_5
#PB_Shortcut_6
#PB_Shortcut_7
#PB_Shortcut_8
#PB_Shortcut_9
#PB_Shortcut_A
#PB_Shortcut_B
#PB_Shortcut_C
#PB_Shortcut_D
#PB_Shortcut_E
#PB_Shortcut_F
#PB_Shortcut_G
#PB_Shortcut_H
#PB_Shortcut_I
#PB_Shortcut_J
#PB_Shortcut_K
#PB_Shortcut_L
#PB_Shortcut_M
#PB_Shortcut_N
#PB_Shortcut_O
#PB_Shortcut_P
#PB_Shortcut_Q
#PB_Shortcut_R
#PB_Shortcut_S
#PB_Shortcut_T
#PB_Shortcut_U
#PB_Shortcut_V
#PB_Shortcut_W
#PB_Shortcut_X
#PB_Shortcut_Y
#PB_Shortcut_Z
#PB_Shortcut_LeftWindows
#PB_Shortcut_RightWindows
#PB_Shortcut_Apps
#PB_Shortcut_Pad0
#PB_Shortcut_Pad1
#PB_Shortcut_Pad2
#PB_Shortcut_Pad3
#PB_Shortcut_Pad4
#PB_Shortcut_Pad5
#PB_Shortcut_Pad6
#PB_Shortcut_Pad7
#PB_Shortcut_Pad8
#PB_Shortcut_Pad9
#PB_Shortcut_Multiply
#PB_Shortcut_Add
#PB_Shortcut_Separator
#PB_Shortcut_Subtract
#PB_Shortcut_Decimal
#PB_Shortcut_Divide
#PB_Shortcut_F1
```

```
#PB_Shortcut_F2
#PB_Shortcut_F3
#PB_Shortcut_F4
#PB_Shortcut_F5
#PB_Shortcut_F6
#PB_Shortcut_F7
#PB_Shortcut_F8
#PB_Shortcut_F9
#PB_Shortcut_F10
#PB_Shortcut_F11
#PB_Shortcut_F12
#PB_Shortcut_F13
#PB_Shortcut_F14
#PB_Shortcut_F15
#PB_Shortcut_F16
#PB_Shortcut_F17
#PB_Shortcut_F18
#PB_Shortcut_F19
#PB_Shortcut_F20
#PB_Shortcut_F21
#PB_Shortcut_F22
#PB_Shortcut_F23
#PB_Shortcut_F24
#PB_Shortcut_Numlock
#PB_Shortcut_Scroll
```

The above key can be combined with any of the following constants:

```
#PB_Shortcut_Shift
#PB_Shortcut_Control
#PB_Shortcut_Alt
#PB_Shortcut_Command
```

Note: the #PB_Shortcut_Command constant is only useful on Mac OSX and allow to use the 'Apple' key (left or right) to define shortcuts. This constant is also supported on others OS (to ease portability), but will act like #PB_Shortcut_Control. The shortcuts Apple+Q and Apple+P, are predefined on Mac OSX for the #PB_Menu_Quit and #PB_Menu_Preferences menu entries in the application menu and cannot be reassigned.

Example:

```
AddKeyboardShortcut(0, #PB_Shortcut_Control | #PB_ShortCut_F, 15) ; Will create a keyboard shortcut CTRL+F on the window 0 which will fires a menu event '15'
```

132.2 AddWindowTimer

Syntax

```
AddWindowTimer(#Window, Timer, Timeout)
```

Description

Adds a new timer to the specified #Window. This will cause #PB_Event_Timer events to be received periodically in the WindowEvent() or WaitWindowEvent() functions. The RemoveWindowTimer() function can be used to remove the timer again.

The #Window parameter specifies the window for the timer. A timer is always attached to a window and will be removed if that window is closed.

The Timer parameter is a userdefined number that identifies this timer. Timers on separate windows may have overlapping numbers. This value will later be returned from EventTimer() when a #PB_Event_Timer is received. It can also be used to remove the timer again with the RemoveWindowTimer() function.

The Timeout parameter specifies the amount of time in milliseconds between the #PB_Event_Timer events.

Note: The timer events will only be generated when there are no other events to be processed (timers are low-priority events). This means that the time that ellapses between two timer events may be larger than the specified Timeout value. Timers are therefore not suited for precise timing but are rather intended to perform periodic tasks such as updating a gadget content or similar.

Example:

```
If OpenWindow(0, 0, 0, 400, 100, "Timer Example",
 #PB_Window_SystemMenu|#PB_Window_ScreenCentered)
  ProgressBarGadget(0, 10, 10, 380, 20, 0, 100)
  AddWindowTimer(0, 123, 250)

  Value = 0
  Repeat
    Event = WaitWindowEvent()

    If Event = #PB_Event_Timer And EventTimer() = 123
      Value = (Value + 5) % 100
      SetGadgetState(0, Value)
    EndIf

  Until Event = #PB_Event_CloseWindow
EndIf
```

132.3 RemoveWindowTimer

Syntax

```
RemoveWindowTimer(#Window, Timer)
```

Description

Removes the specified Timer from #Window. Timer has to be the same value that was used in AddWindowTimer() to create the timer. There will be no further events received for this timer.

132.4 EventTimer

Syntax

```
Timer = EventTimer()
```

Description

After an event of type #PB_Event_Timer (returned by WindowEvent() or WaitWindowEvent()), use this function to determine which timer caused the event. The returned value is the same that was used in AddWindowTimer() to create the timer.

132.5 CloseWindow

Syntax

```
CloseWindow(#Window)
```

Description

Close the specified window.

Note: all remaining opened windows are automatically closed when the program ends.

132.6 DisableWindow

Syntax

```
DisableWindow(#Window, State)
```

Description

Enables or disables user input to the specified Window.

'State' can take the following values:

```
1: The window is disabled.
0: The window is enabled.
```

132.7 EventGadget

Syntax

```
GadgetNumber = EventGadget()
```

Description

After an event of type #PB_Event_Gadget (returned by WindowEvent() or WaitWindowEvent()), use this function to determine which gadget has been triggered. It returns the #Gadget number.

132.8 EventMenu

Syntax

```
MenuItem = EventMenu()
```

Description

After an event of the type #PB_Event_Menu (returned by WindowEvent() or WaitWindowEvent()), use this function to determine which menu item has been selected (return the #MenuItem number).

This function also checks the selected item of a toolbar created with CreateToolBar() .

A toolBar event is like a menu event (as tools are shortcuts for menu items most of the time). So it's a good idea, if the toolbar buttons and the menu items have the same ID, then the same operation can be done on both without any additional code.

132.9 EventType

Syntax

```
EventType = EventType()
```

Description

After a WindowEvent() or WaitWindowEvent() function, use this function to determine of which type the event is. The following values are possible, if an event of the type #PB_Event_Gadget (library Gadget) or #PB_Event_SysTray (library Systray) occurs:

```
  #PB_EventType_LeftClick        : Left mouse button click
  #PB_EventType_RightClick       : right mouse button click
  #PB_EventType_LeftDoubleClick  : Left mouse button double click
  #PB_EventType_RightDoubleClick: Right mouse button double
   click
  #PB_EventType_Focus            : Get the focus.
  #PB_EventType_LostFocus        : Lose the focus.
  #PB_EventType_Change           : Content change.
  #PB_EventType_DragStart        : The user tries to start a
   Drag & Drop
 operation.
```

The following gadgets support EventType():

- ExplorerListGadget()
- ExplorerTreeGadget()
- ImageGadget()
- ListViewGadget()
- ListIconGadget()
- MDIGadget()
- SpinGadget()
- StringGadget()
- WebGadget() - The WebGadget has a special set of event types.

(See the gadget definition to see which events are supported.)

Example:

```
1   If OpenWindow(0, 0, 0, 230, 120, "Eventtypes example...",
     #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
2     ListIconGadget(1, 10, 10, 150, 100, "ListIcon", 140,
     #PB_ListIcon_GridLines)
3     For a = 1 To 4
4       AddGadgetItem(1, -1, "Line "+Str(a))
5     Next
6
7     Repeat
8       Event = WaitWindowEvent()
9
10      Select Event
11
12        Case #PB_Event_Gadget
13          Select EventGadget()
14            Case 1
```

```
                Select EventType()
                  Case #PB_EventType_LeftClick        : Debug "Click with left mouse button"
                  Case #PB_EventType_RightClick       : Debug "Click with right mouse button"
                  Case #PB_EventType_LeftDoubleClick  : Debug "Double-click with left mouse button"
                  Case #PB_EventType_RightDoubleClick : Debug "Double-click with right mouse button"
                EndSelect
            EndSelect

        EndSelect
      Until Event = #PB_Event_CloseWindow
  EndIf
```

132.10 EventWindow

Syntax

```
WindowNumber = EventWindow()
```

Description

After a WindowEvent() or WaitWindowEvent() function, use this function to determine on which window the event has occurred.

132.11 GetActiveWindow

Syntax

```
WindowNumber = GetActiveWindow()
```

Description

Returns the PB Number of the window which currently has the keyboard focus.

A window can be activated (set the focus on it) with the SetActiveWindow() function.

132.12 GetWindowColor

Syntax

```
Color = GetWindowColor(#Window)
```

Description

Returns the background color of the specified window that was set with SetWindowColor() .

If no background color was set yet, -1 is returned.

132.13 GetWindowState

Syntax

```
State = GetWindowState(#Window)
```

Description

Checks wether the specified window is maximized, minimized or displayed normally.

It returns one of the following values:

```
#PB_Window_Normal   : The window is neither maximized nor
 minimized.
#PB_Window_Maximize: The window is maximized.
#PB_Window_Minimize: The window is minimized.
```

The status of a window can be changed with the SetWindowState() function.

132.14 GetWindowTitle

Syntax

```
Result\$ = GetWindowTitle(#Window)
```

Description

Returns in 'Result$' the text which is currently displayed in the specified #Window's title bar.

```
1  If OpenWindow(2, 100, 100, 200, 100, "My cool title")
2    Title$ = GetWindowTitle(2)  ; Will return "My cool title"
3  EndIf
```

The title of a window can be changed with SetWindowTitle() .

132.15 HideWindow

Syntax

```
HideWindow(#Window, State)
```

Description

Hides or Shows the specified #Window. 'State' can take the following values:

```
1: the #Window is hidden
0: the #Window is shown
```

132.16 IsWindow

Syntax

```
Result = IsWindow(#Window)
```

Description

Tests if the given #Window is a valid and correctly initialized window.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that a window is ready to use.

132.17 OpenWindow

Syntax

```
Result = OpenWindow(#Window, x, y, InnerWidth, InnerHeight,
    Title$ [, Flags [, ParentWindowID]])
```

Description

Opens a new window according to the specified parameters. If #PB_Any is used as '#Window' parameter, the new window number will be returned as 'Result'.

The new window becomes the active window, it's not needed to use SetActiveWindow() (unless the window is created as invisible). If the 'Result' is 0, the window can't be opened.

'InnerWidth' and 'InnerHeight' specify the required client area (without borders and window decorations), for Windows XP, AmigaOS and Linux skinning compatibility. 'x' and 'y' specify the initial position of the window. (unless one of the center flags is used.) If one of them is #PB_Ignore, the OS will choose a position for the window.

'ParentWindowID' is an optional parameter which specify on which window the new window belongs to. 'ParentWindowID' value can be easily obtained with WindowID() .

Possible flags are:

```
  #PB_Window_SystemMenu     : Enables the system menu on the
   window title bar.
  #PB_Window_MinimizeGadget: Adds the minimize gadget to the
   window title bar. #PB_Window_SystemMenu is automatically
   added.
  #PB_Window_MaximizeGadget: Adds the maximize gadget to the
   window title bar. #PB_Window_SystemMenu is automatically
   added.
  #PB_Window_SizeGadget     : Adds the sizeable feature to a
   window.
  #PB_Window_Invisible      : Creates the window but don't
   display. Not supported on AmigaOS.
  #PB_Window_TitleBar       : Creates a window with a titlebar.
  #PB_Window_Tool           : Creates a window with a smaller
   titlebar and no taskbar entry.
  #PB_Window_BorderLess     : Creates a window without any
   borders.
  #PB_Window_ScreenCentered: Centers the window in the middle
   of the screen. x,y parameters are ignored.
  #PB_Window_WindowCentered: Centers the window in the middle
   of the parent window ('ParentWindowID' must be specified).
   x,y parameters are ignored.
  #PB_Window_Maximize       : Opens the window maximized. (Note:
   on Linux, not all Windowmanagers support this)
  #PB_Window_Minimize       : Opens the window minimized.
  #PB_Window_NoGadgets      : Prevents the creation of a
   GadgetList. UseGadgetList()
 can be used to do this later.
```

If the optional Flags parameter is not specified, #PB_Window_SystemMenu will be used.

All possible events in a window are handled with the WindowEvent() and WaitWindowEvent() functions. For special situations callbacks are used, see description at SetWindowCallback() . On Windows, the height and width of a window with a titlebar can't be less than around 100 pixels. To open smaller windows, use the "#PB_Window_BorderLess" flag.

Note: When opening a Window from a thread , the thread must also call WindowEvent() or WaitWindowEvent() in a loop to process events for this window, as window events are not sent between different threads.

132.18 RemoveKeyboardShortcut

Syntax

```
RemoveKeyboardShortcut(#Window, Shortcut)
```

Description

Remove a keyboard shortcut previously defined with AddKeyboardShortcut() . For a full list of the available shortcut, just look to the AddKeyboardShortcut() function. If 'Shortcut' is #PB_Shortcut_All the shortcuts are all removed.

132.19 ResizeWindow

Syntax

```
ResizeWindow(#Window, x, y, Width, Height)
```

Description

Move and resize the given window to the given position and size. If any of the parameters should be ignored (not be changed) #PB_Ignore can be passed at this place.

132.20 SetActiveWindow

Syntax

```
SetActiveWindow(#Window)
```

Description

Activate the specified window, which means the focus has been put on this window. The function will only change the focus within the program. It can not bring the program to the foreground when another program has the focus.

132.21 SetWindowCallback

Syntax

```
SetWindowCallback(@ProcedureName() [, #Window])
```

Description

For experienced programmers only. It's only supported on Microsoft Windows OS.

Normal events should be handled with the regular WaitWindowEvent() or WindowEvent() .

This function associates a callback to handle the events of the all open windows. All the events are catched by this callback and can be processed here. To set a callback for a specific window only, the optional parameter can be used to pass the PB window number.

To remove/disable a previous set Callback just call SetWindowCallback(0 [, #Window]).

Warning: this way is lowlevel. Incorrect handling of the messages in the callback can interfere with PB's own message processing.

The callback procedure must have 4 parameters. Here is a sample code to use a callback correctly:

```
Procedure MyWindowCallback(WindowID, Message, wParam, lParam)
  Result = #PB_ProcessPureBasicEvents
  ;
  ; you code here
  ;
  ProcedureReturn Result
EndProcedure
```

Here you see a working example checking some window parameters (using Windows API constants):

```
Procedure WinCallback(hWnd, uMsg, wParam, lParam)
  ; Windows fills the parameter automatically, which we will use in the callback...

  If uMsg = #WM_SIZE
    Select wParam
      Case #SIZE_MINIMIZED
        Debug "Window was minimized"
      Case #SIZE_RESTORED
        Debug "Window was restored"
      Case #SIZE_MAXIMIZED
        Debug "Window was maximized"
    EndSelect
  EndIf

  ProcedureReturn #PB_ProcessPureBasicEvents
EndProcedure


If OpenWindow(0, 0, 0, 200, 100, "Messages", #PB_Window_MinimizeGadget|#PB_Window_MaximizeGadget)

  SetWindowCallback(@WinCallback())    ; activate the callback

  Repeat
    Select WaitWindowEvent()
      Case #PB_Event_CloseWindow
        End
    EndSelect
  ForEver

EndIf
```

Supported OS

Windows

132.22 SetWindowColor

Syntax

```
SetWindowColor(#Window, Color)
```

Description

Changes the background color of the specified window.

RGB() can be used to get a valid color value. A color table with common colors is available here .

Use -1 as 'Color' to reset the background to the default.

The actual background color of the window can later be get with GetWindowColor() .

132.23 SetWindowState

Syntax

```
SetWindowState(#Window, State)
```

Description

Changes the minimized/maximized of the specified window.

'State' can have the following values:

```
#PB_Window_Normal   : The window will be neither maximized nor
 minimized.
#PB_Window_Maximize: The window will be maximized.
#PB_Window_Minimize: The window will be minimized.
```

The actual status of the window can later be get with GetWindowState() .

132.24 SetWindowTitle

Syntax

```
SetWindowTitle(#Window, Title$)
```

Description

Changes the text which is currently displayed in the specified #Window's title bar.

Example:

```
1  If OpenWindow(2, 100, 100, 200, 100, "My cool title")
2    SetWindowTitle(2, "Even cooler title !")
3
4    Repeat
5    Until WaitWindowEvent() = #PB_Event_CloseWindow
6  EndIf
```

The actual window title can later be get with GetWindowTitle() .

132.25 SmartWindowRefresh

Syntax

```
SmartWindowRefresh(#Window, State)
```

Description

Enable a smart way to refresh the window to reduce the flickering when resizing the window. If the window isn't resizable, the function isn't needed. This function just try to help with the flickering problems, but it won't always give good results. The only way to see if it will work for a specific window is to enable it, see if there is any difference and see if the window content isn't trashed when a resize is done. If 'State' is 0, the smart refresh is disabled, else it is enabled.

Supported OS

Windows

132.26 StickyWindow

Syntax

```
StickyWindow(#Window, State)
```

Description

Makes the specified window stay on top of all other open windows (also from other programs), even if it does not have the focus.

'State' can have the following values:

```
1: the window will stay on top of all others.
0: the window will not stay on top of all others when it does
 not have the focus.
```

132.27 WindowEvent

Syntax

```
Event = WindowEvent()
```

Description

Check if an event has occured on any of the opened windows.

WindowEvent() returns the next event from the event queue and returns zero when there are no more events. Unlike WaitWindowEvent() it doesn't wait for the next event - it always returns immediately.

This makes it useful for window event loops, where other processing needs to be done without waiting for an event to happen (e.g. Network transactions) and therefore WaitWindowEvent() can't be used.

It must be handled with care though if used on a continuing basis, because unlike WaitWindowEvent() , it will not give CPU time to other programs while waiting for an event and therefore consume all CPU power. In this case, either Delay() should be used somewhere in the loop or WaitWindowEvent() with a small timeout value.

To get the window number in which the event occurred, use the EventWindow() function.

Possible Events are :

```
  #PB_Event_Menu            : a menu
 has been selected
  #PB_Event_Gadget          : a gadget
 has been pushed
  #PB_Event_SysTray         : an icon in the systray
 zone was clicked
  #PB_Event_Timer           : a timer
 has reached its timeout
  #PB_Event_CloseWindow     : the window close gadget has been
   pushed
  #PB_Event_Repaint         : the window content has been
   destroyed and must be repainted (useful for 2D graphics
   operations
)
  #PB_Event_SizeWindow      : the window has been resized
  #PB_Event_MoveWindow      : the window has been moved
  #PB_Event_MinimizeWindow : the window has been minimized
  #PB_Event_MaximizeWindow : the window has been maximized
  #PB_Event_RestoreWindow  : the window has been restored to
   normal size (either from a minimum or maximum size)
  #PB_Event_ActivateWindow : the window has been activated (got
   the focus)
  #PB_Event_WindowDrop      : a Drag & Drop
 operation was finished on a window
  #PB_Event_GadgetDrop      : a Drag & Drop
 operation was finished on a gadget
```

A basic example for event handling can be found in the WaitWindowEvent() description.

After a #PB_Event_WindowDrop or #PB_Event_GadgetDrop Event, the event functions of the Drag & Drop library can be used to examine and read the dropped data.

Note: When opening a Window from a thread, the thread must also call WindowEvent() or WaitWindowEvent() in a loop to process events for this window, as window events are not sent between different threads.

The correct way to handle an WindowEvent() loop is something like that:

```
1   Repeat
2     Event = WindowEvent()
3
4     If Event     ; An event was in the queue so process it
5       ....
6     Else
7       Delay(1)   ; No event, let the others apps get some CPU
    time too !
8     EndIf
9   Until Event = #PB_Event_CloseWindow
```

Note: The 'Delay' shouldn't be put after each event, because when lot of events will come (like refresh, gadgets updates etc..) the app will wait 20 ms (on windows at least) between each event. So the delay need to be put when no events are received. Another way is to use WaitWindowEvent() with a small timeout value.

132.28 WaitWindowEvent

Syntax

```
Event = WaitWindowEvent([Timeout])
```

Description

Wait until an event occurs. It's the same function as WindowEvent() but locks the program execution, which is very important in a multitasking environment.

An optional timeout (in milliseconds) can be specified, which causes the function to return after the specified amount of time if no events are occuring.

An application should always use this function instead of WindowEvent() if possible. For more information, see the documentation of WindowEvent() .

Notes:
When opening a Window from a thread, the thread must also call WindowEvent() or WaitWindowEvent() in a loop to process events for this window, as window events are not sent between different threads.

WaitWindowEvent() can only be called once per event loop, because else events will be "lost" (every event can only be processed once and isn't available anymore for a second time after first processing).

Basic example:

```
  If OpenWindow(0, 0, 0, 230, 90, "Event handling example...",
   #PB_Window_SystemMenu | #PB_Window_ScreenCentered)

   ButtonGadget  (1, 10, 10, 200, 20, "Click me")
   CheckBoxGadget(2, 10, 40, 200, 20, "Check me")

   If CreateMenu(0, WindowID(0))
     MenuTitle("Menu")
     MenuItem(1, "Item 1")
     MenuItem(2, "Item 2")
     MenuItem(3, "Item 3")
   EndIf

   Repeat
     Event = WaitWindowEvent()

     Select Event

       Case #PB_Event_Gadget
         Select EventGadget()
           Case 1 : Debug "Button 1 clicked!"
           Case 2 : Debug "Button 2 clicked!"
         EndSelect

       Case #PB_Event_Menu
         Select EventMenu()
           Case 1 : Debug "Menu item 1 clicked!"
           Case 2 : Debug "Menu item 2 clicked!"
           Case 3 : Debug "Menu item 3 clicked!"
         EndSelect

     EndSelect
   Until Event = #PB_Event_CloseWindow
 EndIf
```

132.29 WindowBounds

Syntax

```
WindowBounds(#Window, MinimalWidth, MinimalHeight,
    MaximalWidth, MaximalHeight)
```

Description

Changes the minimal and maximal #Window dimensions (in pixels). This is useful to prevent a window from becomming too small or too big when the user resizes it. It's possible to use #PB_Ignore instead of specifying a width/height value to keep the current value.

Example:

```
1   If OpenWindow(0, 0, 0, 300, 300, "Resize me !",
     #PB_Window_SystemMenu | #PB_Window_ScreenCentered |
     #PB_Window_SizeGadget)
2    WindowBounds(0, 200, 200, 400, 400)
3
4    Repeat
5      Event = WaitWindowEvent()
6    Until Event = #PB_Event_CloseWindow
7   EndIf
```

132.30 WindowHeight

Syntax

```
Height = WindowHeight(#Window)
```

Description

Returns the height, in pixels, of the given window.

To get the height of the desktop, use DesktopHeight() from the Desktop library.

132.31 WindowID

Syntax

```
WindowID = WindowID(#Window)
```

Description

Returns the unique ID which identifies the given '#Window' in the operating system. This function is very useful when another library needs a window reference.

This result is sometimes also known as 'Handle'. Look at the extra chapter Handles and Numbers for more information.

132.32 WindowWidth

Syntax

```
Width = WindowWidth(#Window)
```

Description

Return the width, in pixels, of the given window.

To get the width of the desktop, use DesktopWidth() from the Desktop library.

132.33 WindowX

Syntax

```
x = WindowX(#Window)
```

Description

Returns the left position on the screen, in pixels, of the given window.

132.34 WindowY

Syntax

```
y = WindowY(#Window)
```

Description

Returns the top position on the screen, in pixels, of the given window.

132.35 WindowMouseX

Syntax

```
x = WindowMouseX(#Window)
```

Description

Returns the mouse x position in the inner area of the given window. If the mouse is outside of the window area, it will return -1.

To get the absolute x position of the mouse on the desktop, use the DesktopMouseX() function.

```
If OpenWindow(0, 0, 0, 300, 30, "Window mouse monitor",
 #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  TextGadget(0, 10, 6, 200, 20, "")

  Repeat
    Event = WaitWindowEvent(20) ; return at least every 20ms
 for an update

    SetGadgetText(0, "Window mouse position: " +
 Str(WindowMouseX(0)) + "," + Str(WindowMouseY(0)))
  Until Event = #PB_Event_CloseWindow
EndIf
```

132.36 WindowMouseY

Syntax

```
y = WindowMouseY(#Window)
```

Description

Returns the mouse y position in the inner area of the given window. If the mouse is outside of the window area, it will return -1.

To get the absolute y position of the mouse on the desktop, use the DesktopMouseY() function.

```
If OpenWindow(0, 0, 0, 300, 30, "Window mouse monitor", #PB_Window_SystemMenu | #PB_Window_ScreenCentered)
  TextGadget(0, 10, 6, 200, 20, "")

  Repeat
    Event = WaitWindowEvent(20) ; return at least every 20ms for an update

    SetGadgetText(0, "Window mouse position: " + Str(WindowMouseX(0)) + "," + Str(WindowMouseY(0)))
  Until Event = #PB_Event_CloseWindow
EndIf
```

132.37 WindowOutput

Syntax

```
OutputID = WindowOutput(#Window)
```

Description

Returns the OutputID of the given window to perform 2D rendering operation on it. It will use the PureBasic 2DDrawing library (see StartDrawing()).

Note: Content drawn on a window will be erased whenever the window or a part of it is covered by another window, moved outside of the screen or when the window is hidden or minimized. So to keep the drawn content visible, it must be redrawn after every #PB_Event_Repaint event. A more convenient alternative is to draw the content to an image via ImageOutput() and display it as ImageGadget() in the application window and if necessary, update it with SetGadgetState() . This way all needed refreshing will be handled by the ImageGadget.

Chapter 133

Window3D

The Window3D library allows to create complex Graphical User Interface (GUI) directly over the screen area, using the 3D engine. This is mainly intended for game or application which needs user inputs while running in fullscreen mode. This library is is based on the regular PureBasic window library , and offer similar syntax and behaviour. The GUI engine used is CEGUI, which offer some nice options like skinning, good speed and a lot of build-in gadgets. More information about CEGUI can be found here: http://www.cegui.org.uk.

It uses the 3D engine, so InitEngine3D() has to be called succesfully before using these functions.

133.1 CloseWindow3D

Syntax

```
CloseWindow3D(#Window3D)
```

Description

Close the specified window.

Note: all remaining opened windows are automatically closed when the program ends.

133.2 DisableWindow3D

Syntax

```
DisableWindow3D(#Window3D, State)
```

Description

Enables or disables user input to the specified Window.

'State' can take the following values:

```
1: The window is disabled.
0: The window is enabled.
```

133.3 EventGadget3D

Syntax

```
GadgetNumber = EventGadget3D()
```

Description

After an event of type #PB_Event3D_Gadget (returned by WindowEvent3D()), use this function to determine which gadget has been triggered. It returns the #Gadget3D number.

133.4 EventType3D

Syntax

```
EventType = EventType3D()
```

Description

After a WindowEvent3D() function, use this function to determine of which type the event is. The following values are possible, if an event of the type #PB_Event3D_Gadget (library Gadget3D) occurs:

```
#PB_EventType3D_Focus     : Get the focus.
#PB_EventType3D_LostFocus: Lose the focus.
#PB_EventType3D_Change    : Content change.
```

The following gadgets support EventType3D():

- SpinGadget3D()
- StringGadget3D()

(See the gadget definition to see which events are supported.)

133.5 EventWindow3D

Syntax

```
WindowNumber = EventWindow3D()
```

Description

After a WindowEvent3D() function, use this function to determine on which window the event has occurred.

133.6 GetActiveWindow3D

Syntax

```
WindowNumber = GetActiveWindow3D()
```

Description

Returns the PB Number of the window which currently has the keyboard focus.

A window can be activated (set the focus on it) with the SetActiveWindow3D() function.

133.7 GetWindowTitle3D

Syntax

```
Result\$ = GetWindowTitle3D(#Window3D)
```

Description

Returns the text which is currently displayed in the specified #Window3D's title bar.

The title of a window can be changed with SetWindowTitle3D() .

133.8 HideWindow3D

Syntax

```
HideWindow3D(#Window3D, State)
```

Description

Hides or shows the specified #Window3D. 'State' can take the following values:

```
1: the #Window3D is hidden
0: the #Window3D is shown
```

133.9 IsWindow3D

Syntax

```
Result = IsWindow3D(#Window3D)
```

Description

Tests if the given #Window3D is a valid and correctly initialized window.

This function is bullet-proof and can be used with any value. If the 'Result' is not zero then the object is valid and initialized, else it returns zero. This is the correct way to ensure that a window is ready to use.

133.10 OpenWindow3D

Syntax

```
Result = OpenWindow3D(#Window3D, x, y, InnerWidth, InnerHeight,
   Title$ [, Flags])
```

Description

Opens a new window on the current screen according to the specified parameters. If #PB_Any is used as '#Window3D' parameter, the new window number will be returned as 'Result'.

The new window becomes the active window, it's not needed to use SetActiveWindow3D() (unless the window is created as invisible). If the 'Result' is 0, the window can't be opened.

'InnerWidth' and 'InnerHeight' specify the required client area (without borders and window decorations). 'x' and 'y' specify the initial position of the window. (unless one of the center flags is used.) If one of them is #PB_Ignore, the OS will choose a position for the window.

Possible flags are:

```
#PB_Window3D_SizeGadget : Adds the sizeable feature to a
 window.
#PB_Window3D_Invisible  : Creates the window but don't
 display it.
#PB_Window3D_BorderLess : Creates a window without any
 borders.
```

All possible events in a window are handled with WindowEvent3D() .

133.11 ResizeWindow3D

Syntax

```
ResizeWindow3D(#Window3D, x, y, Width, Height)
```

Description

Move and resize the given window to the given position and size. If any of the parameters should be ignored (not be changed) #PB_Ignore can be passed at this place.

133.12 SetActiveWindow3D

Syntax

```
SetActiveWindow3D(#Window3D)
```

Description

Activate the specified window, which means the focus has been put on this window. The function will only change the focus within the program. It can not bring the program to the foreground when another program has the focus.

133.13 SetWindowTitle3D

Syntax

```
SetWindowTitle3D(#Window3D, Title$)
```

Description

Changes the text which is currently displayed in the specified #Window3D's title bar.

The actual window title can later be get with GetWindowTitle3D() .

133.14 WindowEvent3D

Syntax

```
Event = WindowEvent3D()
```

Description

Check if an event has occured on any of the opened windows.

WindowEvent3D() returns the next event from the event queue and returns zero when there are no more events. It doesn't wait for the next event - it always returns immediately.

To get the window number in which the event occurred, use the EventWindow3D() function.
Possible Events are :

```
  #PB_Event3D_Gadget          : a Gadget3D
 has been pushed
  #PB_Event3D_CloseWindow     : the window close gadget has been
   pushed
  #PB_Event3D_SizeWindow      : the window has been resized
  #PB_Event3D_MoveWindow      : the window has been moved
  #PB_Event3D_ActivateWindow  : the window has been activated
   (got the focus)
```

133.15 WindowHeight3D

Syntax

```
Height = WindowHeight3D(#Window3D)
```

Description

Returns the height, in pixels, of the given window.

133.16 WindowID3D

Syntax

```
WindowID = WindowID3D(#Window3D)
```

Description

Returns the unique ID which identifies the given '#Window3D' in the operating system.
This function is very useful when another library needs a window reference.

133.17 WindowWidth3D

Syntax

```
Width = WindowWidth3D(#Window3D)
```

Description

Return the width, in pixels, of the given window.

133.18 WindowX3D

Syntax

```
x = WindowX3D(#Window3D)
```

Description

Returns the left position on the screen, in pixels, of the given window.

133.19 WindowY3D

Syntax

```
y = WindowY3D(#Window3D)
```

Description

Returns the top position on the screen, in pixels, of the given window.

Chapter 134

XML

The XML library provides set of functions to easily add XML parsing and creating capability to applications. It is based on the expat XML parser, which is licensed under the MIT license which can be viewed here . expat is used in many projects (like Mozilla or perl). It is very stable and very fast.

Important: The expat license requires that a copyright notice and the license text itself be included in any software that includes the parser. So if this library (or the API import) are used in software that is to be made public, the above linked license must be included with the software.

The expat functions can also be called directly like other api functions with an ending underscore. This allows to take advantage of expat features not directly provided by this library. The constants and structures defined in expat.h are directly available in PureBasic. A unicode compilation of expat is automatically linked if the Compiler-Switch "Create unicode executable" switch is used. For detailed information on the expat functions, refere to the documentation provided in the download package on http://expat.sourceforge.net/

This library has partial support for Document Type Definitions (DTD) and Namespaces. The goal is to keep the commandset very simple while still allowing this library to handle any XML compliant document.

The expat parser is a non-validating parser. This means it checks the parsed documents for errors in the markup (a document must be well-formed according to the XML specification), but it does not validate the document against a DTD. When parsing a document, this library places DTDs inside a special node in the XML tree with the type #PB_XML_DTD. The content of this node is the full DOCTYPE tag. This way it can be easily accessed and manipulated but is also save to ignore if this information is not needed. The tag is simply written back when exporting/saving the document.

Namespaces do not get resolved when parsing a document. This means that in a document using namespaces, the namespace declarations are accessible as normal node attributes, and node/attribute names using namespaces will be visible as "namespace:tagname". This allows a document using namespaces to be read and also saved back like any other document without destroying its structure. To make working with namespaces simpler, the functions ResolveXMLNodeName() and ResolveXMLAttributeName() are provided to resolve names inside documents that use namespaces.

The PureBasic Debugger provides the possibility to examine #XML objects during runtime with the Library Viewer tool.

The official specification of XML and XML Namespaces by the W3C can be found here:
XML specification
XML Namespaces
Various translations of XML related documents

Also the Wikipedia article on XML provides a good starting point for people new to XML.

134.1 IsXML

Syntax

```
Result = IsXML(#XML)
```

Description

Returns nonzero if #XML referes to an existing XML tree.

134.2 FreeXML

Syntax

```
FreeXML(#XML)
```

Description

Frees the XML tree and all data it contains.

Note: all remaining xml objects are automatically freed when the program ends.

134.3 CreateXML

Syntax

```
Result = CreateXML(#XML [, Encoding])
```

Description

Creates a new empty XML tree identified by the #XML number. If #PB_Any is used as '#XML' parameter, the new XML tree number will be returned as 'Result'. 'Encoding' specifies the encoding to use for the tree. Valid values are #PB_Ascii, #PB_Unicode or #PB_UTF8. (#PB_UTF8 is the default)

The new tree will only have a root node which can be accessed with RootXMLNode() . To add new nodes, CreateXMLNode() can be used.

Example:

```
; Create xml tree
xml = CreateXML(#PB_Any)
mainNode = CreateXMLNode(RootXMLNode(xml))
SetXMLNodeName(mainNode, "Zoo")

; Create first xml node (in main node)
item = CreateXMLNode(mainNode)
SetXMLNodeName(item, "Animal")
SetXMLAttribute(item, "id", "1")
SetXMLNodeText(item, "Elephant")

; Create second xml node (in main node)
item = CreateXMLNode(mainNode)
SetXMLNodeName(item, "Animal")
SetXMLAttribute(item, "id", "2")
SetXMLNodeText(item, "Tiger")

```

```
; Save the xml tree into a xml file
SaveXML(xml, "demo.xml")
```

134.4 LoadXML

Syntax

```
Result = LoadXML(#XML, Filename$ [, Encoding])
```

Description

Loads a XML tree from the given file. The tree can later be accessed trough the #XML value. If #PB_Any is used as '#XML' parameter, the new XML tree number will be returned as 'Result'.

The optional 'Encoding' parameter can be used to force the parser to use a specific encoding. (This overwrites the encoding set in the XML declaration!) Possible values are #PB_Ascii, #PB_Unicode or #PB_UTF8. This parameter should be used when the document does not have an XML declaration, or the encoding information is provided outside of the XML document, for example through a mime type header in a communication protocol.

Returns nonzero if the file could be opened and read. Note that this does not mean that the XML contained in the file was valid. To check for parser errors XMLStatus() should be used. In case of a parsing error, all data parsed before the error is accessible in the XML tree.

134.5 CatchXML

Syntax

```
Result = CatchXML(#XML, *Address, Length [, Flags [, Encoding]])
```

Description

Creates a new XML tree from XML data in the given memory area. The markup can be parsed in blocks by multiple calls to this function to allow parsing XML data while it arrives from the network for example. If #PB_Any is used as '#XML' parameter, the new XML tree number will be returned as 'Result'.

'*Address' and 'Length' must specify a readable memory location. If 'Flags' is not specified, the memory location must contain all XML data. To parse XML in multiple blocks, the following flags can be used:

```
#PB_XML_StreamStart: Start parsing the first block
#PB_XML_StreamNext : Continue parsing with a new block
#PB_XML_StreamEnd  : End parsing after this block
```

When calling this function with #PB_XML_StreamStart or #PB_XML_StreamEnd, the 'Length' parameter can be 0 to start/end a parsing operation without actually parsing more data.

Note that when parsing in blocks, all data already parsed is accessible in the XML tree even before a call with #PB_XML_StreamEnd is made.

The optional 'Encoding' parameter can be used to force the parser to use a specific encoding. (This overwrites the encoding set in the XML declaration!) Possible values are #PB_Ascii, #PB_Unicode or #PB_UTF8. This parameter should be used when the document does not have an XML declaration, or the encoding information is provided outside of the XML document, for example through a mime type header in a communication protocol.

This function only returns 0 on memory errors or invalid Flags. To check for parser errors XMLStatus() should be used. In case of a parsing error, all data parsed before the error is accessible in the XML tree.

134.6 XMLStatus

Syntax

```
Result = XMLStatus(#XML)
```

Description

Returns the status of the last parsing operation done on this XML tree (using LoadXML() or CatchXML()). This function should be called after every LoadXML() or CatchXML() call to ensure that the parsing succeeded. A string representation of the parsing status (ie a readable error message) is returned by the XMLError() function.

A value of 0 (#PB_XML_Success) indicates a successful parsing, all other values indicate various error conditions.

The following returnvalues are possible:

```
#PB_XML_Success              : no error
#PB_XML_NoMemory             : out of memory
#PB_XML_Syntax               : syntax error
#PB_XML_NoElements           : no element found
#PB_XML_InvalidToken         : not well-formed (invalid token)
#PB_XML_UnclosedToken        : unclosed token
#PB_XML_PartialCharacter     : partial character
#PB_XML_TagMismatch          : mismatched tag
#PB_XML_DublicateAttribute   : duplicate attribute
#PB_XML_JunkAfterDocElement  : junk after document element
#PB_XML_ParamEntityRef       : illegal parameter entity
 reference
#PB_XML_UndefinedEntity      : undefined entity
#PB_XML_RecursiveEntityRef   : recursive entity reference
#PB_XML_AsyncEntity          : asynchronous entity
#PB_XML_BadCharacterRef      : reference to invalid character
 number
#PB_XML_BinaryEntityRef      : reference to binary entity
#PB_XML_AttributeExternalEntityRef: reference to external
 entity in attribute
#PB_XML_MisplacedXML         : XML or text declaration not at
 start of entity
#PB_XML_UnknownEncoding      : unknown encoding
#PB_XML_IncorrectEncoding    : encoding specified in XML
 declaration is incorrect
#PB_XML_UnclosedCDataSection: unclosed CDATA section
```

```
#PB_XML_ExternalEntityHandling: error in processing external
 entity reference
#PB_XML_NotStandalone        : document is not standalone
#PB_XML_UnexpectedState      : unexpected parser state
#PB_XML_EntityDeclaredInPE   : entity declared in parameter
 entity
#PB_XML_FeatureRequiresDTD   : requested feature requires
 XML_DTD support in Expat
#PB_XML_CantChangeFeatures   : cannot change setting once
 parsing has begun
#PB_XML_UnboundPrefix        : unbound prefix
#PB_XML_UndeclaringPrefix    : must not undeclare prefix
#PB_XML_IncompletePE         : incomplete markup in parameter
 entity
#PB_XML_XMLDeclaration       : XML declaration not well-formed
#PB_XML_TextDeclaration      : text declaration not well-formed
#PB_XML_PublicID             : illegal character(s) in public
 id
#PB_XML_Suspended            : parser suspended
#PB_XML_NotSuspended         : parser not suspended
#PB_XML_Aborted              : parsing aborted
#PB_XML_Finished             : parsing finished
#PB_XML_SuspendedPE          : cannot suspend in external
 parameter entity
#PB_XML_ReservedPrefixXML    : reserved prefix (xml) must not
 be undeclared or bound to another namespace name
#PB_XML_ReservedPrefixXMLNS : reserved prefix (xmlns) must
 not be declared or undeclared
#PB_XML_ReservedNamespaceURI: prefix must not be bound to one
 of the reserved namespace names
```

134.7 XMLError

Syntax

```
Result\$ = XMLError(#XML)
```

Description

In case of an error while parsing XML data this function returns an error message describing the error. XMLStatus() can be used to detect parsing errors.

To get more information about the error, XMLErrorLine() or XMLErrorPosition() can be used.

134.8 XMLErrorLine

Syntax

```
Result = XMLErrorLine(#XML)
```

Description

In case of an error while parsing XML data this function returns the line in the input that caused the error (one based). XMLStatus() can be used to detect parsing errors.

To get the position within the line at which the error happened, XMLErrorPosition() can be used.

134.9 XMLErrorPosition

Syntax

```
Result = XMLErrorPosition(#XML)
```

Description

In case of an error while parsing XML data this function returns character position within the line returned by XMLErrorLine() at which the error was caused. (The first character of the line is at position 1) XMLStatus() can be used to detect parsing errors.

134.10 SaveXML

Syntax

```
Result = SaveXML(#XML, Filename$ [, Flags])
```

Description

Saves the #XML tree to the given file.

The created XML markup is not reformated. It is written back as it was initially parsed/created. The amount of newline/whitespace written between the tags is stored in the 'text' of each XML node. (see GetXMLNodeText() for more information) To reformat the XML markup before saving, the 'text' for each XML node can be altered or FormatXML() can be used to apply some common reformating options to the tree.

'Flags' can be a combination of the following values (with the '||' operator):

```
  #PB_XML_StringFormat : Includes a byte order mark. See
   WriteStringFormat()
 for more information.
  #PB_XML_NoDeclaration: Does not include the XML declaration.
```

Note: According to the XML specification, the XML declaration can only be omitted if the document is encoded in UTF-8 or UTF-16 or if the encoding information is provided externally through a transfer protocol for example. Even then, it is advised to keep the declaration in the document.

Returns nonzero if the file was successfully saved and zero otherwise.

134.11 ExportXMLSize

Syntax

```
Result = ExportXMLSize(#XML [, Flags])
```

Description

Returns the size in bytes that will be needed to export the given XML tree to a memory buffer or to save it to a file. This function should be used to determine the needed buffersize for the ExportXML() command.

The 'Flags' parameter can be used to specify the same options as accepted by the ExportXML() command. It allows to include these options in the size calculation.

134.12 ExportXML

Syntax

```
Result = ExportXML(#XML, *Address, Length [, Flags])
```

Description

Writes the XML tree as markup to the given memory buffer .

The created XML markup is not reformated. It is written back as it was initially parsed/created. The amount of newline/whitespace written between the tags is stored in the 'text' of each XML node. (see GetXMLNodeText() for more information) To reformat the XML markup before saving, the 'text' for each XML node can be altered or FormatXML() can be used to apply some common reformating options to the tree.

'Flags' can be a combination of the following values (with the '||' operator):

```
  #PB_XML_StringFormat : Includes a byte order mark. See
   WriteStringFormat()
 for more information.
  #PB_XML_NoDeclaration: Does not include the XML declaration.
```

Note: According to the XML specification, the XML declaration can only be omitted if the document is encoded in UTF-8 or UTF-16 or if the encoding information is provided externally through a transfer protocol for example. Even then, it is advised to keep the declaration in the document.

Returns nonzero if 'Length' was large enought to hold the entire markup of the tree. ExportXMLSize() can be used to determine the needed size for this buffer.

134.13 FormatXML

Syntax

```
FormatXML(#XML, Flags [, IndentStep])
```

Description

Cleans up or reformats the XML tree for a better look when exporting / saving". It can be used to have a very compact output for efficient transfer or a more formated output for better reading.

The formating of the parsed XML document is stored in the 'text' and 'offset' fields of each node in the tree (see GetXMLNodeText() and GetXMLNodeOffset() for more information).

'Flags' can be a combination of the following values (with the '||' operator):

```
  #PB_XML_WindowsNewline: Changes all newline to CRLF
  #PB_XML_LinuxNewline  : Changes all newline to LF
  #PB_XML_MacNewline    : Changes all newline to CR

  #PB_XML_CutNewline    : Removes all newline
```

```
#PB_XML_ReduceNewline : Removes all empty lines

#PB_XML_CutSpace      : Removes all spaces
#PB_XML_ReduceSpace   : Removes all multiple spaces

#PB_XML_ReFormat      : Completely reformats the tree
 structure
#PB_XML_ReIndent      : Changes the indentation of the lines
```

For #PB_XML_ReFormat and #PB_XML_ReIndent the 'IndentStep' parameter specifies how many spaces of indentation to add for each level.

Note: There is no reformating in CData sections and Processing Instructions except for the newline changes, as the whitespace contained inside these sections may be important depending on what is contained in the section.

Note: Since MacOSX, the CR newline has become less common and the LF newline is used mostly like on other unix systems. The #PB_XML_MacNewline is provided for completeness, but it is usually better to use #PB_XML_LinuxNewline even on MacOSX.

134.14 GetXMLEncoding

Syntax

```
Result = GetXMLEncoding(#XML)
```

Description

Returns the text encoding used for exporting/saving the given XML tree.

Returns either #PB_Ascii, #PB_Unicode (= UTF16) or #PB_UTF8.

134.15 SetXMLEncoding

Syntax

```
SetXMLEncoding(#XML, Encoding)
```

Description

Changes the text encoding used for exporting/saving the given XML tree. Encoding can be either #PB_Ascii, #PB_Unicode (= UTF16) or #PB_UTF8.

Note: This only affects the exporting / saving" of the tree. The data in the #XML object is always stored in the PB string format (Ascii or Unicode depending on the compiler option). So a unicode executable can safely change the encoding to #PB_Ascii for saving and then back to something else without loosing any information in the tree in memory.

134.16 GetXMLStandalone

Syntax

```
Result = GetXMLStandalone(#XML)
```

Description

Returns the value of the "standalone" attribute in the XML declaration of the document.

Returns one of the following values:

```
#PB_XML_StandaloneYes  : The document mode is standalone
#PB_XML_StandaloneNo   : The document mode is not standalone
#PB_XML_StandaloneUnset: The standalone mode is not specified
 in the declaration
```

134.17 SetXMLStandalone

Syntax

```
SetXMLStandalone(#XML, Standalone)
```

Description

Changes the "standalone" attribute of the XML declaration when exporting/saving the document.

'Standalone' can be one of these values:

```
#PB_XML_StandaloneYes  : The document mode is standalone
#PB_XML_StandaloneNo   : The document mode is not standalone
#PB_XML_StandaloneUnset: The standalone mode is not specified
 in the declaration
```

Note: Since this library does not validate document type definitions (DTDs), the value of this attribute has no effect on the parsing/saving of documents with this library except that it is read from and written to the XML declaration. This value is however important when working with XML documents intended for validating parsers, thats why this command exists.

134.18 RootXMLNode

Syntax

```
*Result = RootXMLNode(#XML)
```

Description

Returns a pointer to the root node of the XML tree. This node is always present. It represents the XML document itself. The text contained in this node represents the whitespace outside of any XML node (there can be no text outside of nodes). The children of this node are the main node and any comments outside the main node. The type of this node is #PB_Xml_Root.

Always returns a valid XML node pointer if #XML is an existing XML tree.

134.19 MainXMLNode

Syntax

```
*Result = MainXMLNode(#XML)
```

Description

Returns the main XML node of the tree. A valid XML document must have one ”main” or ”document” node which contains all other nodes. Other than this node, there can only be comments on the first level below the root node . The type of this node is #PB_Xml_Normal.

Returns a pointer to the main node, or 0 if the tree has no main node. (which happens if the tree is empty or the main node was deleted)

134.20 ChildXMLNode

Syntax

```
*Result = ChildXMLNode(*Node [, Index])
```

Description

Returns a pointer to a child node of the given XML node. If ’Index’ is not given the first node is returned, else index specifies the one based index of the node to return.

Returns the node pointer for the requested child node or 0 if there are no children or index is to high.

134.21 ParentXMLNode

Syntax

```
*Result = ParentXMLNode(*Node)
```

Description

Returns the parent node of the given XML node. Every XML node has a parent, exept the root node .

Returns the parent node pointer or 0 if ’*Node’ was the root node.

134.22 XMLChildCount

Syntax

```
Result = XMLChildCount(*Node)
```

Description

Returns the number of child nodes inside the given XML node.

134.23 NextXMLNode

Syntax

```
*Result = NextXMLNode(*Node)
```

Description

Returns the next XML node after the given one (inside their parent node).

Returns the node pointer to the next node or 0 if there are no more nodes after the given one.

134.24 PreviousXMLNode

Syntax

```
*Result = PreviousXMLNode(*Node)
```

Description

Returns the previous XML node from the given one (inside their parent node).

Returns the node pointer to the previous node or 0 if the given node was the first child of its parent.

134.25 XMLNodeFromPath

Syntax

```
*Result = XMLNodeFromPath(*ParentNode, Path$)
```

Description

Returns the XML node inside *ParentNode who's relation to *ParentNode is described through 'Path$'. XMLNodePath() can be used to get such a path to a node.

Path$ contains a list of node names separated by '/' to indicate the way to follow from the parent to the target node. For example "childtag/subchildtag" specifies the first node with name "subchildtag" inside the first node with name "childtag" inside *ParentNode.

A node name can have an index (one based) to specify which of multiple child tags of the same name should be selected. "childtag/subchildtag[3]" specifies the 3rd "subchildtag" inside the first "childtag" of *ParentNode.

Other rules:
- If a path starts with '/' it is relative to the tree's root. No matter which node *ParentNode specifies.
- A Wildcard "*" can be used instead of a tag name to specify that any tag is to be selected.
- A Comment node has the tagname "#comment"
- A CData node has the tagname "#cdata"

- A DTD node has the tagname "#dtd"
- A Processing Instruction node has the tagname "#instruction"

Some examples of valid paths:

```
"/mainnode/#comment[4]" - the 4th comment inside the
 "mainnode" node inside the root of the tree
"*[10]"                 - the 10th node (of any type) inside
 *ParentNode
"*/*/*"                 - the 1st node 3 levels below
 *ParentNode independent of its type
"node[3]/*[3]/#cdata"   - the first CData section inside the
 3rd node of any kind inside the 3rd "node" node inside
 *ParentNode
```

Note: This command is no implementation of the XPath specification. The syntax used and understood by this command is only a small subset of XPath. This means a path returned from XMLNodePath() is a valid XPath query, but this command only understands the syntax described here, not just any XPath query.

Returns the node pointer of the target node or 0 if the path did not lead to a valid node.

134.26 XMLNodeFromID

Syntax

```
*Result = XMLNodeFromID(#XML, ID$)
```

Description

In valid XML, if a node has an attribute called "ID", the value of this attribute must be unique within the XML document. This function can be used to search for a node in the document based on its ID attribute.

Returns the node pointer of the node with the given ID tag or 0 if no such node exists within the tree.

134.27 XMLNodeType

Syntax

```
Result = XMLNodeType(*Node)
```

Description

Returns the type of the given XML node. It can be one of the following:

#PB_XML_Root

> This is the trees root node. It represents the document itself. This node cannot be created or deleted manually. Inside the root node, there can be only one node of type #PB_XML_Normal and also no plain text. (this is required to be a well-formed XML document)

#PB_XML_Normal

This is a normal node in the tree. It can have a list of attributes and contain text and/or child nodes.
Example: <node attribute="hello"> contained text </node>

#PB_XML_Comment

This node represents a comment. It can have no children or attributes. Its text represents the content of the comment.
Example: <!– comment text –>

#PB_XML_CData

This is a CData section. A CData section contains only text. Its content is not interpreted by the parser so it can contain unescaped "<" and ">" characters for example. CData sections can be used to include other markup or code inside a document without having to escape all characters that could be interpreted as XML.
Example: <![CDATA[cdata content]]>

#PB_XML_DTD

This is a document type declaration (DTD). This library does not use a validating parser, so these declarations are actually ignored when parsing a document. In order to save them back correctly, they are contained within such a DTD node. The text content of the node is the entire DTD tag. It can be read and modified through commands like SetXMLNodeText() and will be written back to the document when exporting/saving without modification. The SetXMLStandalone() command could be useful as well when working with DTDs.
Example: <!DOCTYPE name SYSTEM "external dtd uri">

#PB_XML_Instruction

This node represents a Processing Instruction. Processing Instructions contain information that is intended to be interpreted/executed by the target application. They have a name to specify the content of the instruction and the instruction data which can be accessed with GetXMLNodeText() .
Example: <?php if (...) ... ?>
(here "php" is the node name, and the rest up to the "?>" is the node text.)

134.28 GetXMLNodeText

Syntax

```
Result\$ = GetXMLNodeText(*Node)
```

Description

Returns the text inside the given XML node.

For a node of type #PB_XML_Normal, this is all text and whitespace within the node that is not contained within a child node.
For the root node , this is all whitespace outside of the main node. (there can be no text outside of the main node)
For #PB_XML_Comment or #PB_XML_CData nodes, this is all text contained in the node.

134.29 SetXMLNodeText

Syntax

```
SetXMLNodeText(*Node, Text$)
```

Description

Changes the text contained within the given XML node. See GetXMLNodeText() for more information.

Note: If the node contains children, changing its contained text may require an adjustment of the child nodes offset values as well.

134.30 GetXMLNodeOffset

Syntax

```
Result = GetXMLNodeOffset(*Node)
```

Description

Returns the character offset of this Node within its parent.

The returned value represents the number of characters in the parent nodes text data that lie between this node and the previous child node. So if this node directly follows the previous one, this value will be 0.

134.31 SetXMLNodeOffset

Syntax

```
SetXMLNodeOffset(*Node, Offset)
```

Description

Changes the character offset of the given XML node within its parent nodes text data . See GetXMLNodeOffset() for more information.

134.32 GetXMLNodeName

Syntax

```
Result\$ = GetXMLNodeName(*Node)
```

Description

Returns the tagname of the given XML node. If the node is not of type #PB_XML_Normal or #PB_XML_Instruction, an empty string is returned.

134.33 SetXMLNodeName

Syntax

```
SetXMLNodeName(*Node, Name$)
```

Description

Changes the tagname of the given XML node. If the node is not of type #PB_XML_Normal or #PB_XML_Instruction, this function is ignored.

134.34 XMLNodePath

Syntax

```
Result\$ = XMLNodePath(*Node [, *ParentNode])
```

Description

Returns a string representing the relation between *Node and *ParentNode. If *ParentNode is specified, it must be a parent or grandparent of *Node. If it is not specified, the root node of the tree is used.

See XMLNodeFromPath() for a description of the returned path string.

134.35 GetXMLAttribute

Syntax

```
Result\$ = GetXMLAttribute(*Node, Attribute$)
```

Description

Returns the value of an attribute in the given XML node. If the attribute does not exist an empty string is returned.

Only nodes of type #PB_XML_Normal can have attributes. For all other node types the compiler raises an error.

134.36 SetXMLAttribute

Syntax

```
SetXMLAttribute(*Node, Attribute$, Value$)
```

Description

Sets the value of the attribute on the given XML node. If the attribute does not exist yet, it will be added.

Only nodes of type #PB_XML_Normal can have attributes. For all other node types this function is ignored.

134.37 RemoveXMLAttribute

Syntax

```
RemoveXMLAttribute(*Node, Attribute$)
```

Description

Removes the attribute from the given XML node.

Only nodes of type #PB_XML_Normal can have attributes. For all other node types this function is ignored.

134.38 ExamineXMLAttributes

Syntax

```
Result = ExamineXMLAttributes(*Node)
```

Description

Starts to examine the attributes of the given XML node.

Returns nonzero if the node is of type #PB_XML_Normal and zero else (as such nodes cannot have attributes).

134.39 NextXMLAttribute

Syntax

```
Result = NextXMLAttribute(*Node)
```

Description

This function must be called after ExamineXMLAttributes() to move step by step through the attributes of the given XML node.

Returns zero if there are no more attributes or nonzero if there still is one.

134.40 XMLAttributeName

Syntax

```
Result\$ = XMLAttributeName(*Node)
```

Description

After calling ExamineXMLAttributes() and NextXMLAttribute() this function returns the attribute name of the currently examined attribute on the given XML node.

134.41 XMLAttributeValue

Syntax

```
Result\$ = XMLAttributeValue(*Node)
```

Description

After calling ExamineXMLAttributes() and NextXMLAttribute() this function returns the attribute value of the currently examined attribute on the given XML node.

134.42 CreateXMLNode

Syntax

```
*Result = CreateXMLNode(*ParentNode [, *PreviousNode [, Type]])
```

Description

Creates a new XML node and inserts it into the given parent node.

'*ParentNode' is the node into which to insert the new node. To insert the new node at the root of the tree, RootXMLNode() can be used here.
'*PreviousNode' specifies a childnode of '*ParentNode' after which the new node should be inserted. If this value is 0 or not specified, the new node is inserted as the first child of its parent. If this value is -1, the node is inserted as the last child of its parent.
'Type' specifies the type for the new node. The default is #PB_XML_Normal. Note that the node type cannot be changed after the node was created.

The following rules must be followed for a successful insertion:
- *ParentNode may not be of type #PB_XML_Comment or #PB_XML_CData
- *PreviousNode must be a direct child of *ParentNode (if it is specified)
- A node of type #PB_XML_Root cannot be created manually
- If the XML tree already has a main node , only nodes other than #PB_XML_Normal and #PB_XML_CData can be inserted at the root level

Returns the pointer to the new XML node if it was created successfully or 0 if no node could be inserted at this point.

134.43 CopyXMLNode

Syntax

```
*Result = CopyXMLNode(*Node, *ParentNode [, *PreviousNode])
```

Description

Copies the given XML node and all its contained text and children to a new location. This function can even be used to copy nodes into a different XML tree. For moving a complete node to a new location MoveXMLNode() can be used.

'*Node' specifies the node to copy.
'*ParentNode' is the node into which to insert the new node. To insert the new node at the root of the tree, RootXMLNode() can be used here.
'*PreviousNode' specifies a childnode of '*ParentNode' after which the new node should be inserted. If this value is 0 or not specified, the new node is inserted as the first child of its parent. If this value is -1, the node is inserted as the last child of its parent.

The following rules must be followed for a successful copying:
- The root node of a tree cannot be copied
- *ParentNode may not be of type #PB_XML_Comment or #PB_XML_CData
- *PreviousNode must be a direct child of *ParentNode (if it is specified)
- If the XML tree already has a main node , only nodes other than #PB_XML_Normal and #PB_XML_CData can be inserted at the root level

Returns the pointer to the new XML node if it was copied successfully or 0 if copying was not possible.

134.44 MoveXMLNode

Syntax

```
Result = MoveXMLNode(*Node, *ParentNode [, *PreviousNode])
```

Description

Moves the given XML node and all its contained text and children to a new location. This function can even be used to move nodes into a different XML tree. For copying a complete node to a new location CopyXMLNode() can be used.

'*Node' specifies the node to move.
'*ParentNode' is the node into which to insert the node. To insert the node at the root of the tree, RootXMLNode() can be used here.
'*PreviousNode' specifies a childnode of '*ParentNode' after which the node should be inserted. If this value is 0 or not specified, the node is inserted as the first child of its parent. If this value is -1, the node is inserted as the last child of its parent.

The following rules must be followed for a successful move:
- The root node of a tree cannot be moved
- *ParentNode may not be of type #PB_XML_Comment or #PB_XML_CData
- *PreviousNode must be a direct child of *ParentNode (if it is specified)
- *Node and *PreviousNode cannot be equal
- *ParentNode cannot be equal to, or a child of *Node (a node cannot be moved into itself)
- If the XML tree already has a main node , only nodes other than #PB_XML_Normal and #PB_XML_CData can be inserted at the root level

Returns nonzero if the move was successfull or zero if the node could not be moved.

134.45 DeleteXMLNode

Syntax

```
DeleteXMLNode(*Node)
```

Description

Deletes the given XML node and all its contained text and children from its XML tree.

Note: the root node of a tree cannot be deleted.

134.46 ResolveXMLNodeName

Syntax

```
Result\$ = ResolveXMLNodeName(*Node [, Separator$])
```

Description

Returns the expanded name of the given node in a document that uses XML namespaces. The expanded name consists of the namespace uri (if any) and the local node name, separated by the separator character given in 'Separator$'. The default separator character is "/".

In a document using namespaces, returns the expanded name of the node if it could be correctly resolved or an empty string if a namespace prefix is used that is never declared (which is invalid).

In a document without namespaces, returns the node name itself.

134.47 ResolveXMLAttributeName

Syntax

```
Result\$ = ResolveXMLAttributeName(*Node, Attribute$ [,
    Separator$])
```

Description

Returns the expanded name of the given node's attribute in a document that uses XML namespaces. The expanded name consists of the namespace uri (if any) and the local attribute name, separated by the separator character given in 'Separator$'. The default separator character is "/".

Note: Unlike with node names , the default namespace is not applied to attribute names that do not have a namespace prefix. So attribute names without a namespace prefix simply get their local name returned.

In a document using namespaces, returns the expanded name of the attribute if it could be correctly resolved or an empty string if a namespace prefix is used that is never declared (which is invalid).

In a document without namespaces, returns the attribute name itself.