

Inhaltsverzeichnis

INHALTSVERZEICHNIS	1
2. EINLEITUNG.....	6
2.1 COPYRIGHT.....	6
2.2 HAFTUNGSAUSSCHLUSS.....	6
2.3 DANKSAGUNGEN	6
1.4 VORWORT VOM AUTOR	7
1.5 VORWORT VOM VERLAG	7
1.6 EIGENSCHAFTEN VON PUREBASIC	8
1.7 ENTSTEHUNGSGESCHICHTE	10
3. INSTALLATION.....	11
3.1 LIEFERUMFANG	11
3.2 SYSTEMVORAUSSETZUNGEN.....	11
3.3 INSTALLATION	11
4. DER PUREBASIC-EDITOR.....	12
4.1 ÜBERBLICK.....	12
4.2 GRUNDLEGENDE BEDIENUNG	12
4.3 MENÜ-FUNKTIONEN	13
4.4 FUNKTIONSSCHALTER IN DER TOOLBAR-LEISTE.....	20
4.5 ONLINE-HILFE	21
5. COMPILER & DEBUGGER.....	23
5.1 EINFÜHRUNG ZUM COMPILER	23
5.2 BENUTZUNG DES COMPILERS.....	23
5.3 EINFÜHRUNG ZUM DEBUGGER.....	25
5.4 BENUTZUNG DES DEBUGGERS	26
6. GRUNDLAGEN	31
6.1 MEIN ERSTES PROGRAMM.....	31
6.2 BASIC ALLGEMEIN	32

6.3 PUREBASIC SYNTAX.....	34
6.4 END	35
6.5 GLOSSAR	36
7. DATENTYPEN.....	38
7.1 EINFÜHRUNG ZU VARIABLEN.....	39
7.2 VARIABLENTYPEN	39
7.3 DEKLARATION UND DEFAULT-DATENTYP	40
7.4 KONSTANTEN	41
7.5 DATA – ANWEISUNG	42
7.6 ARRAYS.....	44
7.7 LISTEN.....	46
7.8 STRUKTUREN.....	48
7.9 ZEIGER (POINTER)	52
8. OPERATOREN	55
8.1 DER EINFACHE ZUWEISUNGSOPERATOR '='	55
8.2 VORZEICHEN	55
8.3 ARITHMETISCHE OPERATOREN	56
8.4 VERGLEICHSOPERATOREN	59
8.5 LOGISCHE OPERATOREN.....	61
8.6 BITWEISE OPERATOREN.....	63
8.7 KLAMMERN	68
9. ABLAUFSTEUERUNG.....	70
9.1 GRUNDLAGEN UND ZWECK DER ABLAUFSTEUERUNG.....	70
9.2 BEDINGUNGEN.....	70
9.3 SCHLEIFEN.....	75
9.4 SPRUNGBEFEHLE	78
10. PROZEDUREN	82
10.1 EINLEITUNG.....	82
10.2 DEFINITION UND AUFRUF	82
10.3. GLOBALE UND LOKALE VARIABLEN	84
10.4 REKURSIVITÄT.....	87

10.5 DEKLARATION MITTELS ‚DECLARE‘	87
11. WINDOWS-PROGRAMMIERUNG	89
11.1 EINFÜHRUNG ZU WINDOWS	89
11.2 ALLGEMEINER PROGRAMMAUFBAU	90
11.3 FENSTER	91
11.4. GADGETS	93
11.5 MENÜS UND WERKZEUG-LEISTEN	100
11.6 REQUESTER	106
11.7 TASTATUR-ABFRAGE IN WINDOWS-APPLIKATIONEN	110
11.8 ABFRAGE VON WINDOWSEREIGNISSEN (MAUS, TASTATUR, GADGETS, MENÜS).....	111
11.9 ERSTELLUNG EINER KOMPLEXEN PROGRAMMOBERFLÄCHE.....	112
12 SPIELEPROGRAMMIERUNG	114
13. DIE WINDOWS-API	115
13.1 EINFÜHRUNG	115
13.2 QUELLEN UND VERWENDUNG DER WINAPI- DOKUMENTATION.....	116
13.3 EINBINDUNG VON WINAPI-FUNKTIONEN IN PUREBASIC	117
13.4 BEISPIEL FÜR UMSETZUNG VON WINAPI-CODE....	117
MESSAGEBOX	118
13.5 UMWANDLUNG VON API-DATENTYPEN	119
14. INLINE-ASSEMBLER UND DIREKTE NASM- PROGRAMMIERUNG	130
14.1 EINFÜHRUNG	130
14.2. VERWENDUNG UND SYNTAX	130
14.3 INLINEASM UND DIREKTE NASM- PROGRAMMIERUNG	131

15. ZUSÄTZLICHE INFORMATIONEN	133
15.1 ‚INCLUDE‘ FUNKTIONEN.....	133
15.2 COMPILER-DIREKTIVEN.....	135
15.3 VERWENDUNG ALTERNATIVER EDITOREN	136
15.4 QUELLEN, INSTALLATION UND NUTZUNG VON USER- LIBRARIES	139
15.5 ERSTELLUNG EINER DLL.....	139
15.6 LINKS ZU DISKUSSIONSFOREN UND SUPPORT	141
16. BEFEHLSREFERENZ.....	143
16.1 CONSOLE	146
16.2 MATH	149
16.3 STRING	151
16.4 SORT.....	155
16.5 LINKEDLIST.....	156
16.6 MEMORY	159
16.7 FILE	161
16.8 FILESYSTEM	165
16.9 MISC.....	168
16.10 PALETTE	172
16.11 IMAGE.....	173
16.12 2DDRAWING.....	176
16.13 SOUND.....	180
16.14 MODULE	183
16.15 CDAUDIO	185
16.16 MOVIE	187
16.17 SPRITE	190
16.18 SPRITE3D	202
16.19 MOUSE	206
16.20 KEYBOARD	208
16.21 JOYSTICK	210
16.22 WINDOW.....	212
16.23 FONT.....	220
16.24 GADGET.....	221

16.25 MENU	239
16.26 REQUESTER	242
16.27 TOOLBAR	245
16.28 STATUSBAR.....	247
16.29 CLIPBOARD.....	248
16.30 PREFERENCE	249
16.31 PRINTER.....	252
16.32 NETWORK.....	254
16.33 HELP.....	258
16.34 DATABASE.....	259
16.35 PACKER	263
16.36 CIPHER	266
16.37 SYSTRAY	268
16.38 THREAD.....	269
16.39 LIBRARY	270
17. FEHLERMELDUNGEN	273
18. INDEX	322

2. Einleitung

2.1 Copyright

Fantaisie Software und der TOPOS-Verlag behalten sich alle Rechte an diesem Programm und allen Original-Archiven und deren Inhalten vor.

2.2 Haftungsausschluss

Dieses Programm wird bereitgestellt "WIE ES IST". Fantaisie Software und der TOPOS-Verlag sind nicht verantwortlich für irgendwelche Schäden, die PureBasic zugeschrieben werden. Garantien werden von Fantaisie Software oder vom TOPOS Verlag oder einem Vertreter weder gegeben noch angedeutet.

2.3 Danksagungen

Wir möchten uns bei vielen Leuten bedanken, die uns bei diesem ambitionierten Projekt geholfen haben. Es wäre nicht ohne sie möglich gewesen!

- *Registrierte Benutzer:* Für die Unterstützung dieser Software... Vielen Dank !
- *Roger BEAUSOLEIL:* Der Erste, der an dieses Projekt glaubte, und für seine unschätzbare Hilfe beim Grund-Design und Layout von PureBasic.
- *Andre BEER:* Für seine Zeit, alle Guides ins Deutsche zu übersetzen. Vielen Dank.
- *Francis G. LOCH:* Welcher alle Fehler in den englischen Guides korrigierte! Danke noch einmal.
- *Steffen HAEUSER:* Für seine wertvolle Zeit und Hilfe. Für seine unschätzbaren Erklärungen betreffs der Aspekte der PPC Programmierung und seinen sehr nützlichen Tipps.
- *Martin KONRAD:* Ein fantastischer Bug-Reporter für die Amiga-Version. Für die Erstellung der zukünftigen visuellen Entwicklungsumgebung für PureBasic und einige nette Spiele...
- *James L. BOYD:* Für das Auffinden von tonnenweise Bugs in der x86 Version, der uns damit auch tonnenweise Arbeit verschaffte :). Bleib aktiv bei der Bug-Suche!
- *LES:* Für das Erstellen der englischen Fantaisie Software Website & und dem PureBasic-Manual. Dieses sieht viel besser aus!
- *NAsm Team:* Für den unglaublichen x86 Assembler, der für die

- Entwicklung der PureBasic Bibliotheken benutzt wurde.
- *Jacob NAVIA*: Für den exzellenten Linker (aus dem Lcc32 Paket), der zurzeit von PureBasic benutzt wird.
 - *Thomas RICHTER*: Für die Erschaffung von "PoolMem." Einem Patch, der die Zeit für das Kompilieren um den Faktor 2-3 verkürzte! Vielen Dank für seine Erlaubnis, den Patch zum Hauptverzeichnis von PureBasic hinzufügen zu dürfen.
 - *Frank WILLE*: Für die Erlaubnis, Deine ausgezeichneten Assembler "PAsm" und "PhxAss" benutzen zu dürfen. Für all Deine Tipps und die unschätzbare Hilfe beim Gebrauch der Tools. Für seine großartige Hilfe beim Debuggen der PowerPC Executables.
 - *Danilo, Rings, Paul, Franco und MrVain/Secretly*: Für ihre intensiven Bug-Reports und ihre Hilfe, den Compiler so wertvoll und bug-frei wie möglich zu machen.
 - *Timo "Freak" Harter*: für die Umsetzung der WindowsAPI-Datentypen in PureBasic

2.4 Vorwort vom Autor

PureBasic wurde geschaffen, um es jedem Programmierer – vom Anfänger bis zum erfahrenen Anwender – zu ermöglichen, alles aus seinem Computer herauszuholen. Es erlaubt das Entwickeln von sehr professionellen und mächtigen Applikationen genauso wie Spielen und erstellt sehr kleine und hoch optimierte ausführbare Programme. Und am besten: PureBasic ist Cross-plattform kompatibel: zurzeit läuft es auf Linux, Windows und Amiga.

Vielen Dank dass Sie sich für PureBasic entschieden haben!

Frederic Laboureur.

2.5 Vorwort vom Verlag

Herzlich willkommen und vielen Dank, dass Sie sich für unser Produkt „PureBasic“ entschieden haben. Dieses innovative Produkt soll aufräumen mit dem Vorurteil, dass mittels BASIC-Programmiersprachen erstellte Programme groß, langsam und in vielen Punkten eingeschränkt sind.

Mit diesem Handbuch wollen wir Ihnen die Grundlagen der BASIC-Programmierung allgemein und im speziellen die einfachen aber mächtigen Funktionen von PureBasic näher bringen, damit Sie schon bald in der Lage sind, eigene professionelle Applikationen und Spiele mit dieser

faszinierenden Sprache zu erstellen.
Viel Spaß bei der Programmierung mit PureBasic!

André Beer
Autor

Ralf Kraft
TOPOS Verlag

2.6 Eigenschaften von PureBasic



PureBasic ist eine neue "Hochsprachen" Programmiersprache, welche auf den bekannten BASIC-Regeln basiert. Sie ist vergleichbar mit jedem anderen BASIC-Compiler, den Sie vielleicht schon benutzt haben, egal ob für den Amiga oder den PC. PureBasic ist sehr leicht zu erlernen! Es wurde für Anfänger genauso wie für Experten geschaffen. Die Übersetzungsgeschwindigkeit ist extrem schnell.

Die Ihnen vorliegende Software wurde für das Windows Operating-System entwickelt. Wir haben eine Menge Anstrengungen in ihre Realisierung gesetzt, um eine schnelle, zuverlässige und systemfreundliche Sprache zu produzieren.

Die Syntax ist einfach, aber die gebotenen Möglichkeiten sind endlos, mit PureBasic's fortgeschrittenen Funktionen, wie Zeiger, Strukturen, Prozeduren, dynamisch verknüpften Listen und vielem mehr. Für den erfahrenen Programmierer gibt es keine Probleme, Zugang zu irgendwelchen legalen OS-Strukturen oder API-Objekten zu bekommen.

PureBasic ist eine portable Sprache, welche zurzeit auf AmigaOS (680x0 und PowerPC), Linux und Windows Computersystemen arbeitet. Dies bedeutet, dass derselbe Programmcode für jedes System nativ kompiliert werden kann und trotzdem die volle Power eines jeden ausnutzt. Es gibt keine Flaschenhälse wie einen virtuellen Prozessor oder einen Code-Übersetzer; der generierte Code produziert ein optimiertes Executable, ungeachtet des OS, auf welchem es kompiliert wird. Die externen Bibliotheken sind vollständig in handoptimiertem Assembler, welcher sehr schnelle Befehle - oftmals schneller als die C/C++ Äquivalente - erzeugt, geschrieben.

Die bedeutendsten Features von PureBasic

- 486, Pentium (Pro, II, III, IV) Support
- Eingebaute Arrays, dynamisch verknüpfte Listen, komplexe Strukturen, Zeiger und Variablen-Definitionen
- Unterstützte Typen: Byte (8 Bit), Word (16 Bit), Long (32 Bit), Float (32 Bit) und auch benutzerdefinierte Typen (Strukturen)
- Eingebaute String-Typen (Charaktere)
- Konstanten, binäre und hexadezimale Zahlen werden unterstützt.
- Verkürzung von Ausdrücken (Zusammenstellen von Konstanten und numerischen Zahlen).
- Standard Arithmetik-Unterstützung unter Beachtung von Vorzeichen und Klammern sowie logischen und bitweisen Operatoren
- Sehr schnelle Kompilierung (über 300.000 Zeilen/Minute bereits auf einem P200).
- Unterstützung von Prozeduren für strukturiertes Programmieren mit lokalen und globalen Variablen.
- Alle Standard BASIC Schlüsselworte: If-Else-EndIf, Repeat-Until, etc.
- Unterstützung von externen Bibliotheken, um Objekte wie BMP-Bilder, Fenster, Gadgets, DirectX etc. einfach zu integrieren.
- Externe Bibliotheken wurden vollständig in handoptimiertem Assembler geschrieben, um Ihnen maximale Geschwindigkeit und Kompaktheit zu bieten.
- Die Win32 API Funktionen werden vollständig unterstützt, so als wären es BASIC Schlüsselworte.
Inline-Assembler
- Vorkompilierte Strukturen mit Konstanten-Dateien für extra-schnelle Kompilierung.
- Konfigurierbarer CLI-Compiler.

- Sehr hohe Leistungsfähigkeit, ausführliche Stichwörter, Online-Hilfe.
- Systemfreundlich, leicht zu installieren und zu benutzen.

2.7 Entstehungsgeschichte

Markante Stationen auf dem Weg zu der Ihnen vorliegenden Version von PureBasic:

01. September 1999	Erste öffentliche Version von <i>PureBasic v1.00</i> für AmigaOS
-----------------------	--

22. Oktober 2000	Erstes öffentliches Release von <i>PureBasic v2.00</i> für x86/Windows (nur für Testzwecke)
---------------------	---

17. Dezember 2000	Veröffentlichung des endgültigen Release von <i>PureBasic v2.00</i>
----------------------	---

23. September 2001	Veröffentlichung von <i>PureBasic v2.50</i> für x86/Windows
-----------------------	---

4. April 2002	Veröffentlichung von <i>PureBasic v3.00</i> für x86/Windows
---------------	---

9. Juni 2002	Veröffentlichung von <i>PureBasic v3.20</i> für x86/Windows
--------------	---

3. Installation

3.1 Lieferumfang

Zusammen mit diesem Handbuch erhielten Sie eine CD-ROM mit PureBasic und Beispielprogrammen, diesem umfangreichen Handbuch, ein Handbuch zur Programmierung von Strategiespielen auf CD-ROM und zahlreichen Grafiken und Animationen zur Verwendung in Ihren eignen Spielen.

3.2 Systemvoraussetzungen

PureBasic läuft auf jedem Windows-System (das bedeutet: Windows 95/98/ME, Windows NT/2000 und Windows XP). Für einzelne Bibliotheken ist zusätzlich mindestens eine installierte Version 7 von DirectX erforderlich.

Falls irgendwelche Probleme auftreten, informieren Sie uns bitte.

3.3 Installation

Zum Installieren von PureBasic ziehen Sie einfach die 'PureBasic' Schublade dorthin, wo auch immer Sie möchten. Es sind keine weiteren Vorkehrungen notwendig. Um den PureBasic-Editor zu starten, klicken Sie einfach auf das Programm-Icon.

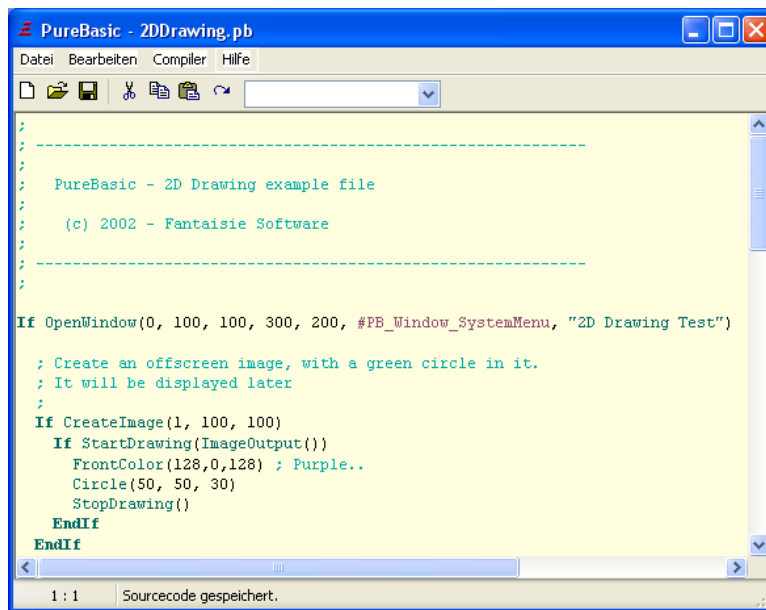
Um den Shell-basierten Compiler zu benutzen, öffnen Sie ein Konsolenfenster (auch „MSDOS – Fenster“) und schauen in den Unterordner Compilers\ nach der Anwendung PBCompiler.

PureBasic verwendet für seine Programmcode-Dateien die Endung „.pb“ und assoziiert nach dem ersten Programmstart automatisch alle Dateien mit dieser Endung mit dem PureBasic-Editor. Das bedeutet, dass nach einem Doppelklick auf eine solche Programmcode-Datei, der Editor gestartet und die Datei geladen wird.

4. Der PureBasic-Editor

4.1 Überblick

Der PureBasic Editor (Datei „PureBasic.exe“ im Verzeichnis „PureBasic/“) wurde speziell für die PureBasic Programmiersprache entwickelt und enthält viele spezielle Features, die extra dafür geschaffen wurden.



4.2 Grundlegende Bedienung

Der PureBasic Editor akzeptiert alle Standard ASCII-Zeichen, und lädt und speichert die Dateien im ASCII-Format. Er benutzt alle Windows Standard Shortcuts (Tastenkürzel) zum Editieren des Textes:

Pfeiltasten	: Bewegen den Cursor in die vier Richtungen.
Shift + Pfeiltasten	: Markieren den Text in die jeweilige Richtung.
Strg + Pos1	: Springt an den Anfang des Sourcecodes.
Strg + Ende	: Springt an das Ende des Sourcecodes.

Das Scrollen im Sourcecode ist auch einfach mit der Maus über die Scrollbalken am Fensterrand möglich. Ebenso das Markieren eines Text-Abschnitts.

Der Editor hat eine "Auto-Ident" Funktion (automatisches Einrücken), was den Cursor immer innerhalb des aktuell eingerückten Blocks hält und somit einfaches Bearbeiten des Sourcecodes ermöglicht.

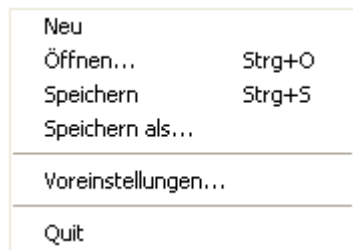
PureBasic-Dateien werden standardmäßig mit der Dateiendung „.pb“ abgespeichert. Da diese Datei-Endung automatisch mit dem PureBasic-Editor verknüpft wird, können Sourcecode-Dateien durch einfachen Doppelklick auf die entsprechende Datei in den PureBasic-Editor geladen werden.

4.3 Menü-Funktionen

Der PureBasic-Editor besitzt vier Menüs: Datei, Bearbeiten, Compiler und Hilfe.

Die einzelnen Menüpunkte werden in nachfolgender Tabelle vorgestellt. Sofern ein Menü-Shortcut (Möglichkeit, den Menüpunkt per Tasten-Kombination aufzurufen) existiert, sehen Sie diesen im Bildschirmausschnitt auf der linken Seite. Z.B. für das Öffnen einer Datei „STRG + O“ oder für das Kompilieren des Sourcecode und Starten des erzeugten Programms „F5“.

Menü Datei



Neu

Löscht einen evtl. vorhandenen Editor-Inhalt und ermöglicht die Erstellung einer neuen PureBasic Sourcecode-Datei.

Öffnen

Öffnet eine vorhandene PureBasic Sourcecode-Datei und zeigt deren Inhalt im Editor an.

Speichern

Speichert den aktuellen Sourcecode ohne weitere Nachfrage. Falls noch kein Dateiname vergeben wurde, wird automatisch der Datei-Auswahlrequester aufgerufen.

Speichern als

Speichert den aktuellen Sourcecode in die Datei, welche von Ihnen über den Datei-Auswahlrequester ausgewählt wurde.

Voreinstellungen

Ruft das entsprechende Fenster auf, in dem weitere Einstellungen zum Editor gemacht werden können.

Quit

Schließt den Editor (ggf. nach vorheriger Sicherheitsabfrage, falls der Sourcecode verändert wurde).

Menü Bearbeiten

Undo	Strg+Z
Redo	Strg+Y
Ausschneiden	Strg+X
Kopieren	Strg+C
Einfügen	Strg+V
Gehe zu...	Strg+G
Suchen...	Strg+F
Weitersuchen	F3

Undo

Macht die letzte(n) Änderung(en) im Sourcecode rückgängig.

Redo

Wiederholt die letzte(n) Änderung(en) nach einem Undo.

Ausschneiden

Entfernt den markierten Sourcecode und kopiert ihn in die Zwischenablage.

Kopieren

Kopiert den markierten Sourcecode in die Zwischenablage ohne ihn aus dem Editor zu entfernen.

Einfügen:

Fügt den gerade in der Zwischenablage vorhandenen Text (i.d.R. Sourcecode) an der aktuellen Cursor-Position im Editor ein.

Gehe zu

Ermöglicht die Eingabe einer Zeilennummer und springt nach Bestätigung mit „OK“ zu der betreffenden Zeile im Editor.

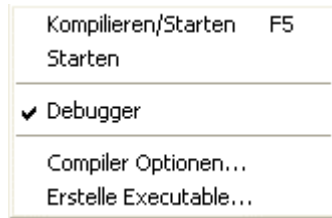
Suchen

Ruft ein Fenster mit den auch aus anderen Editoren bekannten „Suchen & Ersetzen“ Funktionen auf.

Weitersuchen

Springt zum nächsten Vorkommen des im „Suchen & Ersetzen“ eingegebenen Suchbegriffs.

Menü Compiler



Kompilieren/Starten

Kompiliert und startet den aktuellen Sourcecode.

Starten

Startet das bereits kompilierte Programm, ohne nochmaliges Kompilieren.

Debugger

Schaltet den eingebauten Runtime-Debugger ein oder aus. Ein eingeschalteter Debugger wird durch ein vorangestelltes Häkchen gekennzeichnet.

Compiler Optionen

Ruft das entsprechende Fenster auf, in dem weitere Einstellungen zum Compiler getroffen werden können.

Erstelle Executable

Ruft einen Datei-Auswahlrequester auf, in dem die gewünschte Datei festgelegt werden kann. Danach wird der aktuelle Sourcecode kompiliert und das erstellte Executable (ausführbare Programm) unter dem ausgewählten Dateinamen abgespeichert.

Menü Hilfe



Hilfe

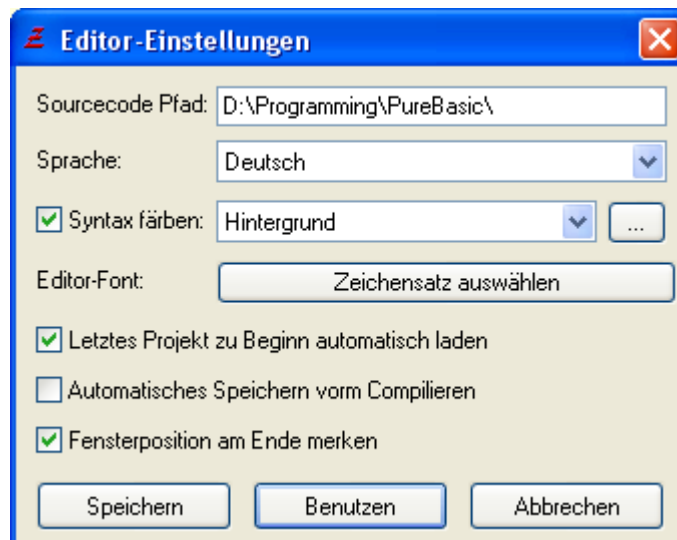
Ruft das Referenz-Handbuch in der PureBasic Online-Hilfe auf. Wird „F1“ über einem PureBasic oder Windows® API Befehl gedrückt, wird direkt die Beschreibung des entsprechenden Befehls aufgerufen.

Info

Öffnet ein Fenster mit Informationen zur aktuellen Version und den Autoren von PureBasic.

Menüpunkt „Voreinstellungen“

Hier werden wie schon gesagt, grundlegende Merkmale zum Aussehen und Verhalten des PureBasic – Editors festgelegt.

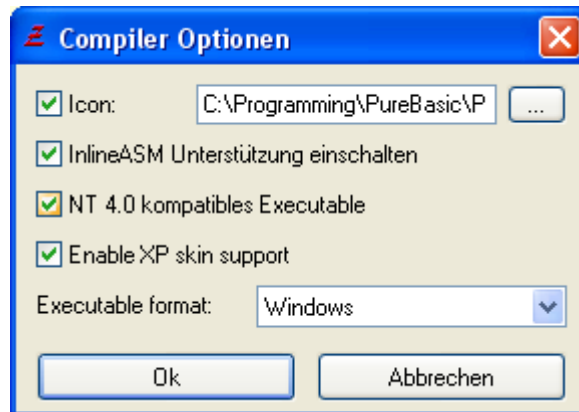


- *Sourcecode Pfad*: Standard-Verzeichnis, dass nach dem Starten des Editors eingestellt ist.

- *Sprache*: Landessprache, die für alle Schalter- und Menü-Beschriftungen verwendet wird.
- *Syntax färben*: Über das Aufklapp-Menü und dem rechts daneben liegenden Schalter können dem Hintergrund, dem normalem Text sowie den verschiedenen PureBasic Objekten, wie Befehlen, Konstanten, Schlüsselwörter usw., verschiedene Farben zugeordnet werden.
- *Editor-Font*: Mit dem Anklicken von „Zeichensatz auswählen“ kann der im Editor zu benutzende Zeichensatz festgelegt werden. Empfehlenswert ist „Courier New“.
- *Letztes Projekt zu Beginn automatisch laden*: Wenn dieser Punkt markiert ist, wird beim Starten des PureBasic-Editors automatisch der zuletzt bearbeitete Sourcecode geladen.
- *Automatisches Speichern vorm Compilieren*: Wenn mit „F5“ oder dem entsprechenden Menüpunkt „Kompilieren/Starten“ der Sourcecode kompiliert wird, wird automatisch vorher der aktuelle Sourcecode gespeichert.
- *Fensterposition am Ende merken*: Beim Schließen des Editors wird die aktuelle Fensterposition und -größe für den nächsten Programmstart gespeichert.
- *Speichern*: Übernimmt die aktuellen Einstellungen und speichert diese für den nächsten Programmstart. Das Einstellungs-Fenster wird geschlossen.
- *Benutzen*: Übernimmt die aktuellen Einstellungen, speichert diese jedoch nicht. Sie sind damit für den nächsten Programmstart verloren. Das Einstellungs-Fenster wird geschlossen.
- *Abbrechen*: Schließt das Einstellungs-Fenster ohne die vorgenommenen Einstellungen zu verwenden.

Menüpunkt „Compiler Optionen“

Hier sind weitere Einstellungen möglich, die das Verhalten von PureBasic beim Kompilieren und die Art des erstellten Programms beeinflussen.



OK

Bestätigt die gemachten Einstellungen und schließt das Fenster.

Abbrechen

Schließt das „Compiler Optionen“ Fenster, ohne die gemachten Einstellungen zu übernehmen.

Icon:

Falls ausgewählt, wird ein Icon zum erstellten Executable hinzugefügt. Das Icon wird im Auswahlrequester, der über den rechts gelegenen Schalter aufgerufen wird, ausgewählt.

InlineASM

Unterstützung einschalten: ermöglicht die Benutzung von ASM Schlüsselwörtern direkt im PureBasic Sourcecode.

NT 4.0

NT 4.0 kompatibles Executable: ermöglicht einer Multimedia-Applikation oder einem Spiel (welches DirectX benutzt) das problemlo-

se Laufen unter NT4. In diesem Fall wird DirectX 3 anstelle von DirectX 7 benutzt (hierzu wird das Service-Pack 3 unter NT4 benötigt).

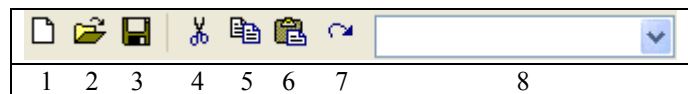
Windows XP Skin Support

Aktiviert den Skin-Support unter MS Windows XP. Hierfür wird automatisch eine Datei „DeinProgramm.manifest“ erstellt.

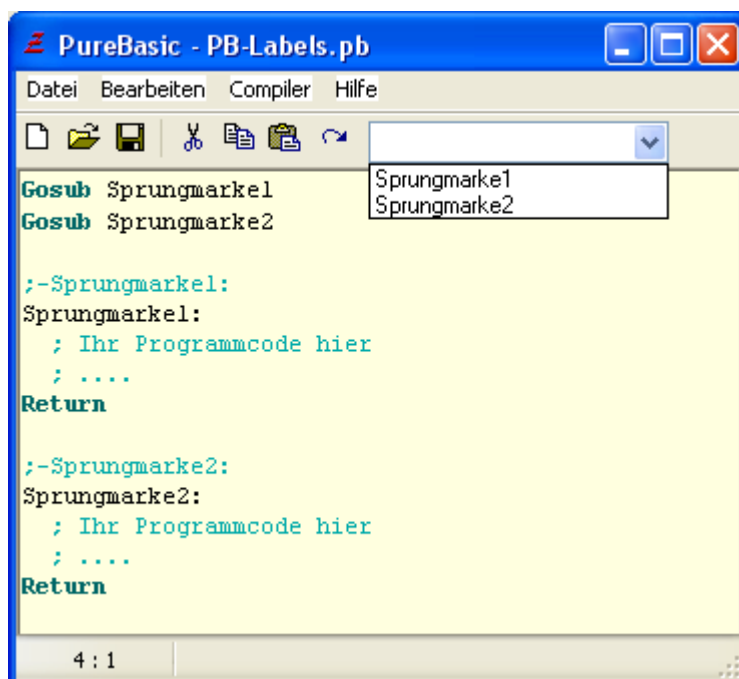
Executable Format:

Hier stehen drei Typen zur Verfügung – Windows, Console und Shared DLL. Damit kann bestimmt werden, ob das Executable ein ausführbares Programm für Windows®, für das MSDOS-Konsolenfenster oder eine Funktionsbibliothek (DLL) werden soll.

4.4 Funktionsschalter in der Toolbar-Leiste



- 1) entspricht der Option „Neu“ in den Menü-Funktionen
- 2) entspricht der Option „Öffnen“ in den Menü-Funktionen
- 3) entspricht der Option „Speichern“ in den Menü-Funktionen
- 4) entspricht der Option „Ausschneiden“ in den Menü-Funktionen
- 5) entspricht der Option „Kopieren“ in den Menü-Funktionen
- 6) entspricht der Option „Einfügen“ in den Menü-Funktionen
- 7) entspricht der Option „Kompilieren & Starten“ in den Menü-Funktionen
- 8) ist ein Aufklapp-Menü für Sprungmarken im Sourcecode. Dieses ist nützlich für die Navigation in sehr umfangreichen Sourcecodes und wird wie folgt benutzt:



Fügen Sie einfach vor den Unterprogrammen in Ihrem Sourcecode, die Sie mit Hilfe von „Gosub“ und den entsprechenden Sprungmarken anspringen, eine Kommentarzeile ein. Wie Sie in nebenstehendem Beispiel sehen können, werden alle Kommentarzeilen, die mit Semikolon und nachfolgendem Bindestrich eingeleitet (z.B. „;-Sprungmarke1:“ werden, im Sprungmarken-Aufklappenmenü angezeigt. Durch Auswahl eines Eintrags im Aufklappenmenü springt der Cursor im Editor zur entsprechenden Zeile im Sourcecode.

Die Nutzung dieser Funktion ist natürlich nicht ausschließlich auf Unterprogramme beschränkt. Entsprechende „Sprungmarken“ können an jeder wichtigen Stelle im Sourcecode eingefügt werden.

4.5 Online-Hilfe

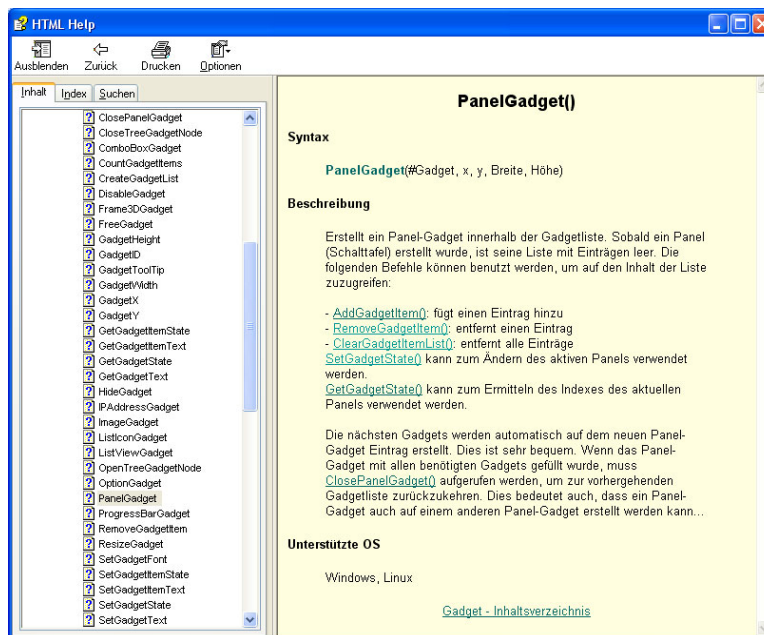
Die Online-Hilfe zu PureBasic wird in der Datei „PureBasic.chm“ im PureBasic Hauptverzeichnis mitgeliefert. Darin sind alle Beschreibungen zu den PureBasic Befehlen, Schlüsselwörtern und Syntax-Konventionen enthalten.

Erreichbar ist die Online-Hilfe über das Menü „Hilfe“ im PureBasic-

Editor oder über Druck auf die „F1“-Taste.

Die Benutzung der Taste „F1“ ist auch besonders empfehlenswert. Denn befindet sich der Cursor im PureBasic-Editor über einem Befehl, verzweigt die Online-Hilfe direkt zu dessen Beschreibung.

Hier ein Beispiel zum PanelGadget:



Dies funktioniert nicht nur mit den eigentlichen PureBasic-Befehlen, sondern auch mit den vielen Funktionen der Windows API (Hilfedatei „Win32.hlp“) sowie Befehlen von Dritt-Anbietern. Die Hilfe-Dateien müssen sich hierbei jeweils im Verzeichnis „PureBasic/Help“ befinden.

5. Compiler & Debugger

5.1 Einführung zum Compiler

Das Programm „PBCompiler.exe“ im Verzeichnis „PureBasic\Compilers“ ist der eigentliche Compiler von PureBasic, der aus dem Sourcecode ein ausführbares Programm erstellt. Auch der PureBasic-Editor benutzt diesen Compiler für seine integrierten Funktionen zum Kompilieren und Starten von Programmen.

Der PureBasic Shell-Compiler kann entweder in der Windows-Konsole („Eingabeaufforderung“ im Windows-Start-Menü im Menü „Programme/Zubehör“) zum Einsatz kommen oder bei Verwendung von alternativen Editoren von diesen aufgerufen werden.

5.2 Benutzung des Compilers

Durch das Hinzufügen des Verzeichnisses „...\PureBasic\Compilers\“ zur Pfad-Variable von Windows ist der einfache Zugriff auf PureBasic-Compiler gewährleistet, ohne jedes Mal erst dessen Verzeichnis aufrufen zu müssen.

Verschiedene Optionen – einzeln oder auch kombiniert – können beim Start des Compilers angegeben werden:

<i>Option</i>	<i>Zugehörige Funktion</i>
<i>/?</i>	Zeigt einen kurzen Hilfstext über den Compiler an.
<i>/COMMENTED</i>	Erstellt eine kommentierte '.asm' Ausgabedatei beim Erstellen des Executable. Diese Datei kann später re-assembliert werden, wenn die notwendigen Modifikationen erfolgt sind. Diese Option ist nur für fortgeschrittene Programmierer bestimmt.
<i>/DEBUGGER</i>	Schaltet die Debugger Unterstützung ein.
<i>/EXE "Dateiname"</i>	Erstellt ein ausführbares Programm ('standalone Executable') mit dem angegebenen Dateinamen.
<i>/ICON "IconName"</i>	Fügt das angegebene Icon zum Executable hinzu.
<i>/INLINEASM</i>	Schaltet das InlineASM Feature ein (ASM-Routinen können direkt im BASIC - Sourcecode geschrieben werden).
<i>/RESIDENT "Datei-</i>	Erstellt eine Resident-Datei mit dem angegebe-

name"	nen Dateinamen. Damit können Sie sich selbst eine Datei mit eigenen vordefinierten Konstanten und Strukturen erstellen. Wenn Sie die erstellte Datei (achten Sie auf die Endung .res) nach „PureBasic\Residents\“ kopieren, steht Ihnen der Dateinhalt jederzeit in PureBasic zur Verfügung extra eine Include-Datei benutzen zu müssen. Achten Sie jedoch unbedingt darauf, für Konstanten und Strukturen keine Namen zu verwenden, die bereits in PureBasic vorhanden sind!
/CONSOLE	Erstellt das Executable im Konsole Format. Das Standard-Format ist Win32.
/NT4	Schaltet die Windows NT4 Kompatibilität für Multimedia-Funktionen (Sprite, Keyboard, Mouse) ein.
/DLL	Als Ausgabedatei wird eine DLL erstellt.
/REASM	Re-Assembliert die PureBasic.asm Datei in ein Executable. Dies ermöglicht die Benutzung der /COMMENTED Funktion, das anschließende Modifizieren der ASM-Ausgabe und das erneute Kompilieren des Executable.
/QUIET	Schaltet alle unnötige Text-Ausgabe ab, sehr nützlich bei Verwendung eines anderen Editors.

Beispiel

Wir möchten den Sourcecode des Beispiels zur 2DDrawing Befehlsbibliothek kompilieren:

```

G:\Programming\PureBasic\Compilers>pbcompiler "c:\programming\purebasic\examples\sources\2ddrawing.pb"
*****
PureBasic v3.20 - Windows x86
*****
Compiling c:\programming\purebasic\examples\sources\2ddrawing.pb
Loading external libraries...
Starting compilation...
76 lines processed.
Creating executable.
- Feel the ..PuRe.. Power -
G:\Programming\PureBasic\Compilers>

```

Wie Sie sehen, muss das Verzeichnis des PureBasic-Compilers aktiv sein (sofern nicht unser Beispiel-Pfad

`„C:\Programming\PureBasic\Compilers“`

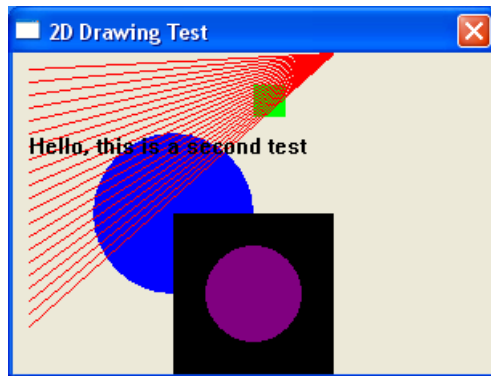
dem Windows-Betriebssystem durch die Pfad-Variable bekannt und damit der PureBasic-Compiler von überall startbar ist).

Durch den Aufruf von:

`pbcompiler "c:\programming\purebasic\examples\sources\2ddrawing.pb"`

wird der Sourcecode geladen, kompiliert und gestartet.

Wenn Sie damit erfolgreich waren, müsste jetzt folgendes Fenster sichtbar werden:



Beispiel

1.) `PBCompiler Programm.pb /EXE „Programm.exe“`

Der Compiler kompiliert den Sourcecode „Programm.pb“ und erstellt daraus ein ausführbares Programm „Programm.exe“.

2.) `PBCompiler Programm.pb /DEBUGGER /INLINEASM`

Der Compiler kompiliert den Sourcecode „Programm.pb“ und führt ihn mit Debugger- und Inline-Assembler Unterstützung aus.

5.3 Einführung zum Debugger

Debuggen könnte man auf Deutsch mit „Suchen von Fehlern“ übersetzen. Fehler im Programmcode werden im international üblichen Englisch auch als „Bug“ (Mehrzahl: „Bugs“) bezeichnet.

Was macht nun ein Debugger? Der Debugger ist – wie Sie vielleicht schon vermuten – sozusagen das Hilfsmittel für den Programmierer zum „Aufspüren der Fehler (Bugs)“ im Programmcode.

Die Fehler aufzuspüren, das ist Aufgabe des PureBasic Runtime-Debuggers, einem externen Programm, das während der Programmausführung (also in Echtzeit oder auf englisch „Runtime“) das kompilierte

Programm auf mögliche Fehler überprüft und dem Programmierer durch eine entsprechende Mitteilung Art des Fehlers und die Stelle im Programmcode mitteilt. Der Debugger überwacht und kontrolliert z.B. alle den Funktionen übergebene Parameter, verhindert verbotene Operationen (wie Division durch Null) usw.

Mit PureBasic kann die Programmausführung angehalten und Schritt für Schritt analysiert werden, um alle Fehler aufzuspüren; und sie kann zu jeder Zeit unterbrochen werden! Dies kann sehr nützlich sein, wenn ein Programm in eine Endlosschleife verfällt. Und noch mehr, der Debugger erlaubt auch das Überwachen („Monitoring“) von beliebigen Variablen und sogar ASM-Registern, wenn 'InlineASM Unterstützung einschalten' im Compiler-Optionen-Menü aktiviert ist.

5.4 Benutzung des Debuggers

Damit der Debugger beim Kompilieren und Starten Ihres Programmes aktiv ist, muss im PureBasic-Editor der Menüpunkt „Compiler – Debugger“ mit einem Häkchen versehen sein. Falls dies noch nicht der Fall ist, markieren Sie bitte den entsprechenden Menüpunkt.

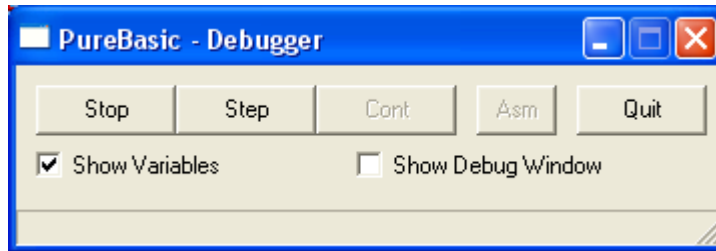
Wir wollen uns nachfolgend am Beispielprogramm „Window.pb“ aus dem Verzeichnis „...\\PureBasic\\Examples\\Sources“ die Funktionsweise des Debuggers genauer ansehen. Bitte laden Sie deshalb die genannte Datei in den Editor.

Folgender Programmcode (in etwas modifizierter Form, d.h. zuzüglich weiterer Leer- und Kommentarzeilen und einiger Konstanten a la „PB_xxx“) wird jetzt im Editorfenster angezeigt:

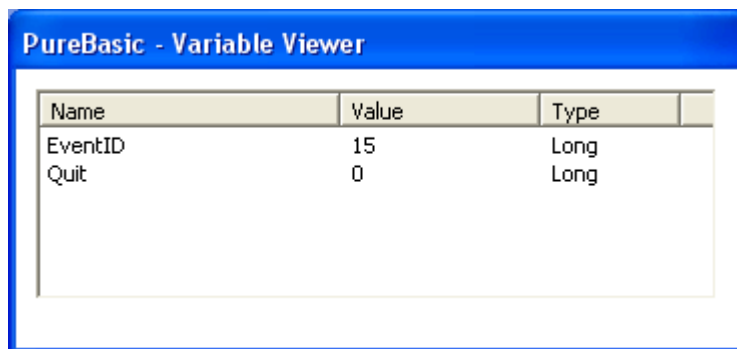
```
If OpenWindow(0, 100, 100, 195, 260, #PB_Window_SystemMenu, „
    PureBasic Window“)
    MessageRequester("Information", "Click to move the ↵
    Window", 0)
    MoveWindow(200,200)
    MessageRequester("Information", "Click to resize the ↵
    Window", 0)
    ResizeWindow(320,200)
    Repeat
        EventID.1 = WaitWindowEvent()
        If EventID = #PB_Event_CloseWindow
            Quit = 1
        EndIf
    Until Quit = 1
EndIf
End
```

Drücken Sie ‚F5‘ zum Kompilieren und Starten des Programms. Neben

dem eigentlichen Programmfenster „PureBasic Window“ bekommen wir von PureBasic jetzt auch das „PureBasic – Debugger“ Fenster zu Gesicht:



Nach Markierung der Funktion „Show Variables“ (Zeige Variablen) erhalten wir folgendes Ausgabefenster:



Hierin werden alle im Programmcode verwendeten Variablen und deren aktuelle Werte angezeigt. In unserem Beispiel die Variablen ‚EventID‘ und ‚Quit‘, die in der Schleife zwischen „Repeat ... Until“ zur Verwendung kommen.

Funktionen der einzelnen Schalter im Debugger-Fenster:

<i>Stop</i>	Stoppt die laufende Programmausführung und zeigt die aktuelle Position im Programmcode an.
<i>Step</i>	Dieser Schalter ermöglicht den Code nach und nach abzuarbeiten, d.h. Zeile für Zeile. Dies ist sehr nützlich, um Fehler aufzuspüren.
<i>Cont</i>	Setzt ein zuvor mittels ‚Stop‘ angehaltenes Programm fort.
<i>Asm</i>	Zeigt ein ASM Monitoring Fenster mit allen CPU-Registern und CPU-Flags. Dies ist nur für erfahrene Programmierer, welche die ASM Spra-

che bereits kennen. Die 'Inline ASM Unterstützung einschalten' Option im Compiler-Menü muss aktiviert sein und der Debugger muss angehalten (mit STOP oder STEP) worden sein, um Zugriff auf dieses Fenster zu erhalten. Dieser Monitor ist nur nützlich beim Betrachten von Inline ASM, nicht bei normalem Basic Code. Ein cooles Feature ist die Möglichkeit, den Wert eines beliebigen Registers nach Drücken des Schalters 'Schreibe neuen Wert' verändern zu können.

<i>Quit</i>	Beendet den Debugger, den Compiler und jedes Programm im Falle von Problemen oder wenn eine "Endlosschleife" auf keinem anderen Weg angehalten werden kann.
<i>Show Variables</i> („Zeige Variablen“)	Zeigt ein Fenster, welches jede im Programm benutzte globale Variable anzeigt. Der Inhalt der Variablen wird in Echtzeit aktualisiert, um einfach dem Programmablauf folgen zu können.
<i>Show Debug Window</i> („Zeige Debugger-Fenster“)	Zeigt ein Fenster, welches die Ausgabe des 'Debug' Befehls darstellt.

Der Debugger kann jedoch auch direkt im Programmcode aufgerufen werden. Dafür stehen verschiedene Befehle zur Verfügung, die Sie direkt in Ihrem Programmcode – z.B. zum Testen einer Funktion – verwenden können.

Die Debugger Schlüsselwörter in PureBasic:

<i>CallDebugger</i>	Ruft den "Debugger" auf und hält sofort die Programmausführung an.
<i>Debug <Ausdruck> [, DebugLevel]</i>	Zeigt das DebugOutput Fenster und darin das Ergebnis. Der Ausdruck kann jeder gültige PureBasic Ausdruck sein, von numerisch bis zu einem String. Ein wichtiger Punkt ist, dass der Debug Befehl und sein zugehöriger Ausdruck total ignoriert (nicht kompiliert) werden, wenn der Debugger deaktiviert ist. Dies bedeutet, dass dieser Befehl benutzt werden kann, um einfach den Programmablauf zu verfolgen ("tra-

cen"), ohne beim Erstellen des endgültigen Executable die Debug-Befehle auskommentieren zu müssen.

Der 'DebugLevel' ist die Prioritätsstufe der Debug-Mitteilung. Alle normalen Debug-Mitteilungen (ohne angegebenen DebugLevel) werden automatisch angezeigt. Wenn ein Level angegeben wurde, dann wird die Mitteilung nur angezeigt, wenn der aktuelle DebugLevel (definiert mit dem nachfolgenden 'DebugLevel' Befehl) gleich oder höher als dieser Wert ist. Dies ermöglicht einen hierarchischen Debug-Modus, indem mehr und mehr präzisere Informationen in Abhängigkeit vom benutzten DebugLevel angezeigt werden.

<i>DebugLevel</i>	Legt den aktuellen DebugLevel für die 'Debug' Mitteilung fest.
<i>DisableDebugger</i>	Dies schaltet die Debugger-Prüfroutinen bei nach diesem Befehl folgenden Sourcecode aus.
<i>EnableDebugger</i>	Dies schaltet die Debugger-Prüfroutinen bei nach diesem Befehl folgenden Sourcecode ein (wenn der Debugger vorher mittels DisableDebugger ausgeschaltet wurde).

An dieser Stelle wollen wir uns gemeinsam ein kleines Beispiel zur Verwendung der „Debugger Schlüsselwörter“ anschauen. Wir verwenden wieder das oben angeführte Beispiel „Window.pb“, fügen jedoch eine zusätzliche Programmzeile mit einer Ausgabe ins Debugger-Fenster ein.

Dazu fügen Sie bitte in unserem Beispiel innerhalb der „Repeat Until“ - Schleife die folgende Programmzeile ein: *Debug* EventID.1
Der betreffende Programmabschnitt müsste jetzt wie folgt aussehen:

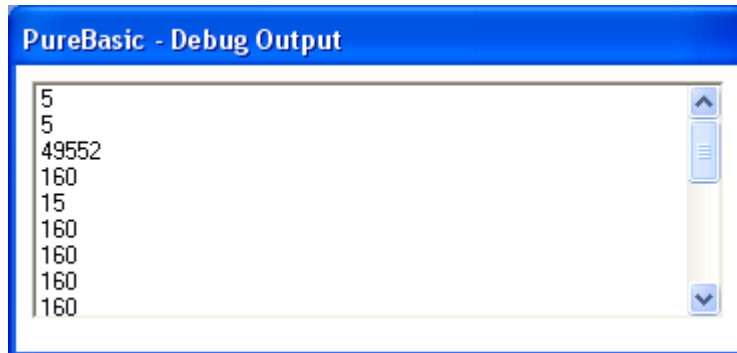
```

    EventID.1 = WaitWindowEvent ()
Debug EventID.1
    If EventID = #PB_Event_CloseWindow
    ...

```

Kompilieren und starten Sie jetzt bitte unser Beispielprogramm. Sobald

die „Repeat ... Until“ Schleife erreicht ist, wird das Debugger-Fenster geöffnet, welches in etwa wie folgt aussehen sollte:



Dieses Fenster gibt bei jedem Durchlauf der Schleife den aktuellen Wert der Variable ‚EventID.1‘ aus.

Was jedoch hier im einzelnen passiert, wollen wir uns in den folgenden Kapiteln dieses Handbuchs anschauen.

Durch die Funktionalität als Ausgabefenster für Variablen-Inhalte können Sie den ‚Debug‘ Befehl an jeder Stelle im Programmcode benutzen, wo Sie schnell und einfach den Inhalt einer Variablen, einer Zeichenkette oder das Ergebnis eines Ausdrucks ausgeben möchten, ohne selbst ein Fenster für die Ausgabe zu öffnen.

6. Grundlagen

6.1 Mein erstes Programm

Bevor wir in die Grundlagen von BASIC und im speziellen von PureBasic einsteigen, wollen wir uns an dieser Stelle das für viele Programmiersprachen so typische „Hallo Welt“ anschauen. Damit erhalten Sie einen anfänglichen kleinen Einblick in die einfachen, aber mächtigen Funktionen von PureBasic.

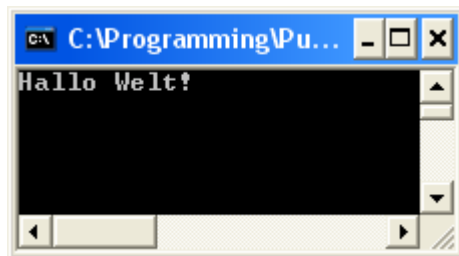
Die Ausgabe eines Programms kann entweder in der Windows-Konsole (auch „Eingabeaufforderung“ genannt, siehe Startmenü von Windows unter Programme/Zubehör) erfolgen oder auf einer graphischen Oberfläche (GUI).

Beide Möglichkeiten wollen wir uns nachfolgend anschauen:

Ausgabe über Konsole

```
OpenConsole()  
PrintN("Hallo Welt!")  
dummy$=Input()  
CloseConsole()
```

Ergebnis



Wie Sie sehen, genügen vier Programmzeilen, um diese Ausgabe zu erzeugen. Eigentlich sind dies ja sogar nur zwei Zeilen: die erste Zeile mit dem Befehl „OpenConsole()“ öffnet das Konsolenfenster und die zweite Zeile gibt mittels dem Befehl „PrintN“ den Text „Hallo Welt!“ aus. Die dritte und vierte Zeile ist lediglich dazu da, auf einen Tastendruck des Anwenders zu warten und das Konsolenfenster wieder zu schließen.

Ausgabe über GUI

```
MessageRequester("Info","Hallo Welt!",0)
```

Ergebnis



Für diese Ausgabe genügt lediglich eine Programmzeile. Wow! Benutzt wird hierzu die Funktion „MessageRequester“ aus der Bibliothek „Requesters“.

6.2 BASIC Allgemein

Die Geschichte von BASIC

BASIC – diese Abkürzung steht für „Beginners All Purpose Symbolic Instruction Code“. Dies steht sinngemäß für „Symbolischer Befehlscode für Anfänger und alle Zwecke“. BASIC wurde ursprünglich in den 60er Jahren von John Kemeny und Thomas Kurtz entwickelt, um eine selbst für Kinder leicht erlernbare Computersprache zu erschaffen. Und tatsächlich schaffte es BASIC, Tausende Kinder und Erwachsene zum Lernen über den Umgang mit Computern und das Programmieren anzuregen. Es war ganz auf die vielen Anfänger zugeschnitten, welche es als einfache Programmiersprache für alle möglichen Zwecke einsetzen wollten.

In den 70er Jahren – als Desktop-Computer langsam für die breite Masse zugänglich wurden – erfolgte eine Weiterentwicklung von BASIC auf einen höheren Level. Firmen wie Apple, Tandy und Atari bauten Computer für Heimanwender und für Schulen. Alle diese Computer konnten BASIC verwenden. Die Kombination von Hardware und Software machte BASIC zur Programmiersprache der Wahl. Als mehr Studenten BASIC erlernten, begann in den späten 70er und frühen 80er Jahren eine beschleunigte Entwicklungsphase.

Während das originale BASIC nach und nach in den Hintergrund gedrängt wurde, entstanden im Laufe der Jahre eine ganze Reihe von verschiedenen BASIC-Programmiersprachen. Alle Versionen hatten viele Gemeinsamkeiten in ihrem BASIC-Ursprung, benutzten jedoch auch eigene erweiterte Befehle.

Einer der Hauptgründe, dass BASIC so populär wurde, bestand darin, dass es lange Zeit eine reine Interpreter-Programmiersprache war. Interpretiert bedeutet, dass Sie Ihren Programmcode schreiben und unmittelbar danach das Programm starten können. Nachteil dieser Funktionsweise ist, dass die geschriebenen Programme zum Starten immer auf den Interpreter angewiesen sind. Typische Vertreter für heute noch verwendete Interpreter-Sprachen sind HTML und Javascript, die beide im Web-Browser Verwendung finden.

Modernere Sprachen sind kompiliert. Der Programmcode muss hier vor dem Starten erst durch den Compiler verarbeitet und in ein ausführbares Programm übersetzt werden. Selbst kleine Änderungen am Programmcode erfordern ein vollständiges Neukompilieren. Vorteil der Funktionsweise ist, dass kompilierte Programme im Gegensatz zu interpretierten Sprachen allein lauffähig sind und oftmals auch eine höhere Geschwindigkeit aufweisen. Heute weitverbreitete Compiler-Programmiersprachen sind C/C++ und Java.

Auch im Bereich der BASIC-Programmiersprachen ging der Trend spätestens in den 90er Jahren zu den Compiler-Sprachen. Die heute am weitesten verbreitete BASIC-Programmsprache ist Visual Basic (VB) von Microsoft®.

Auch PureBasic gehört zur Gattung der Compiler-Programmiersprachen. Im Gegensatz zum Branchenführer soll es jedoch durch ausgesprochen kleine und kompakte sowie vor allem sehr schnelle erzeugte Programme beeindrucken.

Für alle, die noch mehr an den „Internat“ von PureBasic interessiert sind: PureBasic erstellt nicht selbst die Maschinencode-Befehle, sondern wandelt als eine Art „Übersetzer“ den BASIC-Code des Programmierers in optimierten Assemblercode um. Dieser Assembler-Code wird anschließend durch den Assembler „NASM“ in Maschinencode und damit das ausführbare Programm kompiliert.

Grundlegendes über die Programmierung und BASIC

Ein Programm wird meistens deshalb entworfen, um ein Problem oder eine Aufgabe durch das Be-/Verarbeiten von Daten bzw. Informationen zu lösen. Auf den Programmierer kommen dabei folgende Aufgaben zu:

- Eingabe – Eingabe von Daten/Informationen in das Programm
- Speichern – Speichern der erhaltenen Daten/Informationen
- Verarbeitung – Verarbeiten der gespeicherten Daten/Informationen
- Ausgabe – Daten/Informationen an den Benutzer ausgeben

Die Eingabe oder Erfassung von Daten/Informationen geschieht durch das Einlesen von Werten von der Tastatur, aus einer Datei auf Diskette, CD-ROM oder anderem Medium sowie durch weitere Ein- und Ausgabeschchnittstellen (Maus, USB-/Parallelanschluss, Modem/LAN u.v.a.).

Das Speichern von Daten erfolgt durch systematisch bereitgestellten Speicherplatz im Hauptspeicher des Computers, der verschiedene Datentypen (Konstanten, Variablen, Zeichenketten und Strukturen) sowie Adressen (von Variablen und Strukturen) beinhaltet.

Die Verarbeitung von Daten/Informationen erfolgt mittels Befehlen, Funktionen und Operationen (Zuweisen, Kombinieren und Vergleichen von Werten). Diese können so organisiert werden, dass sie nur unter bestimmten Bedingungen ausgeführt werden (bedingte Ausführung), mehrfach ausgeführt werden (Schleifen) oder auch in einzelne Einheiten aufgeteilt werden (Unterprogramme).

Hierbei handelt es sich um die Grundelemente jeder Programmiersprache. Diese sollten Sie beim Entwerfen Ihrer eigenen Programme immer im Auge behalten.

Nähere Einzelheiten zu diesen Elementen in PureBasic erfahren Sie in den nachfolgenden Kapiteln.

6.3 PureBasic Syntax

PureBasic verwendet einige grundlegende Syntax-Regeln, welche sich niemals ändern werden. Dies sind:

- Kommentare werden gekennzeichnet mittels „;“ (Semikolon). Alle nach dem „;“ eingegebenen Texte werden vom Compiler ignoriert. Kommentare sind nützlich, um insbesondere in größeren Programmen das spätere Zurechtfinden zu vereinfachen, indem die Funktion von Befehlen oder Prozeduren, der Inhalt von Variablen etc. kurz erläutert werden. Es können beliebig viele Kommentare in einem Sourcecode eingegeben werden.

Beispiel

```
If a = 10      ; Dies ist ein Kommentar zur  
              ; näheren Erläuterung.
```

- Allen Funktionen müssen „(“ und „)“ (öffnende und schließende Klammern) folgen, oder sie werden nicht als Funktion erkannt, das gilt auch für parameterlose Funktionen.

Beispiel

```
WindowID() ist eine Funktion.  
WindowID ist eine Variable.
```

- Alle Konstanten werden eingeleitet von einem „#“

Beispiel

```
#Hello = 10 ist eine Konstante.
```

Hello = 10 ist eine Variable.

- Allen Sprungmarken (Labels) muss ein **,:** (Doppelpunkt) folgen.

Beispiel

Ich_bin_eine_Sprungmarke:

- Ein Ausdruck ist etwas, was berechnet werden kann. Ein Ausdruck kann beliebige Variablen, Konstanten oder Funktionen desselben Typs beinhalten. Wenn Sie Zahlen innerhalb eines Ausdrucks benutzen, können Sie das **,\$'** Zeichen vor der Zahl einfügen, um die Verwendung einer Hexadezimal-Zahl anzuzeigen. Oder ein **,%'** Zeichen, um eine Binär-Zahl anzuzeigen. Ohne eines der beiden Zeichen wird die Zahl als Dezimal-Zahl behandelt. Strings (Zeichenketten) müssen mit Anführungszeichen eingeschlossen werden.

Beispiele gültiger Ausdrücke

```
a+1+(12*3) a+WindowHeight()+b/2+#MeineKonstante
a <> 12+2 b+2 >= c+3
a.s=b.s+"dies ist ein String Wert"+c.s
a + $69 / %1001 ; Nutzung hexadezimaler und binärer
                  ; Zahlen
```

- Eine beliebige Anzahl an Befehlen kann auf derselben Zeile mittels der **,:** Option aneinandergereiht werden.

Beispiel

```
If OpenScreen(0,320,200,8,0) : PrintN("Ok") : ␣
Else : PrintN("Fehler") : EndIf
```

- Begriffe, die in dieser Anleitung benutzt werden :
 - **<Variable>** : eine Basic-Variable, siehe Kapitel 7.
 - <Ausdruck>** : ein Ausdruck, wie oben beschrieben.
 - <Konstante>** : eine numerische Konstante.
 - <Sprungmarke>** : eine Adresse für ein Unterprogramm oder einen DATA-Block.
 - <Typ>** : jeder Typ (Standard oder in einer Struktur), siehe Kapitel 7.

Was Sie in PureBasic nicht finden werden, das sind Zeilennummern. Im Gegensatz zu manch älteren BASIC-Dialekten verzichtet PureBasic gänzlich darauf, da das „Anspringen“ einer bestimmten Stelle im Programmcode ganz einfach über die Sprungmarken und den entsprechenden Befehlen wie Gosub, Goto etc. möglich ist.

6.4 End

Das Schlüsselwort *End* beendet ein laufendes PureBasic - Programm vollständig und schließt dabei alle geöffneten Fenster und Bildschirme,

gibt reservierten Speicher frei usw. Wenn Sie Ihr Programm aus dem Editor heraus gestartet haben, befinden Sie sich nach dem *End* wieder im Editor.

Sie sollten es immer am Ende Ihres Programms einsetzen, um ein sauberes Beenden Ihres Spiels oder Ihrer Applikation sicherzustellen.

6.5 Glossar

Erklärung wichtiger (größtenteils englischsprachiger) Begriffe, die häufig in diesem Handbuch bzw. bei der PureBasic - Programmierung vorkommen:

API	Application Programming Interface - sinngemäß in Deutsch: Programmierschnittstelle (für Anwendungen). Siehe auch „Windows API“.
Array	Eine Gruppe („Ansammlung“) von Variablen gleichen Typs, die durch Definition eines Felds (Array) einfacher verwaltet werden können.
ASCII	Steht für „American Standard Code for Information Interchange“, einem im ASCII-Code genormten Austauschformat für Ziffern und Zeichen des lateinischen Alphabets. Damit wird der problemlose Datenaustausch zwischen verschiedenen Programmen und Computern erst möglich.
Ausdruck	Mathematisches Gebilde aus Variablen, Operatoren, Funktionen und Klammern, das berechnet werden kann. Kommt oftmals bei der Berechnung und Zuweisung eines Variableninhalts sowie als Bedingung bei der Ablaufsteuerung zum Einsatz.
Bit	Kleinste Informations- und Speichereinheit in der Welt der Computer. Der Begriff setzt sich aus „Binary“ (binär) und „Digit“ (Ziffer) zusammen. Ein Bit enthält entweder den Wert 0 oder 1. Aus 8 Bit wird ein Byte (siehe Kapitel 7) gebildet.

Bug	Wird im englischen für „Ungeziefer“ verwendet und bezeichnet in der Programmierung einen Programmfehler. Achten Sie also darauf, dass Ihre Programme möglichst wenig solcher „Bugs“ (Fehler) enthalten...
Compiler	„Übersetzt“ den vorhandenen Sourcecode in eine für den Computer verständliche Sprache und erstellt daraus ein ausführbares Programm (Executable).
Dynamic-Link Libraries (DLL)	DLL's sind ausführbare Module, welche Funktionen und Daten enthalten. Sie bieten dem Anwender eine Möglichkeit, seine Applikationen zu modularisieren, um die Module je nach Bedarf einzeln zu laden, zu aktualisieren oder mehrfach zu benutzen. DLL's sind entweder im MS Windows® System enthalten oder werden von einer Applikation mitgeliefert. Für den Endanwender einer Applikation sind DLL weniger von Bedeutung.
Executable	Ausführbares Programm, welches durch einen Compiler aus dem Sourcecode erstellt wird.
GUI	Graphical User Interface. So wird die „graphische Benutzeroberfläche“ eines Programms bezeichnet.
Handle	Eine Variable, welche ein Objekt identifiziert. Sie stellt eine indirekte Referenz auf eine OS-Ressource dar. Oftmals in Verbindung mit WinAPI-Funktionen benötigt.
Library	Gebräuchlicher englischer Begriff für „Befehlsbibliothek“ (Mehrzahl: Libraries), die Befehle und Funktionen zur Nutzung in PureBasic enthalten.
Operating System (OS)	Sys- Als OS wird das Betriebssystem bezeichnet, welches dem Anwender alle benötigten grundlegenden Funktionen und die Oberfläche zum Benutzen eigener Programme zur Verfügung stellt. Weitverbreitetes OS ist sicherlich Windows® von Microsoft®.
Parameter	Als Parameter bezeichnet man die an eine Funktion, einen Befehl, ein Schlüsselwort oder eine Prozedur übergebenen Werte (Variablen). Ist ein Parameter „optional“, d.h. kann aber muss nicht angegeben werden, wird dieser Parameter in diesem Handbuch in eckigen [Klammern] dargestellt.

Pointer	Pointer ist der englischsprachige Begriff für „Zeiger“ und ist eine Variable, die eine Speicheradresse enthält, an der Daten gespeichert sind.
Schlüsselwörter (englisch: Key- words)	Sind die PureBasic eigenen „Befehle“, mit denen Prozeduren, Strukturen u.v.a. definiert werden, also z.B.: Procedure, EndProcedure, Structure, EndStructure usw. Die Schlüsselwörter sind immer in PureBasic verfügbar, dafür muss nicht extra eine Befehlsbibliothek o.ä. geladen sein.
Sourcecode (auch nur: Code)	Als Sourcecode wird der vom Programmierer eingegebene Programmcode (in unserem Fall also PureBasic - Programmcode) bezeichnet, aus dem mittels dem Compiler ein ausführbares Programm verwandelt werden kann.
Syntax	Definiert die Regeln für zur richtigen Anwendung einer Funktion, eines Befehls, eines Operators usw.
Thread	Ein Thread ist die grundlegende Einheit, für welche das OS Rechenzeit reserviert. Eine gerade aktive (ausgeführte) Applikation besteht aus einem oder mehreren Threads und wird auch als Prozess bezeichnet.
Windows API	Das Microsoft® Windows® Betriebssystem ermöglicht dem Anwender den Zugriff auf viele integrierte Funktionen, damit dieser seine Programme einfach mit weiteren Möglichkeiten aufwerten kann. Der Zugriff auf das Betriebssystem erfolgt dabei über eine definierte Software-Schnittstelle, genannt: Application Programming Interface (API).
Zeiger	Siehe unter „Pointer“.

7. Datentypen

Erinnern wir uns: Das Speichern von Daten und Informationen innerhalb eines Programms erfolgt im Speicher des Computers. Dieser Speicherplatz wird in Form von Datentypen bereitgestellt. Die verschiedenen Datentypen, die in PureBasic zur Verfügung stehen, wollen wir uns nachfolgend näher ansehen.

7.1 Einführung zu Variablen

Eine Variable ist eine Art „beschrifteter Speicherbereich“. Vom Programmierer wird der Speicherbereich beschriftet, indem er einen Namen für die Variable vergibt. Eine typische Eigenschaft von Variablen ist, dass sich ihre Werte (also der Inhalt) ständig ändern können. Natürlich muss auch die Art der Variable festgelegt werden.

7.2 Variablentypen

Numerische Datentypen

Numerische Variablen in PureBasic sind vorzeichenbehaftet (englisch: „signed“). Es gibt Ganzzahlen (Byte, Word, Long) und einfach genaue Fließkommazahlen (Float). Diese haben folgende Merkmale:

<i>Name</i>	<i>Erweiterung</i>	<i>Speicher- verbrauch</i>	<i>Werte-Bereich</i>	<i>Beispiel</i>
Byte	.b	1 Byte im Speicher	-128 bis +127	Variable.b = 5
Word	.w	2 Byte im Speicher	-32768 bis +32767	Wert.w = -25000
Long	.l	4 Byte im Speicher	-2147483648 bis +2147483647	Zahl.l = 1521800
Float	.f	4 Byte im Speicher	unlimitiert	Nummer.f = 4.567

Weiterhin ist die Verwendung von Hexadezimal- und Binärzahlen möglich:

- Hexadezimalzahlen werden durch ein vorangestelltes Dollarzeichen gekennzeichnet, z.B. \$69
- Binärzahlen werden durch ein vorangestelltes Prozentzeichen gekennzeichnet, z.B. %1001

Alphanumerische Datentypen

Alphanumerische Datentypen in PureBasic sind besser bekannt als „Strings“. Strings sind im Speicher abgelegte Zeichenketten. Diese haben folgende Merkmale:

Name	Erweiterung	Speicherverbrauch	Werte-Bereich	Beispiel
String	.s	Länge des Strings + 1	unlimitiert (max. 64 KByte)	String.s = „Hallo“

Alternativ zu der Erweiterung ‚s‘ kann auch das Dollarzeichen am Ende des Variablennamens benutzt werden. Das String-Beispiel würde dann so aussehen: String\$ = „Hallo“

Der Inhalt eines Strings muss immer innerhalb von Anführungszeichen übergeben werden, wie oben z.B. „Hallo“.

Boolesche Variablen

Aus anderen Programmiersprachen (z.B. Pascal) kennen Sie vielleicht den Datentyp „Boolean“. Dieser Datentyp kann nur zwei mögliche Werte annehmen: TRUE (wahr) oder FALSE (falsch).

„False“ entspricht stets einem Wert von 0, „True“ jedem anderen Wert (meist wird 1 oder -1 verwendet).

Beide Werte kommen als Ergebnis eines Ausdrucks zum Einsatz, z.B. innerhalb einer Bedingung oder einer Schleife (siehe Kapitel 9).

Falls Sie in Ihrem Programmcode direkt auf diese Begriffe zurückgreifen möchten, können Sie diese einfach als Konstanten definieren: #True=1 (bzw. #True=-1) und #False=0.

7.3 Deklaration und Default-Datentyp

Deklaration von Variablen

Um eine Variable in PureBasic zu definieren, geben Sie einfach ihren Namen ein oder den Typ den die Variable annehmen soll. Das erste Zeichen des Variablennamens muss dabei immer ein Buchstabe oder ein Unterstrich (_) sein. Danach können auch Zahlen, jedoch keine Sonderzeichen folgen. Der Variablenname ist insgesamt auf maximal 100 Zeichen beschränkt. Was aber mehr als ausreichen sollte... ☺

Variablen müssen nicht ausdrücklich am Programmanfang deklariert werden, sie können auch als Variablen "on-the-fly" (automatische Deklaration durch Verwendung innerhalb eines Ausdrucks) benutzt werden.

Beispiel

```
a.b          ; Deklariert eine Variable genannt  
              ; 'a' vom Typ Byte (.b).  
c.l = a*d.w   ; 'd' wird hier mitten im Ausdruck  
              ; deklariert!
```

Bitte beachten Sie, dass Variablen mit gleichem Namen nicht mehrfach deklariert werden können. PureBasic unterscheidet dabei nicht zwischen Groß- und Kleinschreibung, d.h. 'var.w' und 'VaR.w' werden als gleiche Variable behandelt.

Folgender Programmcode erzeugt z.B. einen Fehler:

```
a.b = 1       ; Deklariert eine Variable  
              ; 'a' vom Typ Byte (.b).  
a.w = 2       ; Nochmalige Deklaration von  
              ; 'a' als Word (.w) schlägt fehl, da 'a'  
              ; bereits deklariert wurde.
```

Variablen können von Ihnen auch ohne Angabe eines Typs deklariert werden. Dann wird ein vorher festgelegter Standard- oder Default-Datentyp benutzt.

Der Default-Datentyp

Das *DefType* Schlüsselwort kann benutzt werden, um eine ganze Reihe von Variablen mit einheitlichem Typ zu definieren. In diesem Fall deklariert *DefType* diese Variablen als "definierte Typen" und ändert nicht den Standardtyp.

Syntax

```
DefType.<Typ> [<Variable>, <Variable>, ...]
```

Beispiel

```
DefType.b a, b, c, d   ; a, b, c, d werden als  
                      ; Byte-Typen (.b) deklariert.
```

Wenn keine <Variablen> angegeben werden, wird *DefType* benutzt, um den Standardtyp ("Default-Typ") für zukünftige typenlose Variablen festzulegen.

Syntax

```
DefType.<Typ>
```

Beispiel

```
DefType.l           ; Deklaration von 'Long' (.l)  
                    ; als Default-Datentyp  
a = b+c             ; a, b und c werden als Long-Typen (.l)  
                    ; definiert, da kein  
                    ; anderer Typ angegeben wurde.
```

7.4 Konstanten

Konstanten besitzen im Gegensatz zu Variablen einen festen Wert, der ihnen einmalig (bevorzugt zu Beginn des Sourcecodes) zugewiesen

wird. Sie kommen oft dann zum Einsatz, wenn in einem größeren Programmcode anstelle wenig aussagekräftiger Zahlenwerte gut zu merkende Namen verwendet werden sollen. Beim Kompilieren des Executable werden die eingesetzten Konstanten durch die zugehörigen Werte ersetzt.

Eine Konstante kennzeichnen Sie in PureBasic, in dem Sie ihrer Bezeichnung ein '#' voranstellen. Als Werte sind Ganzzahlen (einschließlich Hexadezimal- und Binärzahlen) und Strings möglich, z.B. #Wert = 1 oder #String = "Hallo".

Neben der Deklaration eigener Konstanten stehen in PureBasic auch vordefinierte Konstanten zur Verfügung. Diese werden in den Dateien „PureBasic_x86.res“ (PureBasic – spezifische Konstanten) und „Windows.res“ (Windows – Systemkonstanten) im Verzeichnis „...\PureBasic\Residents“ mitgeliefert. Zur Anwendung kommen diese Konstanten insbesondere im Zusammenhang mit bestimmten Befehlen, z.B. beim Erstellen von Fenstern (Windows-Library), Tastatur-Abfragen (Keyboard-Library) usw. In der Befehlsreferenz wird darauf an den entsprechenden Stellen näher eingegangen.

7.5 DATA – Anweisung

Einführung

Mit der DATA - Anweisung ist es Ihnen möglich, ganze Blöcke oder Listen mit nützlichen Informationen (Variablen und Strings) im PureBasic - Programmcode zu integrieren. Dies ist sehr nützlich, um z.B. Sprachen-Strings (für die Lokalisierung Ihres Programms in verschiedenen Sprachen) oder Variablen mit den Koordinaten eines (vorberechneten) Sprite – Weges zu definieren.

Definition eines DATA - Blockes

Um einen der vorgestellten DATA - Blöcke zu definieren, benutzen Sie folgende Befehle:

<i>DataSection</i>	Muss zuerst aufgerufen werden, um einen nachfolgenden DATA-Block einzuleiten.
<i>Data.Type</i>	Wird benutzt, um nachfolgend Daten (Strings und Variablen) einzugeben. Der Typ muss ein einfacher Variablen-Typ (numerisch: Long, Word, Byte, Float;

alphanumerisch: String) sein. Auf jeder ‚Data‘-Zeile kann eine beliebige Anzahl Daten des angegebenen Typs eingegeben werden, jeweils abgegrenzt durch ein Komma ‚,‘.

<i>EndDataSection</i>	Schließt einen vorher mit ‚DataSection‘ eingeleiteten DATA-Block ab. Muss angegeben werden, wenn nach dem DATA-Block weiterer Programmcode folgt.
-----------------------	---

Beispiel für einen gültigen DATA-Block

```
DataSection
Data.l 100, 300, -25, -182, 4395
Data.s "PureBasic", "Feel", "the", "pure", "Power!"
Data.f 1.5, 89.99, -4.5, 3.75, 236.4
EndDataSection
```

Anmerkung

Sie können beliebig viele DATA - Blöcke in Ihrem Programmcode definieren. Alle Sprungmarken und Daten innerhalb der DATA - Blöcke werden in der Daten-Sektion des fertig kompilierten Programms gespeichert, die einen wesentlich schnelleren Speicherzugriff als die Code-Sektion bietet.

Auslesen eines DATA - Blockes

Natürlich wollen wir die eingegebenen Daten auch innerhalb des Programms verwenden können. Dazu müssen wir sie aus dem DATA-Block „auslesen“, wofür die folgenden beiden Befehle nützlich sind:

<i>Restore</i> Sprungmarke	Dieses Schlüsselwort wird verwendet, um die Startposition für ein nachfolgendes <i>Read</i> auf das erste Element eine angegebene Sprungmarke zu setzen.
<i>Read</i>	Liest die nächsten verfügbaren Daten ein. Die nächsten verfügbaren Daten können mittels dem Restore Befehl geändert werden. Standardmäßig sind die nächsten verfügbaren Daten die zuerst deklarierten Daten.

Beispiel

```
Restore Strings
Read ErsterString$
Read ZweiterString$
...
...
Restore Werte
Read a
Read b
...
DataSection
Werte:
    Data.l 100, 200, -250, -452, 145
Strings:
    Data.s "Hallo", "lieber", "PureBasic", "Anwender"
; EndDataSection ist hier nicht unbedingt erforderlich,
; da sich der DATA-Block am
; Ende des Programmcodes befindet
```

7.6 Arrays

Einführung

Programme müssen häufig eine ganze Reihe von Variablen verarbeiten, die oftmals in einem gewissen Zusammenhang zueinander stehen. In diesem Fall bietet es sich an, nicht unzählige einzelne Variablen zu deklarieren, sondern diese in einer Gruppe zusammenzufassen. Eine solche Gruppe wird als Array bezeichnet.

Beispiel

Sie besitzen eine Autogrammkartensammlung mit 100 Exemplaren und möchten diese in Ihrem PureBasic-Programm erfassen und weiterverarbeiten. Sie könnten nun 100 Variablen deklarieren, etwa so:

```
Autogrammkarte1.s = "Steffi Graf"
Autogrammkarte2.s = "Boris Becker"
Autogrammkarte3.s = "Michael Schumacher"
usw.
```

Umständlich, oder? Hier kommen nun die Arrays ins Spiel. Wäre es nicht viel einfacher, wenn Sie einfach

```
Autogrammkarte(1) = "Steffi Graf"
Autogrammkarte(2) = "Boris Becker"
Autogrammkarte(3) = "Michael Schumacher"
```

schreiben könnten? „Sicher“ werden Sie sagen, deshalb werden uns nachfolgend der Verwendung von Arrays in Ihren Programm widmen.

Definition von Arrays

Bevor Arrays benutzt werden können, muss für sie Speicherplatz bereitgestellt werden. Diesen Vorgang nennt man auch „Dimensionieren“. Dazu dient der Befehl *Dim*.

Dim Name.Typ(Ausdruck, [Ausdruck], ...)

Erklärungen zu den Parametern dieses Befehls:

Name	Name des Arrays, der innerhalb des Programms verwendet werden soll.
Typ	Typ des Arrays, z.B. ‚s‘ für ein Array von Strings, ‚l‘ für ein Array von Long-Variablen usw. Ein Array in PureBasic kann von beliebigem Typ sein, einschließlich strukturierter und benutzerdefinierter Typen.
Ausdruck	Anzahl der Datenfelder, die das Array beinhalten soll. Hier kann eine Variable (Variablenname oder direkt ein numerischer Wert) oder ein Ausdruck (Berechnung, z.B. ‚Anzahl.w * 5‘) angegeben werden. Hinweis: Das erste Element in Ihrem Array beginnt bei 0. Wenn Sie also genau 100 Elemente in Ihrem Array verwenden möchten, müssen als Ausdruck ‚99‘ übergeben (Elemente 0 – 99).
[Ausdruck], ...	Ein oder mehrere optionale Ausdrücke, welche zur Definition von mehrdimensionalen Arrays, sogenannten „Multiarrays“ dienen. Soll z.B. ein Array mit drei Dimensionen definiert werden, sind innerhalb der Klammern nach ‚Dim Name.Typ‘ auch drei Ausdrücke anzugeben.

Beispiel 1

```
Dim Zahlen.l(40) ; dimensioniert ein Array für
                  ; Long-Variablen mit Platz
                  ; für 41 Elemente (0-40)
Zahlen(0) = 1    ; weist dem Element 0 innerhalb
                  ; des Arrays den Wert 1 zu
Zahlen(1) = 2    ; weist dem Element 1 innerhalb
                  ; des Arrays den Wert 2 zu
```

Beispiel 2

```
Zeilen.w=50
Spalten.w=50
; dimensioniert ein zweidimensionales Array für
; Byte-Variablen mit je 51 Zeilen und Spalten (0-50)
Dim MultiArray.b(Zeilen, Spalten)

; weist dem Element in Zeile 10 / Spalte 20 den Wert 10 zu
MultiArray(10,20) = 10

; weist dem Element in Zeile 20 / Spalte 30 den Wert 20 zu
MultiArray(20,30) = 20
```

Hinweise

Wenn ein Array einmal *Dimensioniert* wurde, kann es später im Programm erneut dimensioniert werden, aber sein Inhalt wird dabei gelöscht. Dies ist nützlich, um den vorher belegten Speicherplatz freizugeben oder das Array mit neuen Werten zu füllen. Arrays werden dynamisch reserviert (angelegt), d.h. es kann einfach eine Variable oder ein Ausdruck benutzt werden kann, um sie erneut zu *Dimensionieren*.

Zugriff auf den Inhalt eines Arrays

Der Zugriff auf ein Element innerhalb des Arrays geht ebenfalls ganz einfach. Auf entgegengesetztem Weg, wie wir gerade dem mehrdimensionalen Array einen Wert zugewiesen haben, ermitteln wir den Inhalt des entsprechenden Elements:

```
a.b = MultiArray(10,20) ; weist der Byte-Variablen a
                        ; den Wert 10 zu
                        ; (Inhalt des Elements in
                        ; Zeile 10 / Spalte 20)
```

7.7 Listen

Einführung

Die Listen bieten ebenso wie die Arrays die Möglichkeit, eine ganze Reihe von Variablen in einer Gruppe zusammenzufassen. Die Listen unterscheiden sich jedoch von den Arrays darin, dass sie beim Einrichten nicht die Anzahl der später benötigten Elemente kennen müssen, da der benötigte Speicherplatz dynamisch reserviert wird. Darum werden die Listen auch dynamisch verknüpfte Listen (englisch: „dynamic linked lists“) genannt.

Es gibt keine Einschränkungen in der Anzahl der Elemente, es können

so viele wie nötig verwendet werden. Der Nachteil besteht jedoch darin, dass bei den Listen nur eine Dimension möglich ist.

Definition einer Liste

Bevor eine Liste benutzt werden kann, muss sie eingerichtet werden. Der dafür erforderliche Befehl lautet *NewList*. Für das Hinzufügen von Elementen benötigter Speicherplatz wird automatisch reserviert.

NewList Name.Typ()

Parameter

Name	Name der Liste, der innerhalb des Programms verwendet werden soll.
Typ	Typ der Liste, z.B. ‚s‘ für eine Liste von Strings, ‚l‘ für eine Liste von Long-Variablen usw. Eine Liste in PureBasic kann jeden Standard- (numerische und alphanumerisch) oder Struktur-Typ haben.

Beispiel

```
NewList Liste.l()      ; Richtet eine neue Liste für
                        ; Long-Variablen ein
AddElement(Liste())    ; Fügt dieser Liste ein neues
                        ; Element hinzu
Liste() = 10           ; Weist dem neuen Element den
                        ; Wert 10 zu
```

Hinweis

Nach dem Namen einer Liste (z.B. Liste.l) stehen immer zwei Klammern (). Diese haben im Gegensatz zu Arrays keinen Wert oder Ausdruck als Inhalt.

Zugriff auf den Listeninhalt

Während bei Arrays auf ein Element einfach zugegriffen werden kann, indem nach dem Arraynamen in Klammern die Position des Elements angegeben wird (z.B. a = Zahlen(10)), bezieht sich der Zugriff auf ein Listenelement immer auf das aktuelle Element.

Für das Verwalten des Listeninhalts gibt es eine gesonderte PureBasic – Befehlsbibliothek. Schauen Sie deshalb zu diesem Thema in die Befehlsreferenz zur Library „Linked List“.

7.8 Strukturen

Einführung

Strukturen sind die Zusammenfassung einer oder mehrerer Variablen zu einer Einheit, die sich dann durch ihren Namen ansprechen lässt. Strukturen werden insbesondere verwendet, um größere Datenbestände durch Gruppieren aller zusammengehörenden Informationen (z.B. Variablen, Strings) zu einer Einheit besser und schneller verwalten zu können. Mit der Benutzung von Strukturen in PureBasic ist es somit möglich, eigene „Benutzer-Typen“ (die aus mehreren Standard-Datentypen bestehen) zu definieren, aber auch Zugriff auf einige Speicherbereiche im OS zu erhalten.

Aus anderen Programmiersprachen sind Strukturen auch unter NewTypes (BlitzBasic) oder Records (Pascal) bekannt. In der Programmiersprache „C“ heißen sie ebenfalls Strukturen und werden mit „struct“ eingeleitet.

Definition einer Struktur

Eine Struktur beschreibt den Aufbau einer Datengruppe, reserviert selbst aber noch keinen Speicher. Zum Definieren einer Struktur benötigen Sie folgende Befehle:

<i>Structure Name</i>	Mit dem Schlüsselwort ‚Structure‘ wird die nachfolgende Strukturdefinition eingeleitet. ‚Name‘ ist der Name für die zu definierende Struktur. Im Anschluss an dieses Schlüsselwort werden die einzelnen Deklarationen von Variablen, Strings etc. vorgenommen.
<i>EndStructure</i>	Damit wird die vorher definierte Struktur abgeschlossen.

Beispiel

Folgendes Beispiel fasst die wichtigen Daten zu einer Person in einer Struktur zusammen. Der Compiler merkt sich unter dem Namen „Person“ den Aufbau des durch den Anwender neu definierten Datentyps.

```
Structure Person
  Name.s      ; deklariert einen String „Name“
  Vorname.s   ; deklariert einen String „Vorname“
  Alter.w     ; deklariert eine Word-Variable „Alter“
  Strasse.s   ; deklariert einen String „Strasse“
  Wohnort.s   ; deklariert einen String „Wohnort“
  Plz.l       ; deklariert eine Long-Variable „Plz“
EndStructure
```

Innerhalb von Strukturen werden auch statische Arrays (siehe auch Kapitel 7.6) unterstützt.

```
Structure Anwender
  Name.s[10]   ; 10 Strings „Name“ verfügbar
EndStructure
```

Benutzen einer Struktur

Bei Daten, die in einer Struktur zusammengefasst sind, handelt es sich oftmals um ständig wiederkehrende Daten, die jeweils mehrfach vorhanden sind. Bei unserem obigen Beispiel „Person“ könnte dies z.B. eine Adressdatenbank sein, die mehrere Datensätze von „Person“ enthält.

Um diese Struktur mit Inhalten füllen zu können, müssen wir daher zuerst ein Array oder eine Liste deklarieren, die die benutzerdefinierte Struktur verwendet. Der Zugriff auf die einzelnen Datenfelder innerhalb der Struktur erfolgt mittels der ‚\‘ Option, die den Name des Arrays bzw. der Liste von den Datenfeldern trennt.

Es ist jedoch auch möglich, lediglich eine einzelne Variable vom Typ der bereits definierten Struktur zu deklarieren.

Beispiel

Definition von drei Benutzertypen mit Hilfe einer bereits definierten Struktur:

```
; Deklaration der Variablen „Thomas“, „Michael“
; und „Sylvia“ unter Benutzung
; unseres bereits definierten Struktur-Typs „Person“
DefType.Person Thomas, Michael, Sylvia
Thomas\Name = "Maier"      ; weist dem Element „Name“ von
                           ; „Thomas“ den String „Maier“ zu
Thomas\Vorname = "Thomas"  ; weist dem Element "Vorname" von
                           ; „Thomas“ den String "Thomas" zu
Michael\Alter = 25         ; weist dem Element "Alter" von
                           ; „Michael“ den Wert 25 zu
```

Da wir aber Bedarf an mehr als nur drei Adressen haben, legen wir ein Array an, das mind. 100 Adressen speichern soll:

```
; Deklaration eines Arrays „Adressen“ unter Benutzung un-
; serer bereits
; definierten Struktur „Person“
Dim Adressen.Person(100)
; Zuweisen der entsprechenden Informationen zur
; Position 0 des Arrays Adressen()
Adressen(0)\Name      = „Mustermann“
Adressen(0)\Vorname   = „Mustermann“
Adressen(0)\Alter     = 25
Adressen(0)\Strasse   = „Müllerstraße 99“
Adressen(0)\Wohnort   = „Musterstadt“
Adressen(0)\Plz       = 99999
```

Wollen Sie statt des Arrays eine dynamisch verknüpfte Liste verwenden, siehe eben aufgeführtes Beispiel so aus:

```
NewList Adressen.Person()      ; Einrichten einer Liste
                                ; „Adressen“ vom Typ „Person“
AddElement(Adressen())         ; Der Liste „Adressen“ ein
                                ; neues Element hinzufügen“
Adressen()\Name                = „Mustermann“      ; Zuweisen der
                                ; entsprechenden Informationen
Adressen()\Vorname             = „Mustermann“
Adressen()\Alter               = 25
Adressen()\Strasse             = „Müllerstraße 99“
Adressen()\Wohnort             = „Musterstadt“
Adressen()\Plz                 = 99999
```

Möchten Sie einfach ermitteln, wie viel Speicherplatz Ihre definierte Struktur benötigt? Dies können Sie ganz mit dem *SizeOf*(Struktur) Schlüsselwort erledigen.

Benutzen wir dafür einfach unsere bereits definierte Struktur 'Person':

```
a = SizeOf(Person)      ; gibt 22 (Byte) zurück, da die
                        ; Struktur für vier Strings und
                        ; eine Long-Variable jeweils 4 Byte und
                        ; für die Word-Variable
                        ; 2 Byte verbraucht.
```

Strukturen können auch verschachtelt werden. Dies erfolgt, indem eine bereits definierte Struktur innerhalb einer neu definierten Struktur aufgerufen und somit weiterverwendet wird.

```

Structure Mitarbeiter
  Daten.Person      ; deklariert eine neue Struktur
                    ; „Daten“ vom bereits
                    ; definierten Typ „Person“
  Personalnummer.w  ; deklariert eine Word-Variable
                    ; „Personalnummer“
  Gehaltsklasse.b   ; deklariert eine Byte-Variable
                    ; „Gehaltsklasse“
EndStructure

```

Für den Zugriff auf die Elemente der Struktur 'Person' sind jetzt allerdings zwei Backslashes notwendig. Der Zugriff auf den Inhalt der Struktur erfolgt dann z.B. wie folgt:

```

Arbeiter.Mitarbeiter\Daten\Name = „Müller“
; deklariert eine neue Variable
; 'Arbeiter' vom Typ ,Mitarbeiter' und
; weist dem Element einen Namen zu
Arbeiter\Daten\Vorname = „Ede“
; die Variable 'Arbeiter' ist bereits definiert,
; daher können wir direkt auf das Element
; 'Vorname' zugreifen und einen String übergeben

```

Ein weiteres Beispiel für eine komplexere Struktur:

```

Structure Window
  *NextWindow.Window ; verweist auf ein
                    ; anderes Window Objekt
  x.w
  y.w
  Name.s[10]         ; statisches Array,
                    ; in dem 10 Namen verfügbar sind
EndStructure

```

Für fortgeschrittene Programmierer gibt es auch noch eine weitere Möglichkeit zum Definieren einer Struktur. Mit Hilfe von *StructureUnion* und *EndStructureUnion* werden innerhalb einer Struktur einige Felder definiert, die sich einen Speicherplatz teilen. Sie funktionieren wie das 'Union' Schlüsselwort in C/C++ und werden im Deutschen auch als Varianten bezeichnet. Diese werden vorwiegend bei systemnaher Programmierung eingesetzt, um Speicherplatz zu sparen, indem verschiedene Daten, die niemals gemeinsam anfallen (daher auch Varianten), an der gleichen Speicheradresse abgelegt werden. Die Größe einer 'Union' ergibt sich aus der Größe ihres größten Elements.

Syntax

Beispiel

```
StructureUnion
  Feld1.Type
  Feld2.Type
  ...
EndStructureUnion

Structure Typ
  Name$
  StructureUnion

  Long.l      ; Jedes Feld (Long, Float und String)
               ; befindet sich
  Float.f     ; an derselben Speicheradresse,
               ; weshalb immer nur eines
  String.s    ; davon benutzt werden kann.

EndStructureUnion
EndStructure
```

7.9 Zeiger (Pointer)

Einführung

Während eine Variable der „Beschriftung“ eines Speicherplatzes entspricht und somit dessen Inhalt (z.B. Zahlenwert) repräsentiert, beinhaltet ein Zeiger nicht den Wert sondern die zugehörige Speicheradresse. Man könnte sagen: der Zeiger (englisch: Pointer) „zeigt“ auf die Variable.

Programmiertechnisch gesehen ist ein Zeiger eine Long-Variable, welche eine Adresse speichert.

Benutzung von Zeigern

In PureBasic werden Zeiger durch das Voranstellen eines * vor den Zeigernamen gekennzeichnet. Sie werden generell in Verbindung mit dem Struktur-Typ benutzt. Mit Hilfe des Zeigers erhalten Sie Zugriff auf die Struktur.

Beispiel

```
*MyScreen.Screen = OpenScreen(0,320,200,8,0)
mouseX = *MyScreen\MouseX      ; setzt voraus, dass die
                                ; Screen Struktur ein MouseX
                                ; Feld beinhaltet
```

Es gibt drei gültige Methoden, den Wert eines Zeigers zu setzen:

- Erhalt als Ergebnis einer Funktion (wie im obigen Beispiel ge-

- zeigt)
- Kopieren des Werts von einem anderen Zeiger
- Ermitteln der Adresse einer Variable, einer Prozedur oder einer Sprungmarke (siehe weiter unten)

Adressen von Variablen

Um die Adresse einer Variable in Ihrem Programmcode zu ermitteln, benutzen Sie das 'at' Symbol (@). Ein häufiger Grund für die Benutzung dieser Methode ist, wenn Sie die Variable eines Struktur-Typs an eine Prozedur übergeben wollen. Sie müssen einen Zeiger auf diese Variable übergeben, da Sie strukturierte Variablen nicht direkt übergeben können.

Beispiel

```
Structure Struktur           ; Definition einer
                             ; Beispiel-Struktur
    a.w
    b.l
    c.w
EndStructure
Procedure SetB(*Zeiger.Struktur) ; Definition einer Prozedur
    ; die den Zeiger erhält
    *Zeiger\b = 69             ; Die Variable b.w in der
    ; Struktur erhält mit
    ; Hilfe des Zeigers den
    ; Wert 69 zugewiesen
EndProcedure
If OpenConsole()
    DefType.Struktur Var      ; Oeffnen Konsolen-Fensters
    ; Deklaration einer
    ; Variable 'Var' vom
    ; Typ Struktur
    Var\b = 0                 ; Die Variable b.w in der
    ; Struktur 'Var' erhält den
    ; Wert 0
    SetB( @Var )              ; Aufrufen der Prozedur SetB
    ; mit dem Zeiger auf 'Var' als
    ; Parameter
    PrintN(Str(Var\b))        ; Ausgabe von '69' als
    ; Nachweis dass der Wert
    ; erfolgreich gesetzt wurde
    Input()                   ; Wartet auf ein Drücken der
    ; 'Return' - Taste
    CloseConsole()            ; Schließt das Konsolenfenster
EndIf
End
```

Adressen von Prozeduren

Diese Funktion ist insbesondere für fortgeschrittene Programmierer gedacht, die darauf angewiesen sind, die Adresse einer Prozedur zu

ermitteln. Der wahrscheinlich häufigste Grund für das Ermitteln einer Adresse liegt darin, wenn auf "low-level" (unterster) Ebene mit dem OS gearbeitet werden soll. Einige OS erlauben (für einige Operationen) die Definition sogenannter "Callback"- oder "Hook"-Funktionen, welche durch das OS aufgerufen werden und dem Programmierer den Ausbau der Fähigkeiten der betreffenden OS-Routine ermöglichen. Die Adresse einer Prozedur wird auf ähnliche Art und Weise wie bei Variablen ermittelt.

Beispiel

```
Procedure WindowCB(WindowID.l, Message.l, ↵
    wParam.l, lParam.l)
    ; Hier wird Ihre Callback Prozedur abgearbeitet.
EndProcedure

; Ein spezielles Callback für Windows OS
; ermöglicht die Verarbeitung von
; Window-Ereignissen.
SetWindowCallback( @WindowCB() )
```

Adressen von Sprungmarken (Labels)

Es kann auch nützlich sein, die Adresse einer Sprungmarke innerhalb Ihres Programmcodes zu ermitteln. Dies ist nötig, wenn Sie Zugriff auf an dieser Sprungmarke gespeicherten Code oder Daten erhalten möchten. Oder jede andere gute Möglichkeit, die Sie sich vorstellen können. Um die Adresse einer Sprungmarke zu ermitteln, schreiben Sie ein Fragezeichen (?) vor den Namen der Sprungmarke.

Beispiel

```
If OpenConsole()
    ; Hier wird durch Ausrechnen der Differenz
    ; zwischen den Adressen der Sprungmarken
    ; vor und nach den eingefügten Daten die
    ; Größe des Datenblocks im Speicher ermittelt.
    PrintN("Größe der Daten = " + Str(?EndeDaten - ? ↵
        BeginnDaten))
    Input()
    CloseConsole()
EndIf
End

BeginnDaten:
IncludeBinary "Datei.bin"
EndeDaten:
```

8. Operatoren

Da wir unsere mit Hilfe der Datentypen gespeicherten Daten und Informationen ja auch weiterverarbeiten möchten, benötigen wir dazu (unter anderem) die Operatoren.

Diese kommen in Ausdrücken vor, wo die Operanden (z.B. Variablen, Konstanten etc.) durch einen oder mehrere Operatoren miteinander verknüpft werden.

PureBasic bietet folgende Operatoren:

- Einfacher Zuweisungsoperator (=)
- Vorzeichen (-)
- Arithmetische Operatoren (+ - * /)
- Vergleichsoperatoren (= < > <= >= <>)
- Logische Operatoren (And, Or)
- Bitweise Operatoren (& | ! ~ << >>)
- Klammern ()

8.1 Der einfache Zuweisungsoperator '='

Dieser Operator speichert einen Wert in einer Variablen, indem er ihr den Wert des Ausdrucks auf der rechten Seite des Operators zuweist. Der Einfachheit kann dazu auch „ist gleich“ (z.B. die Variable auf der linken Seite „ist gleich“ dem Ergebnis des Ausdrucks auf der rechten Seite) gesagt werden.

Beispiel

```
Zahl = 10           ; weist dem Speicherplatz  
                   ; (=Variable) mit Namen 'Zahl'  
                   ; (als Standard-Typ) den Wert 10 zu  
Ergebnis.l = Zahl + 20 ; weist der Long-Variablen  
                   ; 'Ergebnis' das Ergebnis des  
                   ; des Ausdrucks „Zahl + 20“ (in  
                   ; diesem Fall 30) zu  
Zahl = -50          ; Der alte Wert 10 der  
                   ; Variablen 'Zahl' wird mit -50  
                   ; überschrieben
```

Der Operator '=' kommt auch als „Gleich“ bei den Vergleichsoperatoren zum Einsatz. Siehe hierzu das Kapitel 8.3.

8.2 Vorzeichen

Als einziges Vorzeichen für eine numerische Variable kommt in PureBasic das Minus (-) in Frage. Damit wird die Verwendung des negativen

Werts einer Variablen angezeigt. Das Vorzeichen wird stets vorrangig vor weiteren arithmetischen Operationen behandelt.

Beispiel

```
a = 10 : a = 5 ; Deklaration der Variablen 'a' und 'b' mit
                ; ihren anfänglichen Inhalten
Wert = a + -b  ; Rechnet mit dem negativen Wert von
                ; 'b' (-5) und subtrahiert
                ; diesen Wert von 'a' (10). 'Wert'
                ; enthält als Ergebnis schließlich 15.
                ; („Minus minus“ ergibt „Plus“)
```

8.3 Arithmetische Operatoren

Arithmetische Operatoren werden zum Ausführen mathematischer Berechnungen benötigt. Folgende Operatoren sind in PureBasic möglich:

+	Addition (Plus)
-	Subtraktion (Minus)
*	Multiplikation
/	Division

In PureBasic können die arithmetischen Operatoren auf zwei verschiedene Arten verwendet werden:

1. Mit Zuweisungsoperator (=): in diesem Fall werden zuerst die Operatoren des Ausdrucks auf der rechten Seite vom '=' ausgewertet und das Ergebnis dann der Variablen auf der linken Seite zugewiesen.

Beispiel

```
Variable = Zahl + 10
```

2. Ohne Zuweisungsoperator (=): in diesem Fall wird der Wert des Ausdrucks bzw. der Variablen auf der rechten Seite mit Hilfe des Operators direkt zur Variablen auf der linken Seite addiert, von ihr subtrahiert usw.

Beispiel

```
Variable + 10
```

oder

```
Variable / 20
```

Bei der Auswertung der Operatoren auf der rechten Seite gelten bestimmte Regeln:

1. Es gilt die mathematische Regel „Punkt vor Strich“. Die Operatoren + und – besitzen die gleiche Priorität, welche jedoch geringer ist als die der Operatoren * und /. Diese Operatoren haben untereinander wieder die gleiche Priorität.

2. Ein negatives Vorzeichen – (ein positives Vorzeichen + gibt es nicht) vor einer Zahl hat jedoch trotzdem eine höhere Priorität als die Operatoren * und /.
3. Bei gleicher Priorität werden die Operatoren von *links nach rechts* abgearbeitet.
4. Möchten Sie bei Ihren Rechenoperationen eine andere Priorität festlegen, müssen Sie *Klammern* verwenden.

+Addition (Plus).

Addiert das Ergebnis des Ausdrucks auf der rechten Seite zum Wert des Ausdrucks auf der linken Seite. Wird das Ergebnis dieses Operators nicht zusammen mit '=' benutzt und es befindet sich eine Variable auf der linken Seite, dann wird der Wert des Ausdrucks der rechten Seite direkt zur Variablen auf der linken Seite addiert.

Beispiel

```
Zahl = Zahl2 + 2 ; Addiert den Wert 2 zu "Zahl2"
                ; und benutzt das Ergebnis mit dem
                ; Zuweisungsoperator =.

Variable + Ausdruck
            ; Der Wert des Ausdrucks "Ausdruck"
            ; wird direkt zur
            ; Variablen "Variable" addiert.

String$=String2$+"Zusatz" ;
                        ; Verknüpft den Inhalt der
                        ; String-Variablen
                        ; 'String2$' mit dem String „Zusatz“ und
                        ; speichert das Ergebnis im 'String$'.
```

- Subtraktion (Minus)

Subtrahiert das Ergebnis des Ausdrucks auf der rechten Seite vom Wert des Ausdrucks auf der linken Seite. Wenn sich auf der linken Seite kein Ausdruck befindet, dann liefert dieser Operator den negativen Wert des Ausdrucks auf der rechten Seite. Wird das Ergebnis dieses Ausdrucks nicht zusammen mit '=' benutzt und es befindet sich eine Variable auf der linken Seite, dann wird der Wert des Ausdrucks der rechten Seite direkt von der Variablen auf der linken Seite subtrahiert. Dieser Operator kann nicht mit Variablen vom Typ 'String' benutzt werden.

Beispiel

```
ar=#Konstante-Sub ; Subtrahiert den Wert von 'Sub' von  
                  ; '#Konstante' und speichert das  
                  ; Ergebnis mit dem Zuweisungsoperator  
                  ; in 'Var'.  
  
Wert=Wert+ -Var    ; Rechnet mit dem negativen Wert  
                  ; von 'Var' und benutzt das Ergebnis  
                  ; mit dem 'Plus' Operator.  
  
Variable-Ausdruck ; Der Wert des Ausdrucks "Ausdrucks"  
                  ; wird direkt von  
                  ; der Variablen "Variable" subtrahiert.
```

*** Multiplikation.**

Multipliziert den Wert des Ausdrucks auf der linken Seite mit dem Wert des Ausdrucks auf der rechten Seite. Wird das Ergebnis dieses Operators nicht zusammen mit '=' benutzt und es befindet sich eine Variable auf der linken Seite, dann wird der Wert der Variablen direkt mit dem Wert des Ausdrucks auf der rechten Seite multipliziert. Dieser Operator kann nicht mit Variablen vom Typ 'String' benutzt werden.

Beispiel

```
Total=Preis*Anzahl ; Multipliziert den Wert von  
                  ; 'Preis' mit dem Wert  
                  ; von "Anzahl" und speichert das  
                  ; Ergebnis mit dem  
                  ; Zuweisungsoperator in der  
                  ; Variablen 'Total'.  
  
Variable*Ausdruck ; 'Variable' wird direkt mit dem  
                  ; Wert von 'Ausdruck'  
                  ; multipliziert und das Ergebnis  
                  ; in 'Variable' gespeichert.
```

/ Division.

Dividiert den Wert des Ausdrucks auf der linken Seite durch den Wert des Ausdrucks auf der rechten Seite. Wird das Ergebnis dieses Operators nicht zusammen mit '=' benutzt und es befindet sich eine Variable auf der linken Seite, dann wird der Wert der Variablen direkt durch den Wert des Ausdrucks auf der rechten Seite dividiert. Dieser Operator kann nicht mit Variablen vom Typ 'String' benutzt werden.

Beispiel

```
Anzahl=Total/Preis ; Dividiert den Wert von 'Total' durch
                    ; den Wert von 'Preis' und benutzt das
                    ; Ergebnis mit dem Zuweisungsoperator.

Variable/Ausdruck  ; 'variable' wird direkt durch den
                    ; Wert von ,Ausdruck' dividiert und
                    ; das Ergebnis in 'Variable'
                    ; gespeichert.
```

8.4 Vergleichsoperatoren

Die Vergleichsoperatoren (auch relationale Operatoren genannt) werden zum Formulieren von Bedingungen benötigt. Ein Vergleich kann positiv oder negativ ausfallen, d.h. es gibt nur zwei Möglichkeiten:

- der Vergleich ist wahr (englisch: TRUE)
- der Vergleich ist falsch (englisch: FALSE).

Vergleiche werden daher hauptsächlich für Abfragen verwendet, ob eine im Programm formulierte Bedingung zutrifft oder nicht. Abhängig vom Ergebnis (wahr oder falsch) einer solchen Abfrage erfolgt dann eine weitere Verzweigung im Programm.

In PureBasic sind folgende Vergleichsoperatoren möglich:

=	Gleich
<	Kleiner als
>	Größer als
<=	Kleiner als oder gleich
>=	Größer als oder gleich
<>	Ungleich

Die Vergleichsoperatoren haben eine geringere Priorität als die arithmetischen Operatoren, d.h. Rechenoperationen innerhalb eines Ausdrucks werden vor einem Vergleich ausgeführt.

= Gleich

Dies wird benutzt, um die Werte der Ausdrücke auf der linken und der rechten Seite des Operators zu vergleichen. Ist der Wert des Ausdrucks auf der linken Seite gleich dem Wert des Ausdrucks auf der rechten Seite, dann ergibt dieser Operator als Ergebnis wahr (TRUE), andernfalls das Ergebnis falsch (FALSE).

Hinweis

Das Zeichen '=' kommt auch als Zuweisungsoperator zum Einsatz, sehen Sie hierzu das gesonderte Kapitel 8.1.

Beispiel:

```
If abc=def ; Testet, ob die Werte der Variablen
           ; 'abc' und 'def' die gleichen sind und benutzt
           ; dieses Ergebnis für den If Befehl
```

< Kleiner als

Dies wird benutzt, um die Werte der Ausdrücke auf der linken und der rechten Seite des Operators zu vergleichen. Ist der Wert des Ausdrucks auf der linken Seite kleiner als der Wert des Ausdrucks auf der rechten Seite, dann ergibt dieser Operator als Ergebnis wahr (TRUE), andernfalls das Ergebnis falsch (FALSE).

Beispiel

```
If abc<def ; Testet, ob der Wert der
           ; Variablen 'abc' kleiner als der
           ; Wert der Variablen 'def' ist und benutzt
           ; dieses Ergebnis für den If Befehl
```

> Größer als

Dies wird benutzt, um die Werte der Ausdrücke auf der linken und der rechten Seite des Operators zu vergleichen. Ist der Wert des Ausdrucks auf der linken Seite größer als der Wert des Ausdrucks auf der rechten Seite, dann ergibt dieser Operator als Ergebnis wahr (TRUE), andernfalls das Ergebnis falsch (FALSE).

Beispiel

```
If abc>def ; Testet, ob der Wert der Variablen
           ; 'abc' größer als der
           ; Wert der Variablen 'def' ist und benutzt
           ; dieses Ergebnis für den If Befehl
```

<= Kleiner als oder gleich

Dies wird benutzt, um die Werte der Ausdrücke auf der linken und der rechten Seite des Operators zu vergleichen. Ist der Wert des Ausdrucks auf der linken Seite kleiner als oder gleich dem Wert des Ausdrucks auf der rechten Seite, dann ergibt dieser Operator als Ergebnis wahr (TRUE), andernfalls das Ergebnis falsch (FALSE).

Beispiel

```
If abc<=def ; Testet, ob der Wert der
              ; Variablen 'abc' kleiner oder
              ; gleich dem Wert der Variablen 'def'
              ; ist und benutzt dieses
              ; Ergebnis für den If Befehl
```

>= Größer als oder gleich

Dies wird benutzt, um die Werte der Ausdrücke auf der linken und der rechten Seite des Operators zu vergleichen. Ist der Wert des Ausdrucks auf der linken Seite größer als oder gleich dem Wert des Ausdrucks auf der rechten Seite, dann ergibt dieser Operator als Ergebnis wahr (TRUE), andernfalls das Ergebnis falsch (FALSE).

Beispiel

```
If abc>=def ; Testet, ob der Wert der Variablen 'abc'
              größer oder
              ; gleich dem Wert der Variablen 'def' ist und
benutzt dieses
              ; Ergebnis für den If Befehl
```

<> Ungleich (nicht gleich zu).

Dies wird benutzt, um die Werte der Ausdrücke auf der linken und der rechten Seite des Operators zu vergleichen. Ist der Wert des Ausdrucks auf der linken Seite gleich dem Wert des Ausdrucks auf der rechten Seite, dann ergibt dieser Operator als Ergebnis falsch (FALSE, andernfalls das Ergebnis wahr (TRUE).

Beispiel

```
If abc<>def ; Testet, ob der Wert der
              ; Variablen 'abc' ungleich (kleiner
              ; oder größer) dem Wert der Variablen 'def'
              ; ist und benutzt
              ; dieses Ergebnis für den If Befehl
```

8.5 Logische Operatoren

Logische Operatoren verknüpfen die logischen Werte (TRUE oder FALSE) zweier Ausdrücke und geben abhängig vom Operator das Ergebnis 0 (logisch falsch / FALSE) bzw. ein Ergebnis ungleich 0 (logisch richtig / TRUE) zurück.

PureBasic verwendet folgende logischen Operatoren:

And	Logisches Und
Or	Logisches Oder

Mit logischen Operatoren werden in der Regel die Ergebnisse von Vergleichsoperatoren (siehe Kapitel 8.3) verknüpft, es können jedoch keine Variablen verknüpft werden. Dazu dienen die bitweisen Operatoren, siehe hierzu das Kapitel 8.5.

And

Logisches AND (Und). Kann zum Kombinieren der logisch wahren (TRUE) oder falschen (FALSE) Ergebnisse der Vergleichsoperatoren benutzt werden und ergibt die in der folgenden Tabelle enthaltenen Resultate.

<i>Bedingung 1</i>	<i>Bedingung 2</i>	<i>Ergebnis</i>
Falsch	Falsch	Falsch
Falsch	Wahr	Falsch
Wahr	Falsch	Falsch
Wahr	Wahr	Wahr

Beispiel

```
If a=b And c>10 ; Testet im ersten Schritt, ob die Werte
                  ; der Variablen 'a' und 'b' die gleichen
                  ; sind und ob der Wert der
                  ; Variablen 'c' größer als 10 ist.
                  ; Im zweiten Schritt
                  ; erfolgt die Verknüpfung der beiden
                  ; Ergebnisse der Vergleiche mittels 'And'.
                  ; Nur wenn die Ergebnisse
                  ; beider Vergleiche wahr ergaben, ist auch
                  ; das Ergebnis der logischen Verknüpfung
                  ; wahr und führt somit zur Ausführung des
                  ; in der If Bedingung enthaltenen
                  ; Programmcodes.
```

Or

Logisches OR (Oder). Kann zum Kombinieren der logisch wahren (TRUE) oder falschen (FALSE) Ergebnisse der Vergleichsoperatoren benutzt werden und ergibt die in der folgenden Tabelle enthaltenen Resultate.

<i>Bedingung 1</i>	<i>Bedingung 2</i>	<i>Ergebnis</i>
--------------------	--------------------	-----------------

Falsch	Falsch	Falsch
Falsch	Wahr	Wahr
Wahr	Falsch	Wahr
Wahr	Wahr	Wahr

Beispiel

```
If x<y Or z=20 ; Testet im ersten Schritt, ob der Wert der
                ; Variablen 'x' kleiner als der Wert von
                ; 'y' ist und ob der Wert der Variablen 'z'
                ; gleich 10 ist. Im zweiten Schritt
                ; erfolgt die Verknüpfung der beiden
                ; Ergebnisse der Vergleiche mittels 'Or'.
                ; Ist wenigstens eines der beiden
                ; Ergebnisse der Vergleiche wahr, so ergibt
                ; auch das Ergebnis der logischen
                ; Verknüpfung wahr und führt
                ; somit zur Ausführung des in der
                ; If-Bedingung enthaltenen Programmcodes.
```

8.6 Bitweise Operatoren

Bitweise Operatoren dienen – wie der Name schon sagt – zur Verknüpfung von Variablen, und dies Bit für Bit. Sie sollten sich daher mit Binärzahlen auskennen, wenn Sie diese Operatoren anwenden möchten. Nicht möglich ist die Verwendung dieser Operatoren zusammen mit String-Variablen.

In PureBasic sind folgende bitweise Operatoren möglich:

&	Bitweises AND
	Bitweises OR
!	Bitweises XOR
~	Bitweises NOT
<<	Bitweises Linksschieben (Shift)
>>	Bitweises Rechtsschieben (Shift)

& Bitweises AND

Das Ergebnis dieses Operators ist der Wert des Ausdrucks auf der linken Seite, welcher durch 'AND' (Und) mit dem Wert des Ausdrucks auf der rechten Seite verknüpft wird, und dies paarweise Bit für Bit. Der Wert

jedes Bits wird entsprechend der nachfolgenden Tabelle gesetzt. Wird das Ergebnis des Operators nicht zusammen mit '=' benutzt und es befindet sich eine Variable auf der linken Seite, dann wird das Ergebnis des Ausdrucks direkt in dieser Variable gespeichert.

<i>Bit der Variable 1</i>	<i>Bit der Variable 2</i>	<i>Ergebnis</i>
0	0	0
0	1	0
1	0	0
1	1	1

Beispiel

```
; Zeigt die Benutzung mit Binär-Zahlen,
; da es so leichter ist, das Ergebnis zu sehen
a.w = %1000 & %0101 ; Ergebnis wird 0 sein

b.w = %1100 & %1010 ; Ergebnis wird %1000 sein

bits = a & b          ; verknüpft jedes Bit von a und b
                      ; durch AND und benutzt das Ergebnis
                      ; mit dem 'Gleich' Operator

a & b                 ; verknüpft jedes Bit von a und b
                      ; durch AND und speichert das Ergebnis
                      ; direkt in der Variable 'a'
```

| Bitweises OR.

Das Ergebnis dieses Operators ist der Wert des Ausdrucks auf der linken Seite, welcher durch 'OR' (Oder) mit dem Wert des Ausdrucks auf der rechten Seite verknüpft wird, und dies paarweise Bit für Bit. Der Wert jedes Bits wird entsprechend der nachfolgenden Tabelle gesetzt. Wird das Ergebnis des Operators nicht zusammen mit '=' benutzt und es befindet sich eine Variable auf der linken Seite, dann wird das Ergebnis des Ausdrucks direkt in dieser Variable gespeichert.

<i>Bit der Variable 1</i>	<i>Bit der Variable 2</i>	<i>Ergebnis</i>
0	0	0
0	1	1
1	0	1
1	1	1

Beispiel

```
; Zeigt die Benutzung mit Binär-Zahlen, da es so
; leichter ist, das Ergebnis zu sehen
a.w = %1000 | %0101 ; Ergebnis wird %1101 sein

b.w = %1100 | %1010 ; Ergebnis wird %1110 sein

bits = a | b          ; verknüpft jedes Bit von a und b durch
                      ; OR und benutzt das Ergebnis mit dem
                      ; 'Gleich' Operator

a | b                 ; verknüpft jedes Bit von a und b durch
                      ; OR und speichert das Ergebnis direct
                      ; in der Variable 'a'
```

! Bitweises XOR.

Das Ergebnis dieses Operators ist der Wert des Ausdrucks auf der linken Seite, welcher durch 'XOR' (Exklusives Oder) mit dem Wert des Ausdrucks auf der rechten Seite verknüpft wird, und dies paarweise Bit für Bit. Der Wert jedes Bits wird entsprechend der nachfolgenden Tabelle gesetzt. Wird das Ergebnis des Operators nicht zusammen mit '=' benutzt und es befindet sich eine Variable auf der linken Seite, dann wird das Ergebnis des Ausdrucks direkt in dieser Variable gespeichert.

<i>Bit der Variable 1</i>	<i>Bit der Variable 2</i>	<i>Ergebnis</i>
0	0	0
0	1	1
1	0	1
1	1	0

Beispiel

```
; Zeigt die Benutzung mit Binär-Zahlen, da es so leichter
; ist, das Ergebnis zu sehen
a.w = %1000 ! %0101 ; Ergebnis wird %1101 sein

b.w = %1100 ! %1010 ; Ergebnis wird %0110 sein

bits = a ! b          ; verknüpft jedes Bit von a und b durch
                      ; XOR und benutzt das Ergebnis mit dem
                      ; 'Gleich' Operator

a ! b                 ; verknüpft jedes Bit von a und b durch
                      ; XOR und speichert das Ergebnis direct
                      ; in der Variable 'a'
```

~ Bitweises NOT.

Das Ergebnis dieses Operators ist der – durch 'NOT' (Nicht bzw. Negation) verknüpfte – Wert des Ausdrucks auf der rechten Seite, und dies Bit für Bit. Der Wert jedes Bits wird von 0 in 1 umgewandelt bzw. umgekehrt. Siehe hierzu die nachfolgende Tabelle:

<i>Bit der Variable</i>	<i>Ergebnis</i>
0	1
1	0

Beispiel

```
; Zeigt die Benutzung mit Binär-Zahlen, da es so leichter
; ist, das Ergebnis zu sehen
a.w = ~%1000 ; Ergebnis wird %0111 sein

b.w = ~%1010 ; Ergebnis wird %0101 sein
```

<< Bitweises Shift (Verschieben) nach Links

Durch die Verwendung dieses Operators verschieben Sie die einzelnen Bits der Variablen auf der linken Seite des Operators um die auf der rechten Seite des Operators angegebene Anzahl an Bits nach links. Weggefallene Bits auf der rechten Seite der Variablen werden mit 0 aufgefüllt.

Beispiel

```
m = a << x ; entspricht einer x-maligen
            ; Multiplikation von a mit 2

m = 2 << 1 ; ergibt 4
```

Anmerkung

Die Ausführung dieser Verschiebeoperation ist in der Regel weniger aufwendig als die vergleichbare Multiplikation.

>> Bitweises Shift (Verschieben) nach Rechts.

Durch die Verwendung dieses Operators verschieben Sie die einzelnen Bits der Variablen auf der linken Seite des Operators um die auf der rechten Seite des Operators angegebene Anzahl an Bits nach rechts. Weggefallene Bits auf der linken Seite der Variablen werden mit 0 aufgefüllt.

Beispiel

```
m = a >> x    ; entspricht einer x-maligen Division  
               ; von a durch 2  
  
m = 8 >> 2     ; ergibt 2
```

Anmerkung

Die Ausführung dieser Verschiebeoperation ist in der Regel weniger aufwendig als die vergleichbare Division.

Nachfolgend soll noch einmal die Wirkung der einzelnen „bitweisen“ Operatoren an einem Beispiel erläutert werden:

```
i = 2 : n = 1 ; Deklaration der Beispiel-Variablen
m = i & n      ; ergibt m = 0
                ; ein Bit von m ist nur dann auf 1 gesetzt,
                ; wenn die entsprechenden Bits von i und n
                ; gleichzeitig auf 1 gesetzt sind

m = i | n      ; ergibt m = 3
                ; ein Bit von m ist dann auf 1 gesetzt, wenn
                ; mindestens eines der entsprechenden Bits
                ; von i und n auf 1 gesetzt ist

m = i ! n      ; ergibt m = 3
                ; ein Bit von m ist dann auf 1 gesetzt, wenn
                ; die entsprechenden Bits von i und n
                ; verschiedene Werte besitzen

m = ~i         ; ergibt m = -3
                ; jedes Bit von m wird aus dem Komplement des
                ; entsprechenden Bits aus i gebildet

m = i << n     ; ergibt m = 4
                ; entspricht einer n-maligen Multiplikation
                ; von i mit 2

m = i >> n     ; ergibt m = 1
                ; entspricht einer n-maligen Division
                ; von i durch 2
```

8.7 Klammern

Klammern sind keine Operatoren im eigentlichen Sinne. Sie sind aber im Zusammenhang mit anderen Operatoren elementar wichtig, wenn Sie die Reihenfolge der Abarbeitung von arithmetischen Operationen innerhalb eines Ausdrucks bestimmen wollen.

Durch paarweises Setzen von Klammern können Sie einen Teil eines Ausdrucks zuerst berechnen und damit das Ergebnis eines Ausdrucks wesentlich beeinflussen.

Beispiel

```
a = (5 + 6) * 3 ; Ergebnis wird 33 sein,
                ; da 5 + 6 zuerst berechnet wird
```

Die gleiche Berechnung ohne Klammern sähe so aus:

```
a = 5 + 6 * 3   ; Ergebnis wird 23 sein, da 6 * 3
                ; zuerst berechnet wird
                ; („Punkt vor Strich“ - Rechnung)
```

Beispiel

```
b = 4 * (2 - (3 - 4)) ; Ergebnis wird 12 sein,  
                      ; da zuerst 3 - 4 berechnet wird,  
                      ; dann 2 - Zwischenergebnis (-1),  
                      ; danach die Multiplikation
```

9. Ablaufsteuerung

9.1 Grundlagen und Zweck der Ablaufsteuerung

Die Ablaufsteuerung eines Programms dient zur gezielten Abarbeitung von Informationen und Daten innerhalb eines Programms. Während ein Programm normalerweise nacheinander von der ersten bis zur letzten Anweisung (von oben nach unten im Programmcode) abgearbeitet wird, kann diese Reihenfolge durch den Programmierer unterbrochen und gesteuert werden.

Neben den bereits bekannten Operatoren (z.B. Vergleiche) kommen bei der Ablaufsteuerung Bedingungen, Schleifen und Sprungbefehle/Unterprogramme zum Einsatz, die wir uns in diesem Kapitel näher ansehen wollen.

9.2 Bedingungen

Vielfach muss ein Programm während der Ausführung entscheiden, ob der eine oder der andere Befehl, dieser oder jener Codeabschnitt etc. auszuführen ist. Hierfür werden Bedingungen (oft auch Verzweigungen genannt) benötigt. Eine Bedingung wird bei der Programmierung dazu verwendet, um abhängig davon – ob ein vorgegebener Ausdruck wahr oder falsch ist (die Bedingung erfüllt oder nicht erfüllt ist) – einen dazugehörigen Programmabschnitt aufzurufen.

In PureBasic gibt es zwei Möglichkeiten, die bedingte Ausführung von Programmcode zu verwirklichen:

- *If : Else : EndIf*
- *Select : EndSelect*

Die Schlüsselwörter *If : Else : EndIf* sind auch aus anderen Programmiersprachen (BASIC, C, JAVA) wohlbekannt. Damit können Sie einfache Verzweigungen realisieren.

Mit den Schlüsselwörtern *Select : EndSelect* (in C oder Java als Schlüsselwort „Switch“ bekannt) können Sie bequem eine ganze Liste an Bedingungen abarbeiten, indem Sie mittels dem dazugehörigen Schlüsselwort „Case“ zwischen einer Vielzahl an Alternativen auswählen können.

If : Else : EndIf

Die einfachste Form eines solchen Bedingungsblocks sieht wie folgt aus:

```
If <Ausdruck> ; Ist der angegebene Ausdruck "wahr"?  
    <Anweisung> ; Ja: Ausführung der Anweisung; Nein: Programmfortsetzung nach EndIf  
EndIf ; Abschluss des Bedingungsblocks
```

Um den gleichen Bedingungsblock auf einer Programmzeile darstellen zu können, benutzen Sie folgenden Code:

```
If <Ausdruck> : <Anweisung> : EndIf
```

Das aus manch anderen BASIC-Dialekten bekannte „Then“ finden Sie dagegen in PureBasic nicht. Benutzen Sie bitte hierfür einfach die eben angeführte einzeilige Version der *If: EndIf* Bedingung.

PureBasic bietet im Rahmen der *If : EndIf* Bedingungsblöcke jedoch auch noch zwei weitere optionale Schlüsselwörter, die wie folgt zum Einsatz kommen:

```
If <Ausdruck1> ; Ist der angegebene  
    ; Ausdruck1 "wahr"?  
    Anweisung1 ; Ja: Ausführung der Anweisung1 -  
    ; Nein: Fortfahren im Code  
[ElseIf <Ausdruck2>] ; Ausdruck1="falsch" => Ist  
    ; alternativ der Ausdruck2 "wahr"?  
    Anweisung2 ; Ja: Ausführung der Anweisung2 -  
    ; Nein: Fortfahren im Code  
[Else] ; Ausdruck1 und Ausdruck2 waren  
    ; "falsch", daher:  
    Anweisung3 ; Ausführung der Anweisung3  
EndIf
```

Wie Sie sicher erkannt haben, dient das *ElseIf* dazu, einen anderen (alternativen) Codeabschnitt auszuführen, wenn der zum *ElseIf* gehörende Ausdruck „wahr“ ergibt. Man könnte als deutsche Übersetzung also auch „Anderenfalls, wenn“ sagen. *ElseIf* kann einfach oder mehrfach in einem *If: EndIf* Block vorkommen.

Else ist schließlich dazu da, einen weiteren Teil des Programmcodes auszuführen, wenn der/die zu *If* und ggf. *ElseIf* gehörende(n) Test(s) falsch ergeben. Übersetzt lautet der Sinn einfach „Anderenfalls“ (ohne weitere Bedingung).

Beispiel

```
If a=10
    PrintN("a=10")
ElseIf a=20
    PrintN("a=20")
Else
    PrintN("a ist weder 10 noch 20")
EndIf
```

Es kann auch eine beliebige Anzahl an *If* Bedingungen ineinander verschachtelt werden.

Ein Beispiel hierzu:

```
If a=10 And b>=10 Or c=20
    If b=15
        PrintN("b=15")
    Else
        PrintN("Andere Möglichkeit")
    EndIf
Else
    PrintN("Test-Fehler")
EndIf
```

Select : EndSelect

Diese Form des Bedingungsblocks kommt häufig dann zur Anwendung, wenn Sie abhängig vom Wert einer Variablen (bzw. eines ganzen Ausdrucks) zu verschiedenen Befehlsblöcken, Unterprogrammen etc. verzweigen möchten.

Die grundlegende Syntax eines solchen Bedingungsblocks sieht wie folgt aus:

```
Select <Ausdruck>
  Case <Ausdruck1>
    Anweisungen 1
    ...
  [Case <Ausdruck2>]
    Anweisungen 2
    ...
  .
  .
  .
  [Case <AusdruckX>]
    Anweisungen X
    ...
  [Default]
    Standard-Anweisungen
    ...
EndSelect
```

Innerhalb des *Select : EndSelect* Abschnitts können ein oder mehrere der *Case*-Blöcke vorkommen, ebenso ist die Angabe eines *Default*-Blocks (der „Standard“-Block, der ausgeführt wird, wenn keiner der *Case*-Ausdrücke zutrifft) optional. Die Reihenfolge der *Case*- und *Default*-Marken ist beliebig. Der Wert eines verwendeten Ausdrucks muss jedoch eindeutig sein, so ist z.B. die Verwendung von Vergleichsoperatoren wie „ $a < 2$ “ nicht gestattet.

Das PureBasic – Programm wertet den bei *Select* angegebenen Ausdruck aus und vergleicht das Ergebnis mit den Werten der einzelnen *Case*-Ausdrücke.

Wird eine Übereinstimmung gefunden, verzweigt das Programm zu der entsprechenden *Case*-Marke, führt den dazugehörigen Programmcode aus und beendet die *Select : EndSelect* Bedingung. Wird keine Übereinstimmung gefunden, so verzweigt das Programm zur *Default*-Marke, falls diese angegeben wurde.

Wird innerhalb der *Select : EndSelect* keinerlei Übereinstimmung gefunden und es ist auch keine *Default*-Marke angegeben, wird keine Anweisung des *Select : EndSelect* Blocks ausgeführt und dieser damit komplett übersprungen.

Beispiel

```
a = 2
Select a
  Case 1
    PrintN("a = 1")
  Case 2
    PrintN("a = 2")
  Case 20
    PrintN("a = 20")
  Default
    PrintN("a enthält einen anderen Wert.")
EndSelect
```

Bei Verwendung der *Select : EndSelect* Bedingungsblöcke kann es in Einzelfällen notwendig sein, den *Select*-Block vorzeitig zu verlassen, um (mit dem Befehl *Goto*) zu einem anderen Programmteil außerhalb des Bedingungsblocks zu springen. Hierfür benötigen Sie das Schlüsselwort *FakeEndSelect*.

Dieses simuliert ein *EndSelect*, ohne es tatsächlich auszuführen. Wenn Sie also *Goto* innerhalb Ihres *Select : EndSelect* Bedingungsblocks verwenden, achten Sie unbedingt auf vorherige Benutzung von *FakeEndSelect*, da andernfalls Ihr Programm abstürzen wird.

Beispiel

```
Hauptschleife:
...
...
Select a
  Case 10
    ...
  Case 20
    FakeEndSelect
    Goto Hauptschleife
EndSelect
```

Zum Abschluss dieses Kapitels wollen wir uns noch einmal die beiden in PureBasic möglichen Bedingungsblöcke in der direkten Gegenüberstellung ansehen:

If : Else : EndIf

Select : EndSelect

```
a = 2
If a=1
    PrintN("a = 1")
ElseIf a=20
    PrintN("a = 2")
ElseIf a=20
    PrintN("a = 20")
Else
    PrintN("a enthält einen
anderen Wert.")
EndIf
```

```
a = 2
Select a
    Case 1
        PrintN("a = 1")
    Case 2
        PrintN("a = 2")
    Case 20
        PrintN("a = 20")
    Default
        PrintN("a enthält einen
anderen Wert.")
EndSelect
```

Wie Sie sehen, können Sie problemlos beide Varianten verwenden. Bitte beachten Sie dabei jedoch die Einschränkung bezüglich der eindeutigen Werte bei Select/Case-Werten. Entscheiden Sie sich abhängig von Ihrer zu lösenden Aufgabe einfach für die von Ihnen bevorzugte Form.

9.3 Schleifen

Neben der bedingten Ausführung von Anweisungen, die wir uns eben näher angesehen haben, gibt es auch noch die Möglichkeit, dass Sie Anweisungen mehrfach ausführen müssen. Dafür werden in PureBasic wie in den meisten anderen Programmiersprachen die „Schleifen“ verwendet.

Es gibt drei verschiedene Arten von Schleifen:

For : Next

Repeat : Until

While : Wend

Die For : Next Schleife

Bei dieser Schleifenform wird von vornherein festgelegt, wie oft die Schleife durchlaufen wird und damit die in der Schleife enthaltenen Anweisungen ausgeführt werden. Dabei bewegt sich eine Variable (auch Laufzeitvariable genannt) schrittweise durch einen Wertebereich.

Die grundlegende Syntax einer solchen Schleife sieht wie folgt aus:

```
For <Variable> = <Ausdruck1> To <Ausdruck2> [Step <Kon-  
stante>]  
    Anweisung  
    ...  
Next [<Variable>]
```

Bei dieser Schleife erhält die <Variable> ihren Anfangswert durch den <Ausdruck1> zugeteilt. In jedem Schleifendurchlauf wird dieser Wert um den Faktor eins erhöht (oder um den Wert, der hinter dem optionalen *Step* angegeben wurde). Wenn die <Variable> gleich dem <Ausdruck2> ist, wird die Schleife nach dem Erreichen von *Next* beendet. Die nochmalige Angabe der <Variable> hinter dem ist optional und soll bei Verwendung lediglich der besseren Übersicht – welches *Next* zu welchem *For* gehört – dienen.

Beispiel

```
For k = 0 To 10  
    ...  
Next
```

In diesem Beispiel wird die Schleife 11-mal (0 bis 10) durchlaufen und dann beendet.

Beispiel

```
a = 2  
b = 3  
For k = a + 2 To b + 7 Step 2  
    ...  
Next k
```

In diesem Beispiel führt das Programm vier Schleifendurchläufe vorm Beenden aus (k wird in jeder Schleife um den Wert 2 erhöht, so ergeben sich folgende Werte von k: 4 – 6 – 8 – 10). Das "k" nach dem "Next" kennzeichnet, dass "Next" die "For k" Schleife beendet. Wird eine andere Variable angegeben, quittiert dies der Compiler mit einer Fehlermeldung.

Es kann auch sehr nützlich sein, mehrere "For : Next" Schleifen zu verschachteln.

Beispiel

```

For x=0 To 320      ; führt die Schleife 321-mal
                      ; (von x = 0 bis 320) aus
  For y=0 To 200      ; führt die Schleife 201-mal
                      ; (von y = 0 bis 200) aus
    Plot(x, y)         ; zeichnet einen Punkt an den
                      ; Koordinaten x, y
  Next y              ; dies erfolgt insgesamt
                      ; 64521-mal (321 x 201)
Next x               ; und mündet in einem Rechteck vom
                      ; Bereich 0,0 bis 320,200

```

While : Wend

Diese Schleifenart ist der For : Next Schleife sehr ähnlich, da bei beiden die Abfrage der Schleifenbedingung am Anfang stattfindet. Die Grundform einer solchen Schleife sieht wie folgt aus:

```

While <Ausdruck>
  Anweisung
  ...
Wend

```

Eine *While* : *Wend* Schleife wird solange durchlaufen, bis der <Ausdruck> falsch ergibt. Dabei wird der <Ausdruck> vor jedem Schleifendurchlauf berechnet. Ist das Ergebnis ungleich 0 bzw. wahr, wird die Schleife durchlaufen und die enthaltenen Anweisungen ausgeführt. Ist das Ergebnis dagegen gleich 0 bzw. falsch, wird die Schleife sofort beendet (ohne die Anweisungen in der Schleife auszuführen) und die Programmausführung im Code nach der Schleife fortgesetzt.

Beispiel

```

b = 0                ; deklariert die Variable b mit einem An-
fangswert von 0
a = 10               ; deklariert die Variable a mit einem An-
fangswert von 10
While a = 10         ; "Solange a gleich 10 ist" ...
  b = b + 1          ; erhöhe b um 1
  If b=10             ; "Wenn b gleich 10 ist", dann ...
    a=11              ; setze a auf 11 => die While-Bedingung "a
= 10" wird damit nicht
  Endif              ; mehr erfüllt und deshalb die While :
Wend Schleife sofort
Wend                ; abgebrochen

```

Diese Programmschleife wird so oft ausgeführt, bis der Wert 'a' <> 10 ist. Dies ändert sich, wenn b=10 ist; die Schleife wird insgesamt 10-mal ausgeführt.

Repeat : Until

Die dritte Schleifenform Repeat : Until wird solange ausgeführt, bis ein angegebener Ausdruck wahr ergibt. Im Unterschied zur While : Wend Schleife wird der Wert des Ausdrucks dagegen nicht am Anfang der Schleife, sondern erst am Ende der Schleife überprüft. Damit werden die Schleife und ihre enthaltenen Anweisungen mindestens einmal durchlaufen.

Die Grundform einer solchen Schleife sieht wie folgt aus:

```
Repeat
  Anweisung
  ...
Until <Ausdruck>
```

Dabei wird die Schleife solange durchlaufen, bis der <Ausdruck> wahr ergibt. Eine beliebige Zahl an Schleifendurchläufen ist möglich.

Beispiel

```
a = 0
Repeat
  a = a + 1
Until a > 100
```

Die Schleife wird solange ausgeführt, bis die Variable a einen Wert größer als 100 enthält. Somit wird diese Schleife 101-mal durchlaufen. Wenn Sie eine endlose Schleife benötigen, dann benutzen Sie anstelle von *Until* das Schlüsselwort *Forever*.

Diese Schleife sieht dann wie folgt aus:

```
Repeat
  Anweisung
  ...
Forever
```

Wahrscheinlichste Anwendung für diese Form der Repeat-Schleife dürfte die Verwendung als Hauptschleife in Ihrem Programm sein, die dann lediglich am Programmende mittels *End* abgebrochen wird.

9.4 Sprungbefehle

Die Sprungbefehle in PureBasic verhalten sich genauso wie in anderen bekannten BASIC-Dialekten. Es können Programmteile übersprungen bzw. Unterprogramme angesprungen werden.

Verwendung finden dabei folgende beiden Möglichkeiten:

Gosub : *Return*

Goto

Gosub : Return

Gosub steht als Abkürzung für „Go to sub-routine“ (springe zu einer Unteroutine). Mit *Gosub* und den dazugehörigen Unteroutinen lässt sich sehr einfach ein schneller strukturierter Code zu erstellen.

Grundlegende Verwendung von *Gosub*:

```
; Hauptprogramm
Gosub Sprungmarke
; weiter im Hauptprogramm
End
Sprungmarke:
...
Return
```

Nach dem Schlüsselwort *Gosub* muss eine Sprungmarke angegeben werden, dann wird die Programmausführung ab der Sprungmarke bis zum nächsten *Return* fortgesetzt. Wenn das Schlüsselwort *Return* („Zurück“) erreicht wurde, wird die Unteroutine beendet und die Programmausführung nach dem aufrufenden *Gosub* fortgesetzt.

Hinweis

Am Ende des Programmabschnitts mit Ihrem eigentlichen Hauptprogramm sollte immer das Schlüsselwort *End* stehen. Einmal um das Programm zu beenden, und außerdem um das ungewollte Aufrufen Ihrer Unteroutinen zu vermeiden. Diese sollten sich daher immer am Ende Ihres Programmcodes befinden.

Beispiel

```
OpenConsole()           ; öffnet das Konsolenfenster zum
                        ; Ausgaben des Ergebnisses
a = 1                   ; deklariert die Variable a
                        ; mit dem Anfangswert 1
b = 2                   ; deklariert die Variable b
                        ; mit dem Anfangswert 2
Gosub Berechnung        ; ruft die Unteroutine
                        ; "Berechnung" auf
PrintN(Str(a))          ; gibt das Ergebnis der Berechnung
                        ; in der Unteroutine aus
Input()                ; wartet auf Eingabe von RETURN
End                    ; beendet das Programm und
                        ; schließt das Konsolenfenster

Berechnung:            ; Sprungmarke "Berechnung:" und
                        ; Beginn der Unteroutine
    a = b*2 + a*3 + (a+b) ; Erste Berechnung und Speicherung
                        ; des Ergebnisses in a
    a = a+a*a            ; Zweite Berechnung und Speicherung
                        ; des Ergebnisses in a
Return                 ; Rücksprung in das "
                        ; Hauptprogramm", hier zu:
                        ; PrintN(...)
```

In Einzelfällen kann es notwendig sein, dass Sie eine Unteroutine vorzeitig (d.h. vor dem Erreichen des *Return*) verlassen möchten, um (mit dem Befehl *Goto*) zu einem anderen Programmteil außerhalb der Unteroutine zu springen. Hierfür benötigen Sie das Schlüsselwort *FakeReturn*.

Damit wird ein *Return* simuliert, ohne es tatsächlich auszuführen. Wenn Sie also *Goto* innerhalb einer Unteroutine verwenden, achten Sie unbedingt auf vorherige Benutzung von *FakeReturn*, da andernfalls Ihr Programm abstürzen wird.

Beispiel

```
Gosub Unteroutine
...
Unteroutine:
...
If a = 10
    FakeReturn
    Goto Hauptschleife
Endif
Return
```

Eigentlich sollten Sie jedoch das *FakeReturn* nicht benötigen, da ein ordentlich aufgebautes und strukturiertes Programm kein *Goto* benutzt. Aber manchmal, aus Geschwindigkeitsgründen, kann es etwas helfen.

Goto

Das Schlüsselwort *Goto* steht für „gehe zu“. Mit dieser Anweisung können bestimmte Programmabschnitte einfach übersprungen werden, indem durch Angabe einer Sprungmarke die Programmausführung an diesem Punkt fortgesetzt wird.

Die Verwendung ist ganz einfach:

Goto <Sprungmarke>

Beispiel

```
Hauptschleife:      ; Sprungmarke
<Anweisung>        ; führt hier angegebene Befehle aus
...
Goto Hauptschleife ; springt zur Sprungmarke „Haupt-
tschleife“ zurück
```

Anmerkung

Goto ist eine am meistdiskutierten Anweisungen in höheren Programmiersprachen. Insbesondere BASIC-Programmierer werden wegen dessen (häufiger) Verwendung oftmals belächelt, da damit programmierte Programme oft sehr unübersichtlich und schlecht lesbar werden.

Gut strukturierte Programme sollten auf das Goto verzichten können, wenn es auch Anwendungsfälle gibt, in denen das Goto einen eleganten Ausweg aus sehr umständlichen Programmkonstruktionen bietet.

Merken Sie sich einfach den Grundsatz: „Nicht mehr Goto's als notwendig!“

Seien Sie generell vorsichtig mit dieser Anweisung, da falsche Benutzung zu einem Programmabsturz führen kann...

10. Prozeduren

10.1 Einleitung

Sie haben im vergangenen Kapitel gelernt, wie Sie Programmcode bedingt ausführen lassen können. Es gibt jedoch genauso die Möglichkeit, dass Sie die gleichen Anweisungen mehrfach in Ihrem Programmcode ausführen lassen müssen. Anstatt jetzt diese Anweisungen jedes Mal erneut in den Programmcode einzufügen, können wir diese auch einfach in einem Unterprogramm zusammenfassen und dann dieses Unterprogramm aufrufen.

Während wir Unterroutinen bisher bereits bei den Anweisungen Gosub : Return kennen gelernt haben, können wir durch die Verwendung von Prozeduren auch abgekapselte Unterprogramme erstellen, die eigene Parameter und Variablen besitzen und beliebig oft wiederverwendet werden können.

10.2 Definition und Aufruf

Bei der Definition einer Prozedur unterscheiden wir zwei Arten: normale Prozeduren und Prozeduren mit Rückgabewert (auch Funktionen genannt). Die Funktionen unterscheiden sich dadurch von den „normalen“ Prozeduren, dass sie einen Wert zurückgeben und daher auch in Ausdrücken verwendet werden können.

Eine Prozedur wird wie folgt definiert:

```
Procedure Name(<Variable1>[,<Variable2>,...])  
    ...  
EndProcedure
```

Wie Sie sehen, wird der Beginn der Prozedur (des Unterprogramms) mit dem Schlüsselwort *Procedure* mit einem nachfolgenden „Prozedur-Namen“ angezeigt. In Klammern werden eine oder mehrere Variablen als Prozedur-Parameter definiert – die Angabe eine zweiten und Nach dem innerhalb der Prozedur eingeschlossenen Programmcode wird mit *EndProcedure* die Prozedur abgeschlossen.

Aufgerufen wird eine Prozedur einfach durch Angabe Ihres Namens und innerhalb von nachfolgenden Klammern der/die entsprechende(n) Parameter. Anzahl und Art der Parameter beim Aufrufen der Prozedur müssen mit den bei der Definition der Prozedur geforderten Variablen übereinstimmen.

Beispiel

```
Procedure Vergleich(Wert1.w, Wert2.w)
  If Wert1 > Wert2
    PrintN("Wert1 ist größer als Wert2.")
  ElseIf Wert1 = Wert2
    PrintN("Beide Werte sind gleich.")
  Else
    PrintN("Wert1 ist kleiner als Wert2.")
  EndIf
EndProcedure

Vergleich(15, 30)
```

Nun möchten wir aber auch wissen, wie wir einen Wert innerhalb einer Prozedur verarbeiten können, um anschließend das Ergebnis in unserem eigentlichen Hauptprogramm weiterzuverwenden.

Hierfür sind die bereits angesprochenen Funktionen nützlich, die wie folgt definiert werden:

```
Procedure.Type Name(<Variable1>[, <Variable2>, ...])
...
  ProcedureReturn Wert
EndProcedure
```

Der Unterschied gegenüber einer normalen Prozedur besteht darin, dass hier nach dem Schlüsselwort *Procedure* und einem nachfolgenden Punkt (.) der Typ des Rückgabewerts angegeben werden muss, also '.l' für eine Long-Variable, '.w' für eine Word-Variable usw. An einem beliebigen Punkt innerhalb der Prozedur muss schließlich noch das Schlüsselwort *ProcedureReturn* mit einer nachfolgenden Variablen angegeben werden, womit der Inhalt dieser Variablen an das aufrufende Programm zurückgegeben wird.

Beispiel

```
Procedure.l Maximum(Wert1.l, Wert2.l)
  If Wert1 > Wert2
    Ergebnis.l = Wert1
  Else
    Ergebnis = Wert2
  EndIf

  ProcedureReturn Ergebnis
EndProcedure

Wert.l = Maximum(15, 30)
PrintN(Str(Wert))
```

Das Schlüsselwort *ProcedureReturn* kann mehrfach (an verschiedenen

Stellen) innerhalb der Prozedur verwendet werden, da mit der Rückgabe des Wertes die Prozedur sofort beendet wird. Nachfolgendes Beispiel verdeutlicht dies anhand einer If-Bedingung:

```
Procedure.s Deutschland(Wo.s)
  If Wo.s = "Berlin"
    ProcedureReturn "Hauptstadt"
    ; die Prozedur endet hier
  Else
    ProcedureReturn "Anderswo"
    ; die Prozedur endet hier
  EndIf
EndProcedure
MessageRequester ("Information",Deutschland("Berlin"),0)
```

Prozeduren können auch ineinander verschachtelt werden, d.h. innerhalb einer Prozedur kann eine andere aufgerufen (nicht definiert!) werden.

Beispiel

```
Procedure DruckeWert(Wert1.l)
  a$=Str(Wert1) ; wandelt den aus der Pro-
zedur Max() erhaltenen
  PrintN(a$) ; Wert in einen String um
und gibt ihn aus
EndProcedure

Procedure Max(Wert1.l, Wert2.l)
  If Wert1 > Wert2
    Ergebnis.l = Wert1
  Else
    Ergebnis = Wert2
  EndIf
  DruckeWert(Ergebnis) ; ruft die bereits de-
finierte Prozedur auf
EndProcedure

Max(77,30)
```

10.3. Globale und lokale Variablen

Variablen (einschließlich der als Parameter übergebenen Werte) innerhalb einer Prozedur sind grundsätzlich unabhängig von den Variablen im Hauptprogramm, d.h. Variable 'a' innerhalb der Prozedur hat nichts mit der Variable 'a' des Hauptprogramms zu tun. Daher werden diese Variablen auch als *lokale Variablen* bezeichnet, welche nur während der Laufzeit der Prozedur existieren.

Sie können deshalb – ohne Kollisionen befürchten zu müssen – innerhalb der Prozedur beliebige Variablennamen verwenden, selbst solche, die bereits außerhalb deklariert wurden. Deshalb eignen sich die Proze-

duren auch hervorragend dafür, bestimmte Programmcodes zur Wiederverwendung in verschiedenen Applikationen zu archivieren.

Es gibt allerdings auch Situationen, in denen der Programmierer darauf angewiesen ist, bereits im Hauptprogramm deklarierte Variablen, Arrays etc. benutzen zu können. Dazu müssen die Variablen des Hauptprogramms mit der Prozedur „geteilt“ (englisch: „Shared“) werden.

Auch dafür bietet PureBasic mehrere Lösungen:

Global

Global deklariert Variablen als *globale Variablen*, d.h. auf sie kann auch innerhalb einer Prozedur zugegriffen und von dort auch verändert werden. Benutzt wird *Global* in der Regel am Anfang des eigentlichen Sourcecodes, damit die mit diesem Schlüsselwort deklarierten Variablen überall im Programm zur Verfügung stehen.

Benutzt wird *Global* wie folgt:

```
Global <Variable> [, <Variable>, ...]
```

Hinter *Global* geben Sie einfach die von Ihnen als globale Variablen gewünschten Namen mit oder ohne Typ-Zusatz an.

Beispiel

```
Global a.l, b.b, c, d
```

Shared

Shared erlaubt den Zugriff auf eine Variable innerhalb einer Prozedur, ohne dass diese extra als global deklariert wurde. Wenn Sie Zugriff auf ein Array benötigen, müssen Sie den Namen des Arrays ohne Klammern benutzen.

Benutzt wird *Shared* wie folgt:

```
Shared <Variable> [, <Variable>, ...]
```

Hinter *Shared* geben Sie einfach mit Namen und ggf. Typ-Zusatz die Variablen an, die Sie auch in Ihrer Prozedur verwenden und ggf. ändern möchten.

Beispiel

```
a.l = 10 ; deklariert die Variable 'a' vom Typ  
Long mit einem Wert 10  
Dim Zahlen.w(8) ; deklariert ein Array 'Zahlen'
```

```

Procedure Wechsel()
    Shared a          ; ermöglicht der Prozedur Zugriff auf
    die deklarierte Variable
    Shared Zahlen     ; ermöglicht der Prozedur Zugriff auf
    das deklarierte Array
    a = 20             ; weist der Variablen 'a' den neuen
    Wert 20 zu
    Zahlen(0) = 5      ; weist dem Element 0 des Arrays
    'Zahlen' den Wert 5 zu
EndProcedure

Wechsel()
PrintN(Str(a))        ; wird 20 ausgegeben, da die Variable
"geteilt" (shared) wurde.
PrintN(Str(Zahlen(0))) ; wird 5 ausgegeben, da das Array
"geteilt" (shared) wurde.

```

Protected

Protected könnte man mit „gesichert“ übersetzen und ist in anderen Programmiersprachen oftmals als ‚Local‘ bekannt. Damit definieren Sie innerhalb einer Prozedur eine neue Variable – im Hauptprogramm kann unter gleichem Namen bereits eine als global deklarierte Variable existieren – um diese nur innerhalb der Prozedur zu verwenden. Eine als „protected“ deklarierte Variable hat somit (innerhalb der betreffenden Prozedur) eine höhere Priorität, als eine globale Variable mit gleichem Namen.

Deklariert wird eine „gesicherte“ Variable wie folgt:

```
Protected <Variable> [, <Variable>, ...]
```

Hinter *Protected* geben Sie einfach mit Namen und ggf. Typ-Zusatz die Variablen an, die Sie innerhalb Ihrer Prozedur ausschließlich als lokale Variablen verwenden möchten.

Beispiel

```

Global a          ; a wird als global deklariert
a = 10             ; a bekommt den Wert 10 zugewiesen

Procedure Tausch()
    Protected a    ; a wird innerhalb der Prozedur als "gesichert"
    deklariert
    a = 20          ; und erhält den Wert 20; es erfolgt keine
    Überschneidung
EndProcedure      ; mit der globalen Variable 'a'

Tausch()           ; Aufruf der Prozedur
PrintN(Str(a))     ; gibt 10 aus, da die Variable innerhalb der

```

Prozedur gesichert war

10.4 Rekursivität

Es kann in manchen Situationen durchaus sinnvoll sein, dass sich eine Prozedur selbst aufruft. Diese Funktionsweise nennt man „Rekursivität“ bzw. rekursive Prozeduren. In PureBasic wird bei Prozeduren die Rekursivität voll unterstützt.

In einem solchen Fall ruft eine Prozedur während ihres Ablaufs sich selbst auf, wodurch eine zweite Version der Prozedur in den Speicher geladen und aktiviert wird. Nach Beendigung dieser zweiten Version der Prozedur kehrt die Kontrolle zur aufrufenden ersten Version der Prozedur zurück. Erst wenn auch diese beendet ist, kehrt die Kontrolle zum eigentlichen Hauptprogramm zurück. In der Regel wird jedoch die zweite Version während ihres Ablaufs die Prozedur erneut aufrufen, also eine dritte Version erzeugen, die ihrerseits eine vierte Version erzeugt, usw.

Um hier jetzt keine Endlosschleife und damit einen Speicherüberlauf zu provozieren, müssen Sie dafür Sorge tragen, dass die Prozedur an einem bestimmten Punkt auch ohne Neuaufruf beendet wird. Deswegen gibt es in rekursiven Prozeduren immer eine If-Anweisung, in der eine irgendwie geartete Endbedingung abgefragt wird.

Die Verwendung wollen wir uns an einem nachfolgenden Beispiel näher ansehen:

```
Procedure.l Fact(n.l)
  If n > 1
    n = Fact(n-1) * n          ; hier ruft sich die Prozedur
    selbst auf, jedoch nur    ; solange wie n > 1 ist (End-
  EndIf                      bedingung)
  ProcedureReturn n
EndProcedure

a = Fact(3)
PrintN(Str(a))
```

10.5 Deklaration mittels ‚Declare‘

Prozeduren müssen grundsätzlich erst definiert werden, bevor sie durch den Programmierer aufgerufen werden können. Daher stehen sie auch meist am Anfang eines Sourcecodes.

Manchmal ist es jedoch notwendig, dass eine Prozedur von einer anderen Prozedur aufgerufen wird, bevor sie überhaupt deklariert wurde. Da ist es sehr ärgerlich, wenn der Compiler dies mit 'Prozedur xxxx nicht gefunden' reklamiert.

Zur Lösung dieses Problems gibt es in PureBasic eine weitere Möglichkeit:

Mittels dem Schlüsselwort *Declare* wird zu Beginn nur der Kopf (englisch: „Header“) der Prozedur deklariert. Die tatsächliche Definition der Prozedur erfolgt erst später im Programmcode.

Verwendet wird *Declare* wie das Schlüsselwort *Procedure* auch, indem der Name, ein optionaler Typ bei Verwendung als Funktion sowie ein oder mehrere Parameter angegeben werden:

Declare[.<Typ>] Name(<Variable1>[,<Variable2>,...])

Hinweis: Die Deklaration des „Headers“ mittels *Declare* und die (spätere) tatsächliche Deklaration der Prozedur mittels *Procedure* müssen identisch sein.

Beispiel

```
Declare Minimum(Wert1, Wert2)

Procedure Operate(Wert)
    Minimum(10, 2)                ; Zu dieser Zeit ist
                                ; Maximum() nicht bekannt.
EndProcedure

Procedure Minimum(Wert1, Wert2)
    ProcedureReturn 0
EndProcedure
```

11. Windows-Programmierung

Nachdem wir uns nun die ganzen Grundlagen der Programmierung in PureBasic angesehen haben, wollen wir dies natürlich auch anwenden.

Neben den grundlegenden Anweisungen und Schlüsselwörtern zur Speicherung und Verarbeitung von Daten, Steuerung des Programmablaufs, mathematischen Berechnungen und Vergleichen (Operatoren) usw. bietet PureBasic noch über 450 weitere Befehle, die Ihnen bei der Realisierung all Ihrer Projekte behilflich sein sollen.

Die wichtigsten Befehle zur Programmierung von Windows-Applikationen wollen wir uns in diesem Kapitel näher ansehen.

11.1 Einführung zu Windows

Microsoft Windows® - nachfolgend einfach Windows genannt – ist sicherlich das am meisten verbreitete Betriebssystem. Allen Entwicklern, die sich für diese Plattform entscheiden, steht damit die wohl größte Softwareplattform der Welt offen. Mit PureBasic werden Sie bei der Entwicklung Ihrer Spiele und Anwendungen für Windows vorbildlich unterstützt.

Windows bietet eine große Palette eingebauter Funktionen, die dem Programmierer beim Erstellen seiner Applikationen behilflich sind. Diese Funktionen stehen – sofern sie nicht direkt in PureBasic als Befehl bzw. Funktion (siehe Befehlsreferenz im Kapitel 15) eingebaut sind – als Windows API - Funktion (siehe Kapitel 13) zur Verfügung.

Auf dem Windows Betriebssystem können viele Programme gleichzeitig ablaufen. Dabei teilen sie sich Speicher, Prozessorleistung und weitere Ressourcen, die von Windows automatisch verwaltet werden.

Eine graphische Windows-basierte Applikation kommuniziert mit dem Anwender über Fenster (damit sollte auch klar sein, woher der Name des Betriebssystems „Windows“ rührt), die zur Ausgabe von Informationen sowie zum Empfang von Anwendereingaben dienen. Jedes Fenster kann weitere abhängige „Child“-Fenster beinhalten, die jeweils ein Element der Benutzerschnittstelle – wie Schalter, Text-Boxen usw. – repräsentieren.

Ein Fenster teilt sich den Bildschirm mit anderen Fenstern, einschließlich derer von anderen Applikationen. Nur ein Fenster kann gleichzeitig Anwendereingaben empfangen. Der Anwender kann für seine Eingaben die Maus, die Tastatur oder jedes andere Eingabegerät benutzen, um mit einem Fenster und dessen übergeordneter Applikation zu kommunizieren.

Wie Sie sehen, kommt dem „Fenster“ unter Windows eine entscheidende Bedeutung zu. Deshalb ist es beim Starten einer Applikation einer der ersten Vorgänge, dass ein Fenster geöffnet wird.

11.2 Allgemeiner Programmaufbau

Wenn der Anwender eine Applikation startet, geht in der Regel stets ein ähnlicher Ablauf von Prozessen vonstatten. Dieser Ablauf könnte wie folgt aussehen:

Der Anwender startet das Programm

Die Applikation initialisiert sich selbst (lädt erforderliche Daten, Benutzereinstellungen etc.)

Die Applikation zeigt ihre Benutzeroberfläche und wartet auf Eingaben des Anwenders

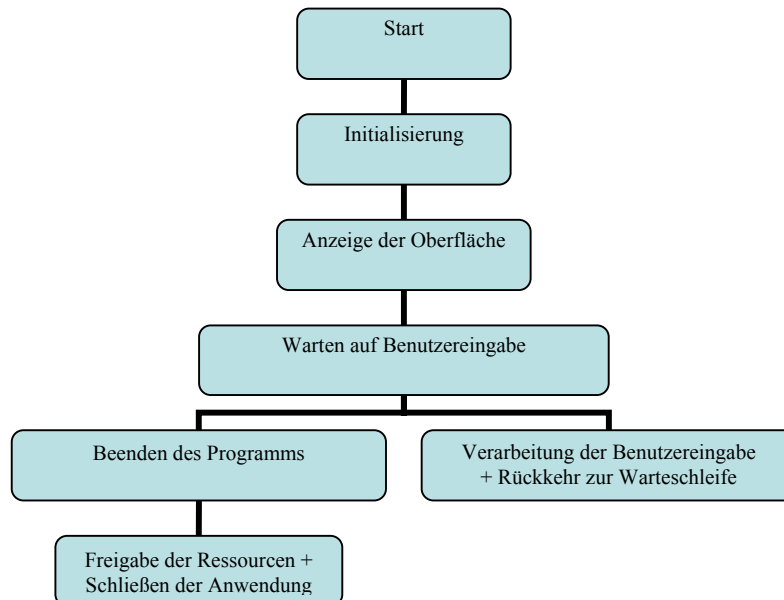
Der Anwender macht eine Eingabe (drückt einen Schalter, wählt einen Menüpunkt aus etc.)

Die Applikation reagiert auf die Anwendereingabe

Dies wiederholt sich solange, bis der Anwender die Applikation schließt.

Beim Schließen der Anwendung werden die Ressourcen freigegeben (speichern von ungesicherten Daten, freigeben von Speicher etc.) und das Programm selbst beendet.

Der dazugehörige Ablaufplan für eine Anwendung mit grafischer Benutzeroberfläche sieht somit wie folgt aus:



Im Grunde handelt es sich bei diesem Ablaufplan um nichts weiter, als um die detaillierter aufgeschlüsselten Grundfunktionen eines jeden Programms:

- Eingabe – Eingabe von Daten/Informationen in das Programm
- Speichern – Speichern der erhaltenen Daten/Informationen
- Verarbeitung – Verarbeiten der gespeicherten Daten/Informationen
- Ausgabe – Daten/Informationen an den Benutzer ausgeben

Diesen grundlegenden Funktionen wollen wir uns in diesem Kapitel widmen, damit Sie die grundlegende Programmierung in PureBasic kennen und verstehen lernen, diese dann auf eigene Projekte anwenden und so nach und nach alle wichtigen Funktionen dieser wundervollen Programmiersprache ausnutzen können.

II.3 Fenster

Wie schon verdeutlicht, kommt kein Windows-Programm mit einer graphischen Benutzeroberfläche (das Gegenteil sind Konsolen-Programme) ohne Fenster aus.

Fenster werden in PureBasic mit dem Befehl: *OpenWindow* geöffnet. Da

im Normalfall ein Programm ohne geöffnetes Fenster nutzlos ist, werden wir dies auch überprüfen und bei einem Fehler das Programm ggf. beenden. Andernfalls verharren wir in einer Warteschleife, und warten auf das Schließen des Fensters durch den Anwender.
Der entsprechende Sourcecode könnte wie folgt aussehen:

```

If OpenWindow(0, 100, 150, 250, 200, #PB_Window_SystemMenu,
"PureBasic Fenster")
    ; Das Fenster wurde erfolgreich geöffnet
    Repeat
        EventID.1 = WaitWindowEvent()
        If EventID = #PB_Event_CloseWindow ; der Schließ-Knopf
wurde gedrückt...
            Ende = 1
        EndIf
    Until Ende = 1
Else
    ; Fehler beim Öffnen des Fensters
    End
EndIf
End

```

Das Öffnen des Fensters erfolgt wie gesagt mit dem Befehl `OpenWindow()`, der in Klammern folgende Parameter übergeben bekommt:

0	Fensternummer - der sogenannte Identifier (ID), eine interne PureBasic-Variable, welche zusammen mit anderen PureBasic-Funktionen benötigt wird
100	Position des linken Fensterrandes in xx Pixel Abstand vom linken Bildschirmrand
150	Position des oberen Fensterrandes in xx Pixel Abstand vom oberen Bildschirmrand
250	Innere Breite des Fensters in Pixel (der Rahmen des Fensters wird automatisch hinzu addiert)
200	Innere Höhe des Fensters in Pixel (der Rahmen des Fensters wird automatisch hinzu addiert)
#PB_Window_SystemMenu	PureBasic-Konstante, durch deren Angabe das Fenster mit dem Standardmenü eines jeden Windows-Fensters geöffnet wird (u.ä. Menü und Schließknopf).

“PureBasic Fenster“	<p>Dieser Parameter kann auch mehrere – mittels dem ‚ ‘ Operator kombinierte – Konstanten enthalten, z.B. #PB_Window_SystemMenu #PB_Window_SizeGadget.</p> <p>Der Titel des Fensters als String. Dieser kann auch als z.B. ‚Titel\$‘ bei einem bereits vorher definierten Titel-String übergeben werden.</p>
---------------------	--

Der Befehl zum Öffnen des Fensters wird im Rahmen der If-Bedingung aufgerufen. Ergibt der Rückgabewert falsch, ist die Bedingung nicht erfüllt und es wird der Else-Abschnitt der Bedingung ausgeführt und somit das Programm beendet.

Im anderen Fall – also bei erfüllter If-Bedingung – verbleiben wir in diesem Code-Abschnitt und benutzen einen weiteren Befehl der Windows-Bibliothek: *WaitWindowEvent()*. Wir haben ja bereits gelernt, dass auf dem Windows®-Betriebssystem die gesamte Kommunikation über Ereignisse abgewickelt wird. Mit diesem Befehl warten wir, bis ein Ereignis (Anklicken eines Schalters, des Schließgadgets etc.) innerhalb unseres Fensters auftritt. Beim Auftreten eines Ereignisses wird dieses in der Variable *EventID.1* gespeichert, welche wir nachfolgend auf die Art des Ereignisses überprüfen. In unserem Code-Beispiel überprüfen wir durch den Vergleich mit der Konstante *#PB_Event_CloseWindow*, ob das Schließknopf am Fenster gedrückt wurde.

Wenn ja, wird die Variable ‚Ende‘ auf 1 gesetzt. Dadurch wird die Bedingung der ‚Repeat‘-Schleife erfüllt und das Programm schließlich beendet.

Mit weiteren Befehlen aus der Windows-Befehlsbibliothek können Sie ein Fenster in der Größe ändern, es verschieben oder verstecken sowie verschiedene Tastatur- und Mausabfragen realisieren. Schauen Sie hierzu in die Befehlsdokumentation (Kapitel 15) und probieren Sie doch einfach die mitgelieferten Beispielcodes aus.

Hinweis

Der Rückgabewert der *OpenWindow* Funktion enthält das sogenannte Window-Handle (Variable ‚hWnd‘ in der MS-Dokumentation), das oftmals in Verbindung mit WinAPI-Funktionen benötigt wird.

11.4. Gadgets

Mit Gadgets – zu deutsch „Schalter“ – werden die graphischen Benutzeroberflächen von Windows-Programmen gestaltet. Sie sind neben den

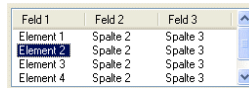
Menüs elementarer Bestandteil eines jeden Programms, um mit dem Anwender zu kommunizieren.

Es gibt eine ganze Palette möglicher Gadget-Typen, die in PureBasic zur Verfügung stehen:

<i>Befehl</i>	<i>Screenshot</i>	<i>Art des Gadgets</i>
<i>ButtonGadget</i>		Ein Schalter, der angeklickt werden kann. Damit kann nachfolgend eine Aktion ausgelöst werden.
<i>ButtonImageGadget</i>		Wie der „normale“ Schalter, jedoch anstelle einer Beschriftung mit einem Bild im Display. Die Größe des Schalters wird dabei automatisch an die Bildgröße angepasst.
<i>CheckBoxGadget</i>		Das sogenannte Häkchen-Gadget. Abhängig davon, ob das Häkchen gesetzt ist oder nicht, kann später im Programm die eine oder die andere Aktion ausgelöst werden. Dient daher oft
<i>ComboBoxGadget</i>		Ein Auswahl-Gadget, welches dem Anwender ermöglicht, eines

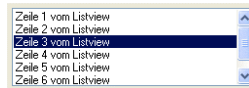
		der enthaltenen Elemente auszuwählen.
<i>Frame3DGadget</i>		Ein sogenannter Rahmen. Ein Gadget, dass selbst kein Auslöser eines Ereignisses ist, sondern zur „Verzierung“ einer Benutzeroberfläche dient. Damit sollt – in dem es mehrere andere Gadgets einschließt – eine bessere Übersicht im Fenster erreicht werden.
<i>ImageGadget</i>		Ein Gadget zum Anzeigen eines Bildes. Dient lediglich zum Gestalten einer Fensteroberfläche. Damit kann kein Ereignis ausgelöst werden.
<i>IPAddressGadget</i>		Ermöglicht die einfache Eingabe einer korrekten IP-Adresse. Dient zur Unterstützung der Netzwerk-Funktionalität von PureBasic.

ListIconGadget



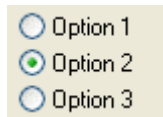
Erstellt ein List-Icon, mit dem es einfach möglich ist, Informationen nach Spalten sortiert auszugeben.

ListViewGadget



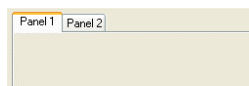
Erstellt eine Auswahlliste, die beliebig viele Informationen zeilenweise anzeigt und diese durch den Anwender auswählen lässt.

OptionGadget



Ein Optionen-Gadget, mit dessen Hilfe der Anwender eine der angebotenen Optionen auswählen kann.

PanelGadget



Erstellt eine oder mehrere Schalttafeln (Karteikartenreiter), durch deren Einsatz Sie Ihre Benutzeroberflächen übersichtlicher gestalten und vor allem mehr Informationen und Gadgets darstellen können, als eigentlich innerhalb der Fenstergröße möglich.

ProgressBarGadget

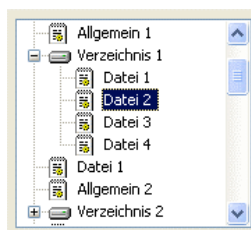


Stellt einen Fortschrittsbalken dar,

		mit dem Sie den Anwender Ihres Programms bei zeitintensiven Operationen über den Programmfortschritt informieren können.
<i>SpinGadget</i>		Mit dem SpinGadget können Sie einen Zahlenwert innerhalb eines vorgegebenen Bereichs durch die Auf-/Ab-Schalter einstellen.
<i>StringGadget</i>		Mit dem StringGadget können Sie vom Anwender Ihres Programms die Eingabe eines Textes, eines Zahlenwertes etc. fordern.
<i>TextGadget</i>		Mit dem TextGadget werden Informationen auf der Programmoberfläche ausgegeben. Im Bild links sehen Sie einmal die rahmenlose Variante und einmal die mit vertieftem Rand.
<i>TrackBarGadget</i>		Auch das TrackBarGadget dient als eine Art Schie-

beregler, mit dem der Anwender innerhalb eines vorgegebenen Wertebereichs eine Einstellung vornehmen kann.

TreeGadget



Mit dem TreeGadget stellen Sie einfach eine Art Baumstruktur dar. Sinnvoll für die Darstellung z.B. eines Verzeichnisbaums im Explorer-Stil oder jede andere Möglichkeit, in der Sie Daten strukturiert anzeigen und auswählen lassen möchten.

WebGadget



Das WebGadget ist die jüngste Errungenschaft von PureBasic. Damit können Sie die komplette Funktionalität des InternetExplorers v4+ in Ihren Programmen einbinden.

Für detaillierte Erklärungen zum Einsatz der einzelnen Gadget-Typen schauen Sie bitte in die Befehlsreferenz im Kapitel 15 unter der Rubrik „Gadget“.

Die einzelnen Schalter werden in PureBasic innerhalb einer oder mehrerer Speicherbereiche verwaltet. Diese nennen sich Gadget-Liste und

werden mit folgenden Befehlen verwaltet: *CreateGadgetList*, *UseGadgetList*, *ClearGadgetItemList*.

Bevor die einzelnen Gadgets definiert werden können, muss zuerst ein Fenster und anschließend eine Gadgetliste initialisiert werden, welches nach folgendem Schema realisiert werden sollte:

```
If OpenWindow( ... )
    If CreateGadgetList(WindowID()) ; ermittelt die
    eindeutige WindowID und erstellt ; auf diesem Fenster die
    neue Gadgetliste
        ButtonGadget (1, 20,#y+ 20, 80, 24, "Schalter")
        CheckBoxGadget (3, 20,#y+160,120, 24, "Markierungsschal-
        ter")
        ...
    EndIf
EndIf
```

Zum Verarbeiten der vom Anwender übermittelten Informationen (zum Beispiel eingegebener Text, ausgewähltes Element in einem Auswahl-Gadget etc.) gibt es PureBasic weitere Befehle, mit denen diese Informationen ausgelesen werden können:

GadgetX, *GadgetY*, *GadgetWidth*, *GadgetHeight* – damit kann die Position (x, y) sowie Breite und Höhe eines Gadgets ausgelesen werden

GetGadgetItemState, *GetGadgetItemText*, *GetGadgetState*, *GetGadgetText* – damit wird der Status oder der Textinhalt eines Gadgets ermittelt, z.B. der gerade ausgewählte Eintrag in einem OptionGadget, in einem ListView usw.

CountGadgetItems – ermittelt die Anzahl an Einträgen in einem Auswahl-Gadget, wie ListView, Listicon etc.

GadgetID – damit wird die eindeutige ID (Identifizier) eines Gadgets innerhalb des Windows-Systems ermittelt

Inhalt und Aussehen eines Gadgets müssen jedoch auch nicht von Programmstart bis Programmende gleich bleiben, deshalb gibt es auch hierfür verschiedene Befehle zum Ändern des Gadgetinhalts:

ActivateGadget – aktiviert bzw. setzt den Focus auf ein angegebenes Gadget

DisableGadget – deaktiviert ein Gadget, es kann nicht mehr angeklickt werden.

HideGadget – versteckt ein Gadget, es verschwindet von der Fensteroberfläche

FreeGadget – entfernt ein Gadget von der Fensteroberfläche und aus der Gadgetliste (Ressourcen werden freigegeben)

ResizeGadget – ändert Position und/oder Größe eines Gadgets

SetGadgetFont – legt einen neuen Zeichensatz für alle nachfolgend erstellten Gadgets mit Textinhalt fest

SetGadgetItemState, *SetGadgetItemText*, *SetGadgetState*, *SetGadgetText* – ändert den Status (z.B. den ausgewählten Eintrag eines ListIcons) bzw. den Textinhalt eines Gadgets

AddGadgetItem, *RemoveGadgetItem* – fügt einen Eintrag hinzu bzw. entfernt einen Eintrag aus einem Auswahl-Gadget

Zu allen Gadgets lässt sich in PureBasic auch ein Hilfstext hinzufügen, der in einem kleinen gelben Fenster erscheint, wenn der Anwender mit dem Mauszeiger eine Weile über einem Gadget verharret. Definiert wird ein solcher Hilfstext mit dem Befehl *GadgetToolTip*.

Es gibt noch einige weitere Befehle der Gadget - Befehlsbibliothek, die nur in Zusammenhang mit einzelnen Gadgettypen nützlich sind:

AddGadgetColumn, *ChangeListItemIconGadgetDisplay* – fügt eine weitere Spalte zu einem ListIcon hinzu bzw. ändert dessen Aussehen

ClosePanelGadget – schließt die Definition eines PanelGadgets ab

OpenTreeGadgetNode, *CloseTreeGadgetNode* – erstellt einen neuen „Knoten“ bei einem TreeGadget bzw. schließt dessen Definition ab.

11.5 Menüs und Werkzeug-Leisten

Wie bereits erwähnt sind die Menüs ein weiterer wichtiger Bestandteil von graphischen Benutzeroberflächen. Innerhalb eines Anwenderprogramms dienen sie in erster Linie dazu, dem Anwender die schnelle Auswahl einer gewünschten Funktion zu ermöglichen. Deshalb finden Sie in den meisten Programmen solche Aktionen wie „Öffnen“, „Speichern“, „Drucken“ etc. in der Menüleiste am oberen Fensterrand.

Standard-Menüs

Diese Menüs lassen sich mit PureBasic ganz einfach erstellen. Zuerst reservieren Sie die Ressourcen für eine Menü-Liste mit dem Befehl *CreateMenu*.

Innerhalb dieser Liste erstellen Sie dann die einzelnen Menüeinträge:

MenuTitle – erstellt die Überschrift für ein Menü

MenuItem – erstellt einen anwählbaren Menüeintrag

MenuBar – erstellt einen Abgrenzungsbalken zwischen zwei Menüeinträgen

OpenSubMenu, *CloseSubMenu* – leitet die Definition eines Untermenüs ein bzw. schließt diese ab

Nachdem Sie ein Menü erstellt haben, können Sie auf die Menüeinträge auch noch weitere Operationen anwenden:

DisableMenuItem – deaktiviert (und aktiviert auch wieder) einen Menüpunkt

SetMenuItemState, *GetMenuItemState* – versieht einen Menüeintrag mit einem Häkchen bzw. ermittelt den Status eines solchen Menüeintrags

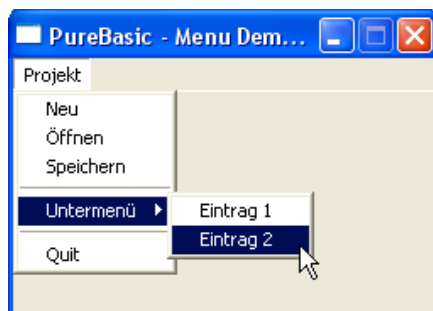
MenuHeight – ermittelt die Höhe der Titelzeile eines Menüs

Die Definition eines einfachen Standardmenüs könnte wie folgt aussehen:

```

If OpenWindow( ... )           ; in Klammern die
                                ; erforderlichen
                                ; Fenster-Parameter
                                If CreateMenu(0, WindowID()) ; ermittelt die eindeutige
                                ; WindowID und erstellt
                                ; auf diesem Fenster die
                                ; neue Menüliste

    MenuTitle("Projekt")
    MenuItem(0, "Neu")
    MenuItem(1, "Öffnen")
    MenuItem(2, "Speichern")
    MenuBar()
    OpenSubMenu("Untermenü")
        MenuItem(3, "Eintrag 1")
        MenuItem(4, "Eintrag 2")
    CloseSubMenu()
    MenuBar()
    MenuItem(5, "Quit")
EndIf
EndIf
    
```



Popup-Menüs

PureBasic unterstützt neben diesen „Standard-Menüs“ auch die sogenannten „Popup-Menüs“, die sich nach einem Klick mit der rechten Maustaste an der aktuellen Position des Mauszeigers öffnen.

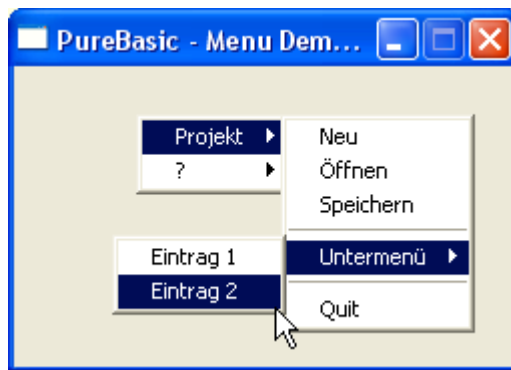
Erstellt und angezeigt wird ein solches Menü mit diesen beiden Befehlen: *CreatePopupMenu* und *DisplayPopupMenu*. Die übrigen Befehle kommen wie beim Standard-Menü zum Einsatz.

Die Definition könnte also so aussehen:

```
If OpenWindow( ... )           ; in Klammern die
                                ; erforderlichen
                                ; Fenster-Parameter

    If CreatePopupMenu(0, WindowID())
                                ; ermittelt die eindeutige
                                ; WindowID und
                                ; erstellt auf diesem
                                ; Fenster die neue
                                ; Menüliste

        MenuTitle("Projekt")
        MenuItem(0, "Neu")
        MenuItem(1, "Öffnen")
        MenuItem(2, "Speichern")
        MenuBar()
        OpenSubMenu("Untermenü")
            MenuItem(3, "Eintrag 1")
            MenuItem(4, "Eintrag 2")
        CloseSubMenu()
        MenuBar()
        MenuItem(5, "Quit")
        DisplayPopupMenu(0, WindowID())
    EndIf
EndIf
```



Hinweis

Entscheiden Sie sich immer für eine Menüart in Ihrem Programm, da

normale Menüs und Popup-Menüs dieselben Ereigniswerte zurückliefern und es bei gleichzeitiger Verwendung zu Überschneidungen kommen kann.

Werkzengleisten (ToolBars)

Die dritte Möglichkeit, die wir an dieser Stelle besprechen wollen, sind die sogenannten Werkzeug-Leisten (englisch: ToolBar) am oberen Fensterrand.

Bevor wir einzelne „Tools“ (Werkzeuge) reservieren können, müssen wir erst wieder die Ressourcen dafür reservieren. Dies geschieht mit dem Befehl *CreateToolBar*.

Auf dieser Werkzengleiste können Sie nun die einzelnen Einträge definieren:

ToolBarStandardButton

Definiert ein durch Windows zur Verfügung gestelltes Standard-Piktogramm. Folgende Konstanten können als zweiter Parameter angegeben werden:

<i>PureBasic-Konstante</i>	<i>Piktogramm</i>	<i>Bedeutung</i>
#PB_ToolBarIcon_New		Neu
#PB_ToolBarIcon_Open		Öffnen
#PB_ToolBarIcon_Save		Speichern
#PB_ToolBarIcon_Print		Drucken
#PB_ToolBarIcon_Find		Suchen
#PB_ToolBarIcon_Replace		Ersetzen
#PB_ToolBarIcon_Cut		Ausschneiden
#PB_ToolBarIcon_Copy		Kopieren

#PB_ToolBarIcon_Paste		Einfügen
#PB_ToolBarIcon_Undo		Rückgängig
#PB_ToolBarIcon_Redo		Wiederholen
#PB_ToolBarIcon_Delete		Löschen
#PB_ToolBarIcon_Properties		Eigenschaften
#PB_ToolBarIcon_Help		Hilfe

ToolBarImageButton

Definiert ein eigenes Piktogramm im Icon-Format (.ico), welches zuvor mittels LoadImage geladen wurde.

ToolBarSeparator

Definiert einen Abgrenzungsbalken zwischen zwei Piktogrammen, der zur optischen Abgrenzung thematisch unterschiedlicher Werkzeuge dient.

ToolBarToolTip

Fügt einem Piktogramm einen Hilfstext hinzu, der in einem kleinen gelben Fenster erscheint, wenn der Anwender eine Weile mit dem Mauszeiger über einem Piktogramm verharrt.
Bereits angezeigte ToolBar-Piktogramme lassen sich auch verändern.
Dies geschieht mit folgenden Befehlen:

DisableToolBarButton

Deaktiviert (und aktiviert ggf. auch wieder) ein Piktogramm.

FreeToolBar

Entfernt eine ToolBar-Leiste vom zugehörigen Fenster und gibt deren Ressourcen frei.

Die Definition einer Werkzeugleiste (ToolBar) können Sie wie folgt vornehmen:

```
If OpenWindow( ... )           ; in Klammern die
                                ; erforderlichen
                                ; Fenster-Parameter
    LoadImage(0, "Gfx\New.ico") ; lädt ein eigenes
                                ; Piktogramm im Icon-Format
    LoadImage(1, "Gfx\Save.ico") ; lädt ein weiteres
                                ; Piktogramm im Icon-Format
    If CreateToolBar(0, WindowID()); ermittelt die eindeutige
                                ; WindowID und erstellt auf
                                ; diesem Fenster die neue
                                ; ToolBar
        ToolBarStandardButton(0, #PB_ToolBarIcon_New)
        ToolBarToolTip(0, "Neu") ; fügt einen Hilfstext
                                ; (ToolTip) „Neu“ hinzu
        ToolBarStandardButton(1, #PB_ToolBarIcon_Open)
        ToolBarToolTip(1, "Öffnen") ; fügt einen Hilfstext
                                ; „Öffnen“ hinzu
        ToolBarStandardButton(2, #PB_ToolBarIcon_Save)
        ToolBarToolTip(2, "Speichern"); fügt einen Hilfstext
                                ; „Speichern“ hinzu
        ToolBarSeparator()       ; fügt einen Agrenzungs
                                ; balken ein
        ToolBarImageButton(3, UseImage(0))
                                ; verwendet das selbst
                                ; geladene Icon 0
        ToolBarImageButton(4, UseImage(1))
                                ; verwendet das selbst
                                ; geladene Icon 1
    EndIf
```

Wichtig zu wissen ist bei den Werkzeug-Leisten, dass diese nach dem Anklicken die gleichen Ereignisse wie Menüs zurückliefern. Dies macht deshalb Sinn, da die Piktogramme der Werkzeugleiste oftmals als „Shortcuts“ für die häufig benötigte Menüpunkte dienen. Sie kennen sicher aus vielen Ihrer Anwendungsprogramme die Piktogramme



wodurch die Menü-Funktionen „Neu“, „Öffnen“, „Speichern“, „Drucken“ und „Suchen“ mit einem Mausklick erreichbar sind.

Praktischerweise verwenden Sie nun bei der Definition der o. g. Menüpunkte die gleichen ID's (erster Parameter bei den MenuItem und ToolBar Befehlen) wie für die dazugehörigen Werkzeug-Piktogramme. Da-

durch können Sie später mit nur EINER Abfrage für beide die gewünschte Aktion einleiten.

Beispiel

```
; der Menüeintrag erhält die ID 0
MenuItem(0, "Neu")
; genauso das Werkzeug die ID 0
ToolBarStandardButton(0, #PB_ToolBarIcon_New)
```

11.6 Requester

Die Requester sind vorgefertigte Dialog-Fenster, über die ein Programm mit dem Anwender kommunizieren kann. Die häufigste Form, der MessageRequester (Nachrichten-Dialog), haben wir beim „Hallo Welt“ – Beispiel und auch in einigen Beispiel-Sourcecodes bereits kennen gelernt.

Allen Requestertypen gemein ist die Tatsache, dass diese das aufrufende Programmfenster blockieren, dort also bis zum Schließen des Requesters keine Eingaben erfolgen können.

Mit dem MessageRequester können einfache Dialogfenster eingeblendet werden, die den Anwender über ein Ereignis informieren, diesem eine Frage stellen etc.

Der Syntax für den Aufruf eines „Nachrichten“-Requesters lautet wie folgt:

```
Ergebnis = MessageRequester(Titel$, Text$, Flag)
```

Titel\$ definiert einen String, der den Text für den Fenstertitel beinhaltet
Text\$ definiert ebenfalls einen String, der den anzuzeigenden Text (Nachricht, Fragestellung etc.) innerhalb des Fensters definiert. Dieser Text kann sich auch über mehrere Zeilen erstrecken, fügen Sie nach jeder Zeile einfach einen Zeilenumbruch mit dem ASCII-Zeichen 10 (oder auch 13) ein.

Beispiel

```
Text$ = "Zeile 1"+Chr(10)+"Zeile 2"+Chr(10)+"Zeile 3"
```

Flag definiert schließlich die Art des Requesters, d.h. ob dieser lediglich eine Nachricht und einen „OK“-Schalter beinhaltet oder ob er ein Information/Fragestellung mit „Ja“ / „Nein“ / „Abbrechen“ und weiteren Schaltertypen beinhaltet.

Nachfolgend eine Tabelle mit den in PureBasic möglichen Flag-Werten:

<i>Flag</i>	<i>PureBasic-Konstante</i>	<i>Dargestellte Schalter</i>
-------------	----------------------------	------------------------------

0	#PB_MessageRequester_Ok	„OK“
1		„OK“ und „Abbrechen“
2		„Abbrechen“, „Wiederholen“ und „Ignorieren“
3	#PB_MessageRequester_YesNoCancel	„Ja“, „Nein“ und „Abbrechen“
4	#PB_MessageRequester_YesNo	„Ja“ und „Nein“
5		„Wiederholen“ und „Abbrechen“
6		„Abbrechen“, „Wiederholen“ und „Weiter“

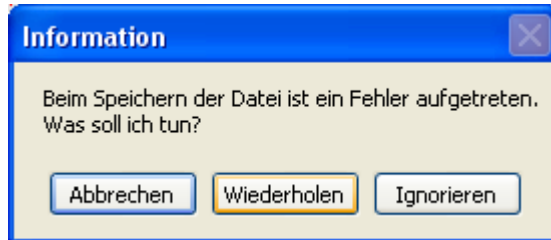
Sie können also einfach die Flag-Werte 0 bis 6 benutzen, eigene Konstanten mit diesen Werten definieren oder – sofern vorhanden – die bereits vordefinierte PureBasic-Konstante benutzen.

Beispiel

Beim Speichern einer Datei ist ein Fehler aufgetreten. Sie informieren darüber den Anwender mit dem MessageRequester – Flag 2.

Erreicht wird die Ausgabe dieses Requesters mit folgendem Programmcode (im Editor auf einer Zeile):

```
MessageRequester("Information", "Beim Speichern der Datei  
ist ein Fehler aufgetreten."+Chr(10)+"Was soll ich tun?", 2)
```



Ergebnis enthält den Rückgabewert des Requesters, nachdem der Anwender einen der angegebenen Schalter angeklickt hat. Nachfolgend eine Tabelle mit den Requestertypen (Flag-Wert) und möglichen Rückgabewerten:

Flag	Schalter 1	Rückgabewert	Schalter 2	Rückgabewert	Schalter 3	Rückgabewert
0	OK	1				
1	OK	1	Abbrechen	2		
2	Abbrechen	2	Wiederholen	4	Ignorieren	5
3	Ja	6	Nein	7	Abbrechen	2
4	Ja	6	Nein	7		
5	Wiederholen	4	Abbrechen	2		
6	Abbrechen	2	Wiederholen	10	Weiter	11

Natürlich stehen in PureBasic auch noch weitere standardmäßige Systemrequester zur Verfügung, die da wären:

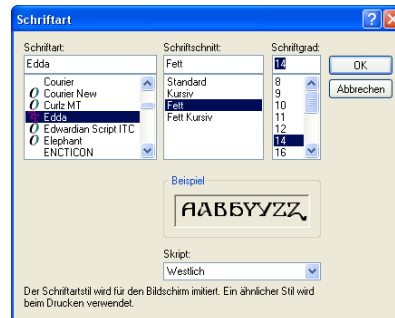
ColorRequester

Der „Farbauswahl“-Requester, mit dem Sie eine der Systemfarben ermitteln können.



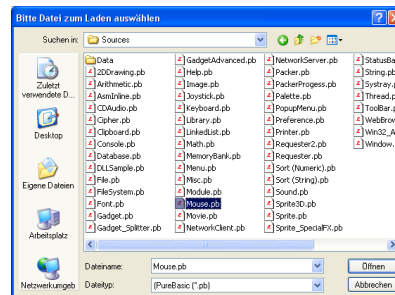
FontRequester

Der „Zeichensatz“-Requester, mit dem Sie Typ und Größe eines vorhandenen Zeichensatzes auswählen können.



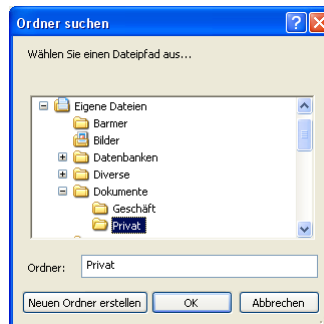
OpenFileRequester

Der Systemrequester zum Auswählen einer zu „ladenden Datei“.



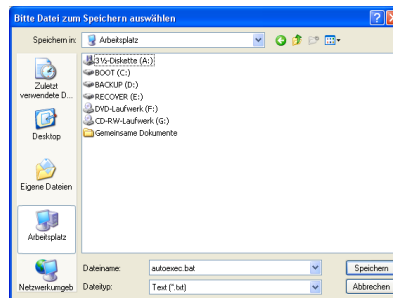
PathRequester

Der Systemrequester zum Auswählen eines Dateipfades.



SaveFileRequester

Der Systemrequester zum Auswählen einer zu „speichernden Datei“.



Alle diese Befehle sind in der Befehlsreferenz dokumentiert und können auch in dem Beispiel-Sourcecode „GadgetDEMO.pb“ in Aktion betrachtet werden.

11.7 Tastatur-Abfrage in Windows-Applikationen

Für schnellen Zugriff auf gewünschte Funktionen bietet es sich an, Tastatur-Kürzel zu definieren. Dies geschieht mittels des Befehls `AddKeyboardShortcut()`.

Beispiel

```
AddKeyboardShortcut(0, #PB_Shortcut_Control | _J
#PB_Shortcut_Z, 16) ; STRG + Z
```

Bei der Definition dieses Tastaturkürzels wird einfach ein Menüereignis (hier: 16) festgelegt, dass beim Drücken der angegebenen Tasten (hier: Strg+Z) ausgelöst wird.

Falls Sie während des Programmablauf das Aussehen bzw. die Funktion einzelner Schalter oder Menüpunkte ändern, können Sie zugeordnete

Tastaturkürzel auch wieder entfernen. Verwenden Sie hierzu den Befehl *RemoveKeyboardShortcut()*.

11.8 Abfrage von Windowseignissen (Maus, Tastatur, Gadgets, Menüs)

Die schönsten Benutzeroberflächen sind nichts wert, wenn das Programm nicht auf die vom Anwender getätigten Eingaben reagiert. Deshalb ist das Auswerten der aufgetretenen Ereignisse ein unerlässlicher Bestandteil eines jeden Programms.

Zuerst einmal benötigen wir eine Warteschleife, die auf das Auftreten eines Windows-Ereignisses (Anklicken eines Gadgets, Auswahl eines Menüpunktes etc.) wartet. Diese Schleife wird idealerweise folgendermaßen realisiert:

WaitWindowEvent()

Hält den Programmablauf bis zum Auftreten eines Ereignisses an

WindowEvent()

Überprüft, ob gerade ein Ereignis im Programmfenster stattfand, hält jedoch nicht den Programmablauf an

Der Befehl *WaitWindowEvent()* ist dabei jedoch eindeutig zu bevorzugen, da *WindowEvent()* nur mit entsprechenden Kenntnissen eingesetzt werden sollte, um nicht unwissentlich eine Menge Rechenzeit zu verschwenden.

Für das Auswerten und Überprüfen der aufgetretenen Ereignisse stehen folgende Befehle zur Verfügung:

EventType()

Ermittelt den Typ des Ereignisses, z.B. Klick mit der linken Maustaste

EventWindowID()

Ermittelt das Programmfenster, in dem das Ereignis stattfand

EventGadgetID()

Ermittelt den gedrückten Schalter, der das Ereignis auslöste

EventMenuID()

ermittelt den ausgewählten Menü-Eintrag, der das Ereignis auslöste. Dies gilt auch für das Anklicken eines ToolBar-Werkzeugs oder dem Drücken eines definierten Tastaturkürzels, da auch diese Menüereignisse zurückliefern.

Nachfolgend noch ein Beispiel für eine Warteschleife, die auf ein Ereignis wartet und dieses schließlich auswertet:

```
Repeat
    EventID.1 = WaitWindowEvent()

    ; Hier folgt jetzt die Überprüfung auf aufgetretene
    ; Ereignisse bei den Gadgets...
If EventID = #PB_EventGadget
        Select EventGadgetID()          ; Abfrage, bei welchem Gadget
                                        ; ein Ereignis auftrat

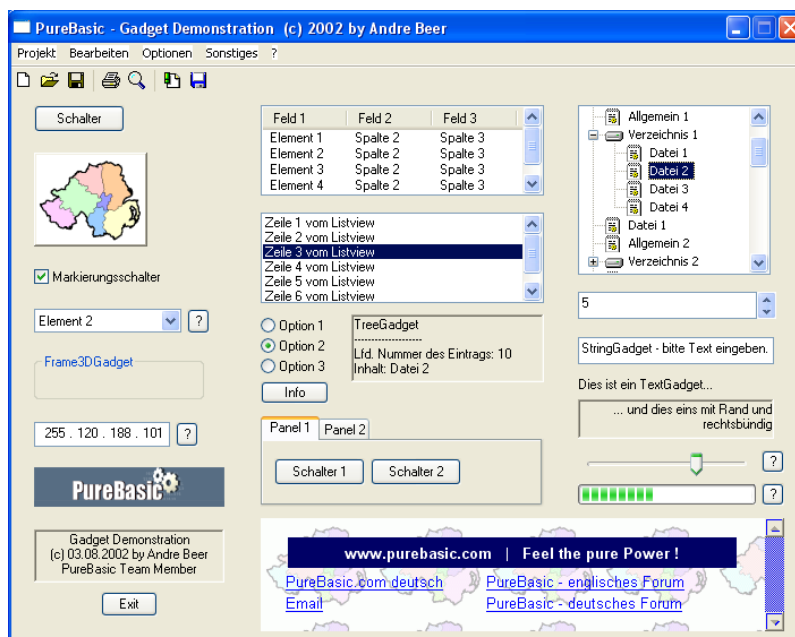
            Case 1      ; Gadget 1
                ...
            Case 2      ; Gadget 2
                ...
        EndSelect
EndIf

    ; Überprüfen der Menü-, Toolbar- und Tastatur-Ereignisse
If EventID = #PB_EventMenu
        Select EventMenuID()
            Case 1      ; Menü 1 (kann auch ein ToolBar-Werkzeug
                        ; oder Tastaturkürzel sein)
                ...
            Case 2      ; Menü 2
                ...
        EndSelect
EndIf
Until EventID = #PB_EventCloseWindow ; beendet die
                                        ; Schleife beim
                                        ; Anklicken des
                                        ; Schließgadget
```

11.9 Erstellung einer komplexen Programmoberfläche

Das Beispiel „*GadgetDEMO.pb*“, welches Sie auf der mitgelieferten Programm-CDRom finden, veranschaulicht die Verwendung von Fens-

ter, Gadgets, Menüs, Requester etc. und deren Zusammenspiel in der Praxis.



Ich möchte Sie an dieser Stelle ermutigen, die mitgelieferten Beispielprogramme (Verzeichnis: „PureBASIC\Examples\Source“) zu jeder Befehlsbibliothek selbst auszuprobieren, Veränderungen vorzunehmen und danach erneut zu kompilieren. Durch die praktische Anwendung von PureBASIC werden Sie sicher schnell zu gewünschten Erfolgen kommen. Ein größeres Praxis-Projekt, das auch noch weitergehende Aspekte der PureBASIC-Programmierung wie verknüpfte Listen, Datei-Operationen u. v. m. berücksichtigt, finden Sie mit dem Verwaltungsprogramm auf der PureBASIC-CDROM.

I 2 Spieleprogrammierung

PureBasic ist eine einfach zu erlernende Programmiersprache – und die Programme die damit realisiert werden können, werden unglaublich schnell ausgeführt. Dads macht PureBasic zu einer idealen Entwicklungsumgebung für Ihre eignen Spiele. Von Abenteuer- über Strategie- bis hin zu Actionspielen gibt Ihnen PureBasic alle Möglichkeiten an die Hand, die Spielesensation des nächsten Jahres selbst zu programmieren. Und weil dieses Thema so enorm wichtig ist – und darüber hinaus die Programmierung eines eigenen Spiels so viel Spaß macht, haben wir diesem Thema ein eigenes Buch gewidmet. Sie finden es auf der CD-ROM unter dem Titel ‚Strategiespiele selbst programmieren‘ im Hauptverzeichnis.

Darüber hinaus sind weitere Bände und Tutorials geplant, die frei zugänglich sein werden. Bitte informieren Sie sich über das neueste Material auf der Webseite des Verlages (www.topos-verlag.de), auf der offiziellen Webseite von PureBasic (www.purebasic.de) oder im Forum von PureBasic (www.pure-board.de)

13. Die Windows-API

13.1 Einführung

Die Microsoft® Windows® API ermöglicht Ihren Applikationen, die ganze Power der Windows Betriebssysteme auszunutzen, indem Sie Gebrauch von den vielen angebotenen Funktionen machen und somit Ihre Programme aufwerten. Mittels dieser API können Sie Applikationen entwickeln, die problemlos auf allen Windows-Versionen laufen, jedoch auch die Vorteile der jeweiligen Features und Eigenheiten ausnutzen.

Unterschiede bei der Verwendung der verschiedenen Programmierelemente hängen von den Möglichkeiten des verwendeten Betriebssystems (Windows-Version) ab. Meistens liegen die Unterschiede darin, dass einige Funktionen nur auf den stärkeren Plattformen unterstützt werden, z.B. die Service-Funktionen unter Windows NT.

Die Windows API gliedert sich in die folgenden Hauptkategorien:

- Grundlagen (Base Services)
- Kontrollfenster-Bibliothek (Common Control Library)
- Grafikfunktionen (GDI - Graphics Device Interface)
- Netzwerk (Network Services)
- Benutzerschnittstelle (User Interface)
- Windows Shell

Die „Base Services“ ermöglichen allen Applikationen Zugriff auf die grundlegenden Ressourcen des OS, wie Dateisysteme, Speicher, Schnittstellen, Prozesse und Threads.

Die „Common Control Library“ stellt dem Programmierer die Schnittstellen zum Erstellen, Manipulieren und Abfragen der bekannten Kontrollfenster zur Verfügung.

Das „Graphics Device Interface“ stellt dem Programmierer alle Funktionen und Strukturen zur Grafikausgabe auf dem Monitor, dem Drucker oder anderen Schnittstellen zur Verfügung.

Die „Network Services“ stellen alle Funktionen zur Kommunikation von Anwendungen innerhalb eines Netzwerks zur Verfügung.

Mit den Funktionen des „User Interface“ können schließlich die typischen Windows-Benutzeroberflächen gestaltet werden, indem Fenster, Schalter, Maus- und Tastaturabfragen und vieles mehr realisiert werden kann.

Die „Windows Shell“ schließlich ist zuständig für die grundlegenden

Funktionen und Objekte der Windowsoberfläche, wie Verzeichnisse, Dateien, Icons etc.

13.2 Quellen und Verwendung der WinAPI-Dokumentation

PureBasic bietet Ihnen vollen Zugriff auf die Windows API, um die bereits eingebauten Funktionen noch um die zahllosen Möglichkeiten der von Microsoft® bereitgestellten Programmierschnittstelle zu erweitern.

Es ist jedoch nicht möglich, die mehreren Tausend Funktionen der Windows API im Rahmen der PureBasic - Dokumentation näher zu erläutern.

Wenn Sie die Windows API – Befehle in Ihren PureBasic-Projekten verwenden möchten, empfiehlt sich unbedingt einen Blick in die Microsoft-Entwickler-Dokumentation (MSDN). Diese ist online unter <http://msdn.microsoft.com/> erreichbar und auch auf CDROM als „Microsoft Library“ erhältlich. Zudem sollten Sie der englischen Sprache mächtig sein, da diese Dokumentation ausschließlich in Englisch verfügbar ist.

Damit Sie für die vielen WindowsAPI-Befehle auch auf die Online-Hilfe zurückgreifen können, empfiehlt sich die Verwendung der „Win32.hlp“ Hilfe-Datei.

Sofern Sie diese nicht bereits besitzen, kann diese hier aus dem Internet geladen werden:

<ftp://ftp.borland.com/pub/delphi/techpubs/delphi2/win32.zip>

oder

<http://www.borland.com/devsupport/delphi/downloads/>

Entpacken Sie die ZIP-Datei und kopieren Sie die „Win32.hlp“ Datei in Ihr Verzeichnis „PureBasic\Help“. Jetzt steht Ihnen die Online-Hilfe beim Drücken der F1-Taste über einem Funktionsnamen genau wie bei eingebauten PureBasic-Befehlen zur Verfügung.

Eine weitere gute Quelle für Dokumentation der WinAPI ist der API-Guide:

<http://www.allapi.net/agnet/apiguide.shtml>

Es ist sehr empfehlenswert, dass Sie die genannten Dokumentationen sorgfältig durchlesen, um die Anwendung von API-Funktionen und deren geforderten Parametern genau zu verstehen.

13.3 Einbindung von WinAPI-Funktionen in PureBasic

Grundsätzlich werden die gleichen Befehlsnamen, wie in der API-Dokumentation angegeben, verwendet. In PureBasic müssen Sie diesen Befehlen lediglich noch *einen Unterstrich* `_'` hinzufügen, damit eine Unterscheidung von den eigenen Befehlen möglich ist.

Vorhandener Beispiel-Code in Visual Basic (VB) wird daher wie folgt nach PureBasic konvertiert:

```
Call APIFuntionName
```

entspricht

```
APIFunctionName_(Parameter)
```

in PureBasic

Hinweise

- Alle API-Konstanten werden in PureBasic mit einer vorangestellten Raute '#' benutzt, so als wären es lokale PureBasic-Konstanten (z.B. #WooHoo, #APIProgramming, #IsFun).
- Versuchen Sie die grundlegende Übersetzung von Visual-Basic Code nach PureBasic zu verstehen. Dann werden Sie in die Lage versetzt, von den zu Tausenden im Internet frei erhältlichen VB-Tutorials zu lernen. In VB-Code werden den verwendeten Variablen oftmals drei Zeichen vorangestellt, die bei der Zuordnung zu einem Typ behilflich sein sollen.

Beispiel

- 1. lngPizza As Long (VB) = Pizza.l (PB)
- 2. strText As String (VB) = Text.s (oder Text\$)
- Wenn eine API-Funktion eine Speicheradresse als Argument erwartet, benutzen Sie einen PureBasic-Zeiger (@, *, ?).

13.4 Beispiel für Umsetzung von WinAPI-Code

Nachfolgend wollen wir uns am Beispiel der API-Funktion „Message-Box“, die weitgehend dem PureBasic-Befehl MessageRequester entspricht, die Einbindung in PureBasic anschauen:

So sieht (auszugsweise) die MSDN-Dokumentation für diese Funktion aus:

MessageBox

The *MessageBox* function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

```
int MessageBox(  
    HWND Fehler! Hyperlink-Referenz ungültig.,    // handle to owner  
    window  
    LPCTSTR Fehler! Hyperlink-Referenz ungültig., // text in message  
    box  
    LPCTSTR Fehler! Hyperlink-Referenz ungültig., // message box title  
    UINT Fehler! Hyperlink-Referenz ungültig.     // message box style  
);
```

*Parameters**hWnd*

[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText

[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption

[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title *Error* is used.

uType

[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

Wie Sie oben erkennen können, verlangt die Funktion „MessageBox“ vier Argumente: hWnd, lpText, lpCaption und uType.

Umgesetzt in PureBasic sieht diese Funktion so aus:

```
MessageBox_(Win,Text$,Titel$;Flags)
```

Ein vollständiges Beispiel, das die Verwendung von MessageBox zeigt:

```
Win.1 = OpenWindow(0,50,50,300,200,„  
#PB_Window_SystemMenu,"Test-Fenster")  
  
Messagebox_(Win,"Wichtiger Hinweis","Information",„  
#MB_OK | #MB_ICONWARNING)  
  
; #MB_ICONEXCLAMATION ergibt das gleiche Symbol  
  
Messagebox_(Win,"Ein wichtige Information","Information",„  
#MB_OK | #MB_ICONINFORMATION)
```

```

MessageBox_(Win,"Geht's Ihnen gut ?","Frage",,
#MB_YESNO | #MB_ICONQUESTION)

MessageBox_(Win,"Es ist ein Fehler aufgetreten!","Info",,
#MB_ABORTRETRYIGNORE | #MB_ICONSTOP)

; #MB_ICONERROR und #MB_ICONHAND ergeben das gleiche Symbol

```

13.5 Umwandlung von API-Datentypen

Um Sie bei der Portierung der zahllosen Beispiele mit Windows API Quellcode zu unterstützen, erhalten Sie nachfolgend eine Auflistung der wichtigsten Windows-Datentypen, deren Bedeutung und die Zuordnung eines passenden PureBasic-Datentyps:

Hinweis

Es gibt eine Menge verschiedener Datentypen in Windows, da es auch eine Vielzahl an verschiedenen Aufgaben gibt. Diese Typen können durch einen äquivalenten PureBasic-Typ ersetzt werden, meistens vom Typ Long.

Mit Strings sind stets null-terminierte Strings gemeint.

Wenn bei einer WinAPI-Funktion als Parameter „VOID“ angegeben ist, bedeutet dies, dass die entsprechende Funktion keinen Parameter erwartet.

Die folgende Tabelle enthält alle Windows-Typen außer LONG (siehe hierfür die gesonderte Tabelle weiter unten) und die zugehörigen PureBasic-Typen, welche Sie stattdessen benutzen können.

<i>WinAPI-Typ</i>	<i>PureBasic-Typ</i>	<i>Bemerkung</i>
BOOL bzw. BOOLEAN	Byte, Word, Long	#TRUE (1) oder #FALSE (0)
BYTE	Byte	
CHAR	Byte	Character (Zeichen)
CONST	Byte, Word, Long	Konstante

FLOAT	Float	
SHORT oder WORD	Word	
TBYTE	Byte oder Word	Entspricht WCHAR wenn ein UNICODE definiert wird, andernfalls einem CHAR.
TCHAR	Byte oder Word	Entspricht WCHAR wenn ein UNICODE definiert wird, andernfalls einem CHAR.
UNSIGNED	normaler Typ	Vorzeichenlose Attribute (muss mit Hex-Werten gefüllt werden, Ergebnisse werden mit StrU() ermittelt.)
UCHAR	Byte	Vorzeichenloses CHAR (muss mit Hex-Werten gefüllt werden, Ergebnisse werden mit StrU() ermittelt.)
USHORT	Word	Vorzeichenlose SHORT (muss mit Hex-Werten gefüllt werden, Ergebnisse werden mit StrU() ermittelt.)
WCHAR	Word	16-bit Unicode Zeichen

Die folgenden Windows-Typen können alle durch den Long-Typ in PureBasic ersetzt werden:

<i>WinAPI-Typ</i>	<i>Bemerkung</i>
-------------------	------------------

DWORD

DWORD32

DWORD_PTR	(Zeiger)
INT	
INT_PTR	(Zeiger)
INT32	
LONG	
LONG_PTR	(Zeiger)
LONG32	
COLORREF	Farb-Format (RGB), eine Long-Variable mit einem Wert von \$000000 - \$FFFFFF
LPARAM	Message-Parameter
LPARAM	Message-Parameter
WPARAM	Message-Parameter
LRESULT	Vorzeichenbehaftetes Ergebnis der Message-Verarbeitung
LANGID	Sprachen-Identifizier, use LONG
LCID	Locale-Identifizier
LCTYPE	Locale-Information Typ

Die folgenden Typen sind Windows-Typen, die mit vorzeichenlosem

Long ersetzt werden können.
(Benutzen Sie dafür Long-Variablen, die Sie mit Hex-Werten füllen. Bei der Ermittlung von Ergebnissen verwenden Sie StrU().)

UINT

UINT_PTR	(Zeiger)
----------	----------

UINT32

ULONG

ULONG_PTR	(Zeiger)
-----------	----------

ULONG32

Die folgenden Windows-Typen können nicht direkt in PureBasic verwendet werden:

ATOM	Atom-Tabellen sind systemseitig definierte String-Tabellen, es gibt keinen PureBasic-Typ dafür
------	--

REGSAM	“Security access mask for registry key“, keinem Typ zuzuordnen.
--------	---

WINAPI	Aufruf-Konvention für Systemfunktionen, kein eigentlicher Typ.
--------	--

CALLBACK	Aufruf-Konvention für Callback-Funktionen, kein eigentlicher Typ.
----------	---

----	Kritische Section-Variable, Typ nicht bekannt.
------	--

CAL_SECTION

LUID	Lokaler Unique-Identifiziert (DOUBLE Wert), der nur durch Funktionen und nicht direkt aufgerufen wird.
SIZE_T	Die maximale Anzahl an Bytes, auf welche ein Pointer (Zeiger) verweisen kann.
SSIZE_T	Vorzeichenbehaftetes SIZE_T.

Die folgenden Windows-Typen sind vom Typ "Double" (wird von PureBasic noch nicht unterstützt).

DWORD64

INT64

LONG64

LONGLONG

UINT64	(vorzeichenlos)
--------	-----------------

ULONG64	(vorzeichenlos)
---------	-----------------

ULONGLONG	(vorzeichenlos)
-----------	-----------------

Die nachfolgenden Datentypen stellen "Handles" dar, sie identifizieren Objekte wie Windows, Bitmaps, Fonts etc. Die „Handles“ werden in PureBasic immer in Long-Variablen gespeichert.

HACCEL	Handle to an accelerator table.
HANDLE	Handle to an object.
HBITMAP	Handle to a bitmap.
HBRUSH	Handle to a brush.
HCONV	Handle to a dynamic data exchange (DDE) conversation.
HCONVLIST	Handle to a DDE conversation list.
HCURSOR	Handle to a cursor.
HDC	Handle to a device context (DC).
HDDEDATA	Handle to DDE data.
HDESK	Handle to a desktop.
HDROP	Handle to an internal drop structure.
HDWP	Handle to a deferred window position structure.
HENHMETAFILE	Handle to an enhanced metafile.
HFILE	Handle to a file opened by OpenFile, not CreateFile.
HFONT	Handle to a font.

HGDIOBJ	Handle to a GDI object.
HGLOBAL	Handle to a global memory block.
HHOOK	Handle to a hook.
HICON	Handle to an icon.
HIMAGELIST	Handle to an image list.
HIMC	Handle to input context.
HINSTANCE	Handle to an instance.
HKEY	Handle to a registry key.
HKL	Input locale identifier.
HLOCAL	Handle to a local memory block.
HMENU	Handle to a menu.
HMETAFILE	Handle to a metafile.
HMODULE	Handle to a module.
HMONITOR	Handle to a display monitor.
HPALETTE	Handle to a palette.
HPEN	Handle to a pen.

HRGN	Handle to a region.
HRSRC	Handle to a resource.
HSZ	Handle to a DDE string.
HWINSTA	Handle to a window station.
HWND	Handle to a Window.
SC_HANDLE	Handle to a service control manager database.
SC_LOCK	Handle to a service control manager database lock.
SERVICE_STATUS_HANDLE	Handle to a service status value

.Die folgenden Datentypen sind Zeiger (Speicheradressen mit Strukturen/Variablen). Sie sind ebenso vom Typ Long.

LPBOOL	Pointer to a BOOL.
LPBYTE	Pointer to a BYTE.
LPCOLORREF	Pointer to a COLORREF value.
LPCritical_SECTION	Pointer to a CRITICAL_SECTION.
LPCSTR	Pointer to a constant String.

LPCTSTR	An LPCWSTR if UNICODE is defined, an LPCSTR otherwise.
LPCVOID	Pointer to a constant of any type.
LPCWSTR	Pointer to a constant string of 16-bit Unicode characters.
LPDWORD	Pointer to a DWORD.
LPHANDLE	Pointer to a HANDLE.
LPINT	Pointer to an INT.
LPLONG	Pointer to a LONG.
LPSTR	Pointer to a string
LPTSTR	An LPWSTR if UNICODE is defined, an LPSTR otherwise.
LPVOID	Pointer to any type.
LPWORD	Pointer to a WORD.
LPWSTR	Pointer to a string of 16-bit Unicode characters.
PBOOL	Pointer to a BOOL.
PBOOLEAN	Pointer to a BOOL.
PBYTE	Pointer to a BYTE.

PCHAR	Pointer to a CHAR.
PCritical_SECTION	Pointer to a CRITICAL_SECTION.
PCSTR	Pointer to a string
PCTSTR	A PCWSTR if UNICODE is defined, a PCSTR otherwise.
PCWCH	Pointer to a constant WCHAR.
PCWSTR	Pointer to a constant string of 16-bit Unicode characters.
PDWORD	Pointer to a DWORD.
PFLOAT	Pointer to a FLOAT.
PHANDLE	Pointer to a HANDLE.
PHKEY	Pointer to an HKEY.
PINT	Pointer to an INT.
PLCID	Pointer to an LCID.
PLONG	Pointer to a LONG.
PLUID	Pointer to a LUID.
POINTER_32	32-bit pointer.

POINTER_64	64-bit pointer. On a 64-bit system, this is a native pointer. On a 32-bit system, this is a sign-extended 32-bit pointer.
PSHORT	Pointer to a SHORT.
PSTR	Pointer to a string
PTBYTE	Pointer to a TBYTE.
PTCHAR	Pointer to a TCHAR.
PTSTR	A PWSTR if UNICODE is defined, a PSTR otherwise.
PUCHAR	Pointer to a UCHAR.
PUINT	Pointer to a UINT.
PULONG	Pointer to a ULONG.
PUSHORT	Pointer to a USHORT.
PVOID	Pointer to any type.
PWCHAR	Pointer to a WCHAR.
PWORD	Pointer to a WORD.
PWSTR	Pointer to a string of 16-bit Unicode characters.

14. Inline-Assembler und direkte NASM-Programmierung

14.1 Einführung

Mit Assembler können Sie in Ihren PureBasic-Programmen zeitkritische Stellen durch Verwendung von eigenen Routinen beschleunigen oder auch komplett eigene Funktionen schreiben. Gute Kenntnisse der ASM-Programmierung sind dafür jedoch Voraussetzung und werden deshalb auch nicht in diesem Handbuch dokumentiert.

PureBasic erlaubt das direkte Einfügen von allen x86 Assembler Befehlen (einschließlich MMX- und FPU-Befehlen) in den Sourcecode, so als wäre es selbst ein echter Assembler.

14.2. Verwendung und Syntax

Einige Hinweise zum Verwenden von Inline-ASM:

- Sie können direkt alle Variablen oder Zeiger in den Assembler Schlüsselwörtern benutzen, diese werden von PureBasic selbst übersetzt.
- Sie können beliebige Assembler Befehle auf derselben Zeile verwenden.
- Der Syntax entspricht dem des NASM. Wenn Sie nach weiteren Informationen zum ASM-Syntax suchen, lesen Sie einfach die NASM-Anleitung (u.a. <http://www.octium.net/nasm/>).

Beachten Sie bitte auch, dass Ihre Variablen im Programmcode nicht denselben Namen wie ein x86 Schlüsselwort (MOV, TST, JMP etc...) haben können.

Sie müssen einige Regeln genau beachten, wenn Sie ASM im Basic-Code einbinden möchten:

- Die benutzten Variablen oder Zeiger müssen *vor* ihrer Benutzung in einem Assembler Schlüsselwort deklariert werden.
- Wenn Sie auf eine Sprungmarke verweisen, müssen Sie das Zeichen 'p' vor dem Namen einfügen. Dies ist erforderlich, weil PureBasic bei der Kompilierung ein 'p' vor jeder BASIC-Sprungmarke einfügt, um Konflikte mit internen Sprungmarken zu vermeiden.

Beispiel

```
MOV ebx, pMyLabel
...
MyLabel:
```

Die Fehler in einem ASM-Programmteil werden nicht vom PureBasic-Debugger gemeldet, jedoch von NASM. Überprüfen Sie einfach Ihren Programmcode, wenn ein solcher Fehler auftritt.

Die verwendbaren Register sind: eax, ebx, edx, esi, edi und ebp. Alle anderen müssen immer reserviert bleiben.

Damit Sie Inline-Assembler in Ihrem PureBasic-Programmcode verwenden können, muss vor der Kompilierung im Editor-Menü „Compiler-Optionen“ der Punkt „InlineASM Unterstützung einschalten“ aktiviert sein.

Sehen Sie sich auch das Beispielprogramm „AsmInline.pb“ in „PureBasic\Examples\Sources“ an.

14.3 InlineASM und direkte NASM-Programmierung

Wie bereits im Grundlagen-Kapitel geschrieben, übersetzt der PureBasic-Compiler den BASIC-Programmcode in optimierten Assembler-Code und lässt diesen anschließend durch den Assembler „NASM“ in Maschinencode umwandeln.

Sie haben deshalb in PureBasic neben der Verwendung des Inline-Assemblers auch noch eine weitere Möglichkeit:

Sie können den NASM-Assembler direkt ansprechen, indem Sie betreffende Befehlszeilen mit einem „!“ (Ausrufezeichen) einleiten. Diese Zeilen werden von PureBasic nicht geändert und direkt dem NASM-Compiler übergeben. Daher kommt auch der Begriff „Direkt-NASM“.

Wenn Sie in der Verwendung von NASM-Assembler geübt sind, können Sie Ihren Programmen noch mehr Power verleihen.

Nachfolgend noch ein kleines Beispiel, das einmal die Verwendung von „Inline-ASM“ und einmal von „Direkt-NASM“ zeigt:

Inline ASM:

```
wert.1 = 12
MOV wert, 85
MessageRequester("Info", StrU(wert,2), 0)
```

Direkt NASM:

```
wert.l = 12
!MOV dword [v_wert], 85      ; Beachten Sie das "!" am
                             ; Anfang der Zeile
MessageRequester("Info", StrU(wert,2), 0)
```

15. Zusätzliche Informationen

Die Informationen in diesem Kapitel sind insbesondere für alle fortgeschrittenen Anwender gedacht, die ihr PureBasic selbst erweitern, eigene Befehlsbibliotheken (DLL) schreiben und alle weiteren Besonderheiten kennen lernen möchten.

15.1 ‚Include‘ Funktionen

‚Include‘ bedeutet zu deutsch „Einfügen“. In PureBasic gibt es verschiedene Möglichkeiten zum Einfügen von Dateien:

- Einfügen von weiterem Programmcode in den aktuellen Code
- Einfügen von Binär-Dateien

Einfügen von Programmcode

Wenn Ihre eigenen PureBasic - Projekte größer werden, wird damit auch der eigentliche Programmcode immer größer und damit u.U. auch unübersichtlicher. Deshalb bietet es sich an, einzelne Programmteile in gesonderte Sourcecode-Dateien auszulagern.

Damit diese beim Kompilieren auch wieder mit eingebunden werden und somit Ihr fertiges Programm ergeben, müssen wir dies im Sourcecode dem PureBasic-Compiler mitteilen.

Dies erfolgt mittels:

```
IncludeFile "Dateiname"  
XIncludeFile "Dateiname"
```

IncludeFile erwartet als Parameter den Dateinamen (ggf. mit komplettem Pfad) und fügt beim Kompilieren die angegebene Programmdatei an der aktuellen Stelle in den Programmcode ein. *XIncludeFile* hat genau dieselbe Funktion, es vermeidet jedoch die gleiche Datei mehrfach einzufügen.

Beispiel

```
IncludeFile "Sources\Projekt1.pb"  
; diese Datei wird eingefügt  
  
XIncludeFile "Sources\Projekt2.pb"  
; diese Datei wird eingefügt  
  
XIncludeFile "Sources\Projekt2.pb"  
; diese Datei wird ignoriert, wie auch alle  
; späteren Aufrufe...
```

Falls Sie viele verschiedene Dateien einzufügen haben, die sich alle im gleichen Verzeichnis befinden, würden Sie sicher gern auf die Pfadangabe verzichten. Kein Problem, benutzen Sie vor den Anweisungen *IncludeFile* bzw. *XIncludeFile* die folgende Anweisung:

```
IncludePath "Dateipfad"
```

Mit *IncludePath* legen Sie den Standard-Dateipfad für alle nach diesem Befehl einzufügenden Sourcecode-Dateien fest.

Beispiel

```
IncludePath "Projekt\Daten" ; definiert "Projekt\Daten"  
als Standard-Dateipfad  
IncludeFile "Sprite.pb" ; fügt die Datei "Sprite.pb"  
an dieser Stelle ein  
XIncludeFile "Music.pb" ; fügt die Datei "Music.pb"  
an dieser Stelle ein
```

Einfügen von Binär-Daten

Wenn Sie in Ihren fertigen Programmen nicht alle zusätzlichen Dateien (Bilder, Sounds, Daten) während der Programmausführung nachladen möchten, können Sie diese auch in Ihr fertiges Programm mit einbinden lassen. Dazu wird der folgende Befehl verwendet:

```
IncludeBinary "Pfad\Dateiname"
```

IncludeBinary erwartet als Parameter den kompletten Pfad sowie Dateinamen und fügt beim Kompilieren die angegebene Binär-Datei an der aktuellen Stelle im Programm ein.

Beispiel

```
IncludeBinary "C:\Data\map.data"
```

Der Zugriff auf die eingebundenen Dateien erfolgt innerhalb des Pro-

gramms mit Hilfe von Zeigern. Sehen Sie für ein vollständiges Beispiel im Kapitel 7.9 „Adressen von Sprungmarken“ nach.

15.2 Compiler-Direktiven

Compiler-Direktiven sind spezielle Schlüsselwörter im Programmcode, durch deren Verwendung das Kompilieren einzelner Programmabschnitte gezielt gesteuert werden kann.

Nützlich ist dies insbesondere, wenn Sie mit PureBasic Multi-OS (auf mehreren Betriebssystemen lauffähige) Programme erstellen und dabei einige Programmabschnitte mit OS-spezifischen Funktionen individuell für jedes Betriebssystem gestalten. Z.B. indem Sie bei der Windows-Version Ihres Programms Befehle der Windows-API einsetzen und dieselben Funktionen unter Linux oder AmigaOS natürlich anderweitig realisieren müssen.

Die Funktionsweise der Compiler-Direktiven entspricht denen regulärer If- und Select-Bedingungen (siehe Kapitel 9) in PureBasic. Beachten Sie jedoch, dass als Parameter für die Bedingungen nur Konstanten oder numerische Werte zum Einsatz kommen können.

Der Syntax einer *If-Bedingung* sieht wie folgt aus:

```
CompilerIf <Ausdruck>
...
[CompilerElse]
...
CompilerEndIf
```

Durch Angabe eines Ausdrucks in der Form: ‚Konstante = Wert oder Konstante‘ nach dem Schlüsselwort *CompilerIf* wird bei Zutreffen der Bedingung (Wert des Ausdrucks = TRUE) der zugehörige Codeabschnitt kompiliert. Andernfalls wird dieser erste Programmabschnitt vollkommen ignoriert.

Bei Verwendung des Schlüsselworts *CompilerElse* wird stattdessen der zweite Programmschnitt mit eingebunden.

Beispiel

```
CompilerIf #OS = #Linux
; etwas Linux spezifischer Programmcode..
CompilerEndIf
```

Der Syntax einer *Select-Bedingung* sieht wie folgt aus:

```
CompilerSelect <numerischer Wert>
  CompilerCase
    ...
  [CompilerElse]
    ...
  [CompilerDefault]
    ...
CompilerEndSelect
```

Diese Bedingung arbeitet exakt wie das reguläre Select : EndSelect (siehe Kapitel 9.2), teilt jedoch in ihrer Funktion als Compiler-Direktive dem Compiler mit, welcher Programmcode kompiliert werden soll.

Beispiel

```
CompilerSelect #OS
  CompilerCase #Windows
    ; etwas Windows spezifischer Programmcode
  CompilerCase #AmigaOS
    ; etwas Amiga spezifischer Programmcode
  CompilerCase #Linux
    ; etwas Linux spezifischer Programmcode
CompilerEndSelect
```

15.3 Verwendung alternativer Editoren

Der mitgelieferte Editor zu PureBasic bietet einige Vorteile wie farblich hervorgehobene Befehle, Schlüsselwörter, Kommentare usw., kontext-sensitive Online-Hilfe (nach Drücken der F1-Taste) zu eingebauten und WinAPI-Befehlen, Sprungmarken-Requester etc. Manche Programmierer werden jedoch trotzdem nicht auf ihren gewohnten Editor verzichten wollen, da dieser einige zusätzliche Funktionen wie gleichzeitige Bearbeitung mehrerer Sourcecodes, Zusammenklappen von Prozeduren u.ä. bietet.

Auch das ist kein Problem – der eigentliche Compiler von PureBasic ist schließlich unabhängig vom Editor. Sie können damit Ihren Programmcode in jedem beliebigen Editor bearbeiten und anschließend mit dem Compiler ein ausführbares Programm erzeugen.

Einige bekannte und beliebte Editoren wären:

<i>Name</i>	<i>Status</i>	<i>Homepage</i>
ConText	Freeware	http://www.fixedsys.com/context/
UltraEdit	Shareware	http://www.ultraedit.com/

EditPad-
Pro

<http://www.editpadpro.com>

Jens File Freeware
Editor

http://home.t-online.de/home/Jens.Altmann/jfe_eng.htm

Die Installation und Konfiguration zweier bekannter Editoren, in die sich der Compiler sogar einbinden lässt, wollen wir uns nachfolgend etwas näher ansehen.

UltraEdit

Die Datei „PEdit.exe“ finden Sie im Archiv „PBUEdit“ auf der PureBasic Resources Site (<http://www.reelmediaproductions.com/pb/>) unter der Rubrik „Editor Resources“. Downloaden und entpacken Sie das Archiv und gehen Sie wie folgt vor:

Einstellen des Syntax-Highlighting (Farbliches Hervorheben von Befehlen etc. im Sourcecode)

Kopieren Sie die Datei PBedit.exe in das Verzeichnis PureBasic/Compilers. Führen Sie die genannte Datei mit einem Doppelklick aus. (Das Programm ermittelt aus den Dateien „PBFunctionListing.txt“ und „Purebasic.exe“ die gültigen Befehle und Schlüsselwörter.)

Eine Datei „UEDIT_SyntaxFile_PureBasic.TXT“ wird automatisch erstellt.

Den Inhalt der genannten Datei in die Datei „Wordfile.txt“ im Verzeichnis UltraEdit kopieren. (Vorher bitte überprüfen, ob der Inhalt bereits in der Datei „Wordfile.txt“ existiert.)

Im Editor UltraEdit im Menü „Extras/Optionen“ aufrufen und im Untermenü „Dateitypen“ einen neuen Dateityp „PureBasic-Dateien“ (Dateiname „*.PB“ / Beschreibung „PureBasic-Dateien, (*.PB)“) erstellen.

Dieses Fenster mit „OK“ schließen, UltraEdit beenden und erneut starten. Voila ! Beim Öffnen einer PureBasic-Sourcecode-Datei steht ab sofort die farbliche Hervorhebung von Befehlen, Schlüsselwörtern etc. zur Verfügung.

Konfigurieren von UltraEdit zur Benutzung des PureBasic - Compilers

Starten Sie UltraEdit und rufen Sie im Menü „Extras“ den Punkt „Werkzeug-Konfiguration“ auf.

Nehmen Sie folgende Einstellungen vor:

Befehlszeile: Pfad zum PureBasic-Compiler (z.B. „C:\Programmierung\PureBasic\Comilers\PBCompiler.exe“) und Zusatz „/QUIET %F“

Bezeichnung für Menüeintrag: „PureBasic – Start“ (oder ähnlich)

Befehlsausgabe: Markierungen bei „Ausgabe in Listenfeld“ und „Ausgabe aufzeichnen“, alle weiteren Felder bleiben leer.

Drücken Sie den Schalter „Hinzufügen“ und anschließend „OK“.

Wenn Sie jetzt im Menü „Extras“ nachschauen, werden Sie einen neuen Menüpunkt „PureBasic – Start“ mit dem Tastenkürzel „Strg-Umschalt-0“ vorfinden. Damit können Sie in UltraEdit geladenen Sourcecode kompilieren lassen.

Um dieses Tastenkürzel auf die vom PureBasic-Editor gewohnte „F5“-Taste zu ändern, gehen Sie wie folgt vor:

Rufen Sie im Menü „Extras“ den Punkt „Optionen“ auf.

Im jetzt geöffneten Fenster „UltraEdit-Optionen“ wählen Sie die Schalttafel „Tastenzuordnung“.

Suchen Sie in der Auswahlliste nach „ExtrasWerkzeug1“ (oder „...2“, „...3“ etc. falls Sie in Ihrem UltraEdit bereits andere Werkzeuge definiert haben) und klicken Sie diesen Eintrag an.

Löschen Sie im Feld „Bestehende Tastenzuordnungen“ den alten Eintrag „Ctrl-Shift-0“, drücken Sie im Feld „Neue Taste drücken“ die Taste „F5“ und klicken auf „Zuordnen“.

Nach Klick auf „OK“ wird das Optionen-Fenster geschlossen und Sie können im Menü den geänderten Eintrag „PureBasic – Start F5“ betrachten.

ConText

Einstellen des Syntax-Highlighting

Downloaden Sie das Archiv „PureBasic.chl“ von der PureBasic Resources Site (<http://www.reelmediaproductions.com/pb/>) unter der Rubrik „Editor Resources“. Entpacken Sie anschließend das Archiv in das Programmverzeichnis von „Context\Highlighters\“.

Nach dem nächsten Neustart von Context steht Ihnen Syntax-Highlighting (farblich hervorgehobene Schlüsselwörter, Befehle usw.) bei allen PureBasic-Dateien (.pb) zur Verfügung.

Konfigurieren von ConText zur Benutzung des PureBasic - Compilers

Öffnen Sie eine beliebige PureBasic-Sourcecode-Datei, um die richtige Funktionsweise der Compilereinstellungen später gleich testen zu können.

Rufen Sie den Menüpunkt „Einstellungen/Umgebungseinstellungen“ auf

und im dann angezeigten Fenster den Punkt „Ausführungstasten“. Wählen Sie den Schalter „Hinzufügen“ und geben Sie im anschließend erscheinenden Dialog-Fenster die Dateiendung „pb“ (ohne Anführungszeichen) ein.

Klicken Sie in der dargestellten Baumstruktur auf „F9“, die Eingabefelder auf der rechten Seite werden damit aktiviert.

Unter „Ausführen“ geben Sie den vollständigen Pfad zum PureBasic-Compiler auf Ihrem System (z.B. „C:\PureBasic\Compilers\PBCompiler.exe“ ein.

Als weitere Optionen wählen Sie aus bzw. geben Sie folgendes ein:

Parameter: %n /Debugger /Quiet

Fenster: normal

Hinweis

Kompilieren (oder jeder andere Text, der im Editor-Menü für „F9“ angezeigt werden soll)

Speichern: Aktuelle Datei bevor Ausführung

Häkchen bei „Aufzeichnung Konsolenausgabe“

Klicken Sie auf „OK“ und diese Einstellungen werden gespeichert.

Zukünftig können Sie Ihren geladenen Sourcecode einfach durch Drücken der F9-Taste kompilieren.

15.4 Quellen, Installation und Nutzung von User-Libraries

Fortgeschrittene PureBasic-Programmierer veröffentlichen teilweise auch selbstgeschriebene PureBasic-Befehlsbibliotheken, die problemlos auch von Ihnen in PureBasic eingebunden werden können.

Eine sehr gute Quelle für neu veröffentlichte Libraries ist die „PureBasic Ressources Site“:

<http://www.reelmediaproductions.com/pb/>

Befehlsbibliotheken von Dritt-Anbietern müssen Sie lediglich in das Verzeichnis „PureBasic\PureLibraries\UserLibraries“ kopieren und nach dem nächsten Neustart von PureBasic stehen Ihnen die darin enthaltenen Befehle zur Verfügung.

Wird zu der Library auch eine Dokumentation im MS-Help-Format (.chm) mitgeliefert, kopieren Sie diese Datei in das Verzeichnis „PureBasic\Help“ und fortan steht Ihnen auch diese Befehle nach Druck auf die „F1-Taste“ die Online-Hilfe zur Verfügung.

15.5 Erstellung einer DLL

PureBasic ermöglicht Ihnen das Erstellen von DLL's im Standard

Microsoft Windows Format. Eine DLL (Dynamic Link Library) enthält Funktionen und ggf. auch Daten, auf die über eine standardisierte Softwareschnittstelle durch weitere Programme zugegriffen werden kann.

Damit können Sie oft verwendete Programmteile bzw. Funktionen auszulagern, um diese in späteren PureBasic-Projekten aber auch mit anderen Programmiersprachen wiederzuverwenden.

Beim Programmieren einer DLL sollten Sie folgendes beachten:

- DLL-Programmcode entspricht „normalem“ PureBasic-Code, jedoch sollte sämtlicher Programmcode innerhalb von Prozeduren deklariert werden.
- Der gesamte Programmcode innerhalb einer DLL wird innerhalb von Prozeduren ausgeführt.
- Die Prozeduren werden nach „nicht öffentlichen“ (nur innerhalb der DLL bekannten) Prozeduren und öffentlichen („public“) Prozeduren unterschieden, auf die durch dritte Programme zugegriffen werden kann. Für nicht öffentliche Prozeduren benutzen Sie bei der Deklaration das bekannte Schlüsselwort *Procedure*, für die „public“ Prozeduren das Schlüsselwort *ProcedureDLL*.

Vor dem Kompilieren Ihres DLL-Programmcodes müssen Sie dem Compiler mitteilen, dass Sie kein „normales“ Windows- oder Konsolenprogramm erstellen wollen, sondern eine DLL. Dazu wählen Sie im Editor (Fenster „Compiler-Optionen“) als Ausgabeformat „Shared DLL“ bzw. im Kommandozeilenmodus den Zusatz „/DLL“.

Erstellt wird die DLL schließlich mit dem Namen „PureBasic.dll“ im Verzeichnis „PureBasic\Compilers\“. Der Name kann von Ihnen anschließend beliebig geändert werden.

Eingebunden werden selbst erstellte DLL's und natürlich auch DLL's von Dritt-Anbietern über die Funktionen der mitgelieferten PureBasic-Befehlsbibliothek „Library“. Nach dem Öffnen der DLL mittels der Funktion *OpenLibrary* können alle mittels *ProcedureDLL* als „public“ deklarierte Prozeduren aufgerufen werden. Der Funktionsname muss jedoch mit einem vorangestellten Unterstrich „_“ aufgerufen werden, damit PureBasic zwischen eigenen Befehlsbibliotheken und DLL's unterscheiden kann.

Beispiel

```
; Nachfolgend wird die DLL deklariert,
; die als „Shared DLL“ kompiliert wird
ProcedureDLL MyFunction()
    MessageRequester("Hallo", „
"Dies ist eine PureBasic DLL !", 0)
EndProcedure

; Hier das Client Programm, welches die DLL benutzt
If OpenLibrary(0, "PureBasic.dll")
    CallFunction(0, "_MyFunction")    ; ruft die Funktion der
                                      ; DLL und damit
    CloseLibrary(0)                  ; letztendlich den
                                      ; MessageRequester auf
EndIf
```

Ein paar weitergehende Hinweise für fortgeschrittene Anwender:

Es gibt vier spezielle Prozeduren, die von Windows automatisch aufgerufen werden, wenn eines der folgenden Ereignisse auftritt:

- DLL wurde zu einem neuen Prozess hinzugefügt
- DLL wurde von einem Prozess entfernt
- DLL wurde zu einem neuen Thread hinzugefügt
- DLL wurde von einem Thread entfernt

Um diese Ereignisse zu verwalten, können vier spezielle Prozeduren deklariert werden: AttachProcess(), DetachProcess(), AttachThread() und DetachThread(). Diese vier Prozedur-Namen sind deshalb auch reserviert und können vom Programmierer nicht für andere Zwecke verwendet werden. Sehen Sie hierzu auch den Beispielcode „DLLSample.pb“ in „PureBasic\Examples\Source“.

15.6 Links zu Diskussionsforen und Support

Offizielle Homepage von PureBasic: <http://www.purebasic.com/>

Email an den Autor: support@purebasic.com

Homepage des deutschen Vertriebs: <http://www.purebasic.de/>

Email an den Vertrieb: support@purebasic.de

PureBasic erfreut sich einer regen und stetig wachsenden Anwender-Gemeinde. Diese trifft sich in Foren zum gegenseitigen Informationsaustausch, Hilfestellungen usw.

<http://forums.purebasic.com/> Englischs Forum (direkter Kontakt zum Autor möglich)

<http://www.pure-board.de/> Deutsches Forum

(Direkter Link: <http://www.shadow-studios.net/phpBB/index.php>)

Der Autor von PureBasic – Frederic Laboureux – ist ausgesprochen interessiert an Anregungen, Bug-Reports usw. seitens der Anwender. Er ist deshalb per Email erreichbar und kommuniziert auch im englischen

Forum mit der PureBasic-Gemeinde.

16. Befehlsreferenz

Dies ist das mit Sicherheit umfangreichste Kapitel des PureBasic-Anwenderhandbuchs. Aber auch kein Wunder – schließlich bietet PureBasic neben den bereits vorgestellten Schlüsselwörtern auch noch über 470 mitgelieferte Befehle.

Bevor wir uns Syntax und Funktion der einzelnen Befehle näher anschauen, vorab ein paar allgemeine Informationen.

Der generelle Aufbau eines PureBasic-Befehls sieht wie folgt aus:

Befehl(Parameter1, Parameter2 [, Parameter])

Der eigentliche Befehls- bzw. Funktionsname ist farblich hervorgehoben.

In Klammern finden sich ein oder mehrere Parameter, je nachdem wie bei dem Befehl gefordert.

Parameter in eckigen [Klammern] sind optional und können je nach Bedarf angegeben oder weggelassen werden.

Sofern die Funktion einen Rückgabewert liefert, ist dies mit „Ergebnis =“ angegeben.

Als Parameter werden bei einigen Funktionen auch „Flags“ gefordert. Ein Flag bezeichnet ein BIT (ähnlich wie bei einer SPS ein Merker) und wird durch eine Konstante angegeben. Es können ein oder mehrere Flag(s) angegeben werden. Dazu verbindet man die einzelnen Konstanten mit dem bitweisen Oder (Symbol „|“), z.B. bei der Funktion

```
OpenWindow(..., #PB_Window_SystemMenu | #PB_Window_MinimizeGadget | #PB_Window_MaximizeGadget, ...).
```

Bei einigen Befehlsbibliotheken ist es erforderlich, vor der Nutzung der enthaltenen Befehle erst die dafür erforderliche Programmumgebung zu initialisieren. Die entsprechenden Befehle beginnen mit *Init...()* bzw. *Start...()* und werden als erstes in der Dokumentation zur jeweiligen Befehlsbibliothek beschrieben, da erst nach einem erfolgreichen Aufruf alle weiteren Befehle aus dieser Befehlsbibliothek nutzbar werden. Diese weiteren Befehle folgend deshalb anschließend in alphabetischer Reihenfolge.

Es ist sehr empfehlenswert, die zu PureBasic mitgelieferten Beispielprogramme im Verzeichnis „PureBasic\Examples\Sources“ auszuprobieren um Funktion und Zusammenspiel der einzelnen PureBasic-Befehle in der Praxis kennenzulernen.

Übersicht über die mitgelieferten Befehlsbibliotheken:

Console	Nutzung des Konsolenmodus
Math	Mathematische Funktionen
String	Funktionen zur Manipulation von Strings
Sort	Sortierung von Arrays
Linked List	Verwaltung von verknüpften Listen
Memory	Funktionen zur Speicherverwaltung
File	Schreib- und Leseoperationen in Dateien
FileSystem	Operationen mit Dateien und Verzeichnissen
Misc	Diverse Befehle (u.a. zu Farbwerten, Dateien, Peek & Poke)
Palette	Verwalten von Farbpaletten
Image	Einbinden und Verwendung von Bildern
2D Drawing	Zeichenoperationen für das zweidimensionale Zeichnen
Sound	Funktionen zur Soundausgabe
Module	Abspielen von Musik-Modulen
CDAudio	Abspielen von Audio-CDs

Movie	Abspielen von Movie- und Musikdateien (AVI, MPG, DivX, Mp3 etc.)
Sprite Screen	& Funktionen zur Erstellung und Anzeige von Sprites
Sprite3D	Funktionen zur 3D-Darstellung von Sprites
Mouse	Abfragen von Mausereignissen (für Spiele)
Keyboard	Abfragen von Tastaturereignissen (für Spiele)
Joystick	Abfragen von Joysticks
Window	Erstellen von Fenstern und Abfragen von Ereignissen
Font	Verwendung von Zeichensätzen
Gadget	Erstellen und Abfragen graphischer Benutzeroberflächen (GUI)
Menu	Erstellen und Abfragen von Menüs
Requester	Nutzung der Standard-Systemrequester
Toolbar	Erstellen von Werkzeugleisten
StatusBar	Erstellen von Status-Leisten für Fenster
Clipboard	Nutzung der Zwischenablage
Preference	Erstellen und Verwalten von Voreinstellungs-Dateien

Printer	Funktionen zur Druckausgabe
Network	Datenübertragung innerhalb eines Netzwerks bzw. im Internet
Help	Einbindung von MS Hilfe-Dateien
Database	Nutzung von ODBC-Datenbanken
Packer	Komprimieren und Dekomprimieren von Dateien und Speicherbereichen
Cipher	Verschlüsselungs-Algorithmen
SysTray	Erstellen und Verwalten von Systray-Icons
Thread	Deklaration von asynchronen Programmteilen (Threads)
Library	Einbindung externer Befehlsbibliotheken (DLL)

16.1 Console

Diese Library ermöglicht das Schreiben von Applikationen im Konsolen-Modus. Dieser wird benutzt, um kleine Programme zu erstellen, die kein User-Interface benötigen oder in einem Skript (Befehlszeilen) benutzt werden sollen. Diese Befehle helfen dem Programmierer auch beim Debuggen eines Programms, indem einige Informationen auf der Konsole ausgegeben werden können, ohne den Programmablauf anzuhalten.

OpenConsole()

Ergebnis = *OpenConsole()*

Öffnet ein Konsolen-Fenster. Diese Funktion muss vor den anderen Befehlen aus dieser Library aufgerufen werden. Nur eine Konsole kann

zur gleichen Zeit in einem PureBasic Programm geöffnet werden. Ist 'Ergebnis' gleich 0, ist das Öffnen fehlgeschlagen und alle weiteren Aufrufe von Console-Befehlen müssen deaktiviert werden. Die Konsole kann mittels dem *CloseConsole()* Befehl geschlossen werden bzw. automatisch beim Aufruf von „End“.

Unter Microsoft Windows gibt es zwei verschiedene "Executable" Formate (von ausführbaren Programmen): Win32 und Console. Wenn Sie eine Standard-Konsolen-Applikation, wie 'Dir', 'Del' etc... erstellen möchten, müssen Sie das Executable im Format "Console" (Compiler-Optionen Menü im PureBasic-Editor) erstellen.

ClearConsole()

Löscht den gesamten Konsolen-Inhalt. Die Hintergrundfarbe wird mittels *ConsoleColor()* festgelegt.

CloseConsole()

Ergebnis = *CloseConsole()*

Schließt die zuvor mittels *OpenConsole()* geöffnete Konsole. Sobald eine Konsole geschlossen wurde, können keine Funktionen aus dieser Library mehr benutzt werden. Ist der Rückgabewert gleich 0, ist der vorherige *OpenConsole()* Befehl fehlgeschlagen.

ConsoleColor(Zeichenfarbe, Hintergrundfarbe)

ConsoleColor(Zeichenfarbe, Hintergrundfarbe)

Ändert die Farben der Textanzeige. Die Farbwerte reichen von 0 bis 15, welches die im Konsolen-Modus verfügbaren 16 Farben sind.

ConsoleCursor(Höhe)

Ändert die Cursor-Anzeige. Ist die Höhe gleich 0, ist der Cursor unsichtbar. Sonst kann 'Höhe' einen Wert zwischen 1 und 10 annehmen.

Zurzeit sind (nur) 3 Werte verfügbar:

1 = Cursor unterstrichen

5 = Cursor mittlere Höhe

10 = Cursor volle Höhe

ConsoleLocate(x, y)

Bewegt den Cursor an die angegebene Position, in Zeichen.

ConsoleTitle(Titel\$)

Ändert den Konsolentitel auf den neu angegebenen.

Inkey()

String\$ = *Inkey()*

Gibt einen "2-Zeichen"-String zurück, wenn eine Taste während dem Aufruf von *Inkey()* gedrückt wurde. Der Programmablauf wird dabei nicht unterbrochen. Ist das Ergebnis 'String\$' nicht leer, beinhaltet es zwei Zeichen: das linke Zeichen ist der ASCII-Wert der gedrückten Taste, und das rechte Zeichen beinhaltet die zugehörige Nummer. Dies ist nützlich für nicht-alphabetische Tasten (Funktionstasten, Cursorpfeile, etc...).

Input()

String\$ = *Input()*

Ermöglicht eine ganze Zeichenkette zu erfassen. Dieser Befehl hält die Programmausführung an und wartet bis der Benutzer die Return-Taste drückt. Die Eingabezeile (Zeichenkette) kann nicht länger als die Konsolen-Breite sein.

Print(Text\$)

Zeigt den 'Text\$' in der Konsole an. Die Text\$-Länge sollte nicht über 250 Zeichen liegen, andernfalls wird das Ende des Strings abgeschnitten. Um die aktuelle Cursor-Position zu ändern, benutzen Sie den Befehl *ConsoleLocate()*.

PrintN(Text\$)

Zeigt den 'Text\$' in der Konsole an und fügt einen Zeilenumbruch hinzu. Die Text\$-Länge sollte nicht über 250 Zeichen liegen, andernfalls wird das Ende des Strings abgeschnitten. Um die aktuelle Cursor-Position zu ändern, benutzen Sie den Befehl *ConsoleLocate()*.

16.2 Math

Die 'Math' (Mathe) Library bietet grundlegende arithmetische Funktionen wie *Cos()*, *Sin()*, *Pow()*, *Log()* etc... Beinahe alle Befehle arbeiten mit Float Zahlen (.f).

Abs(Zahl.f)

Ergebnis.f = *Abs(Zahl.f)*

Gibt den Absolut-Wert (ohne Vorzeichen) der angegebenen Float-Zahl zurück. Diese Funktion arbeitet nur korrekt mit Float-Zahlen. Mit Integer (Ganzzahlen) schlägt sie fehl, wenn die Integerzahl zu groß (Verlust der Präzision) ist. Eine weitere Funktion hierfür wird folgen.

ACos(Winkel.f)

Ergebnis.f = *ACos(Winkel.f)*

Gibt den Arc-Cosinus des angegebenen Winkels zurück, als Bogeneinheit (zwischen 0 und 1).

ASin(Winkel.f)

Ergebnis.f = *ASin(Winkel.f)*

Gibt den Arc-Sinus des angegebenen Winkels zurück, als Bogeneinheit (zwischen 0 und 1).

ATan(Winkel.f)

Ergebnis.f = *ATan(Winkel.f)*

Gibt den Arc-Tangens des angegebenen Winkels zurück, als Bogeneinheit (zwischen 0 und 1).

Cos(Winkel.f)

Ergebnis.f = *Cos(Winkel.f)*

Gibt den Cosinus des angegebenen Winkels zurück, als Bogeneinheit (zwischen 0 und 1).

Log(Zahl.f)

Ergebnis.f = *Log(Zahl.f)*

Gibt den Log(arythmus) der angegebenen Zahl zurück.

Log10(Zahl.f)

Ergebnis.f = *Log10(Zahl.f)*

Gibt den Log(arythmus) auf Basis 10 der angegebenen Zahl zurück.

Pow(Basis.f, Exponent.f)

Ergebnis.f = *Pow(Basis.f, Exponent.f)*

Gibt das Ergebnis von Basis^{Exponent} zurück. (Die Basis.f wird sooft mit sich selbst multipliziert, wie im Exponent.f angegeben.)

Round(Zahl.f, Modus)

Ergebnis.f = *Round(Zahl.f, Modus)*

Rundet die angegebene Float-Zahl abhängig vom angegebenen Modus. Ist Modus = 0, wird die Zahl abgerundet (es wird nur der Integer-Teil benutzt). Ist Modus = 1, wird die Zahl aufgerundet.

Beispiel

```
Ergebnis.f = Round(11.5, 0)    ; ergibt 11  
Ergebnis.f = Round(11.5, 1)    ; ergibt 12
```

Sin(Winkel.f)

Ergebnis.f = *Sin(Winkel.f)*

Gibt den Sinus des angegebenen Winkels zurück, als Bogeneinheit (zwi-

schen 0 und 1).

Sqr(Zahl.f)

Ergebnis.f = *Sqr(Zahl.f)*

Gibt die Quadratwurzel der angegebenen Zahl zurück.

Tan(Winkel.f)

Ergebnis.f = *Tan(Winkel.f)*

Gibt den Tangens des angegebenen Winkels zurück, als Bogeneinheit (zwischen 0 und 1).

16.3 String

Ein String ist eine Möglichkeit, um eine ganze Reihe von Zeichen zu speichern. Mit den Befehlen dieser Library können Sie einige wichtige Aktionen mit den Strings ausführen.

Asc(String\$)

Ascii = *Asc(String\$)*

Ermittelt den ASCII Wert des ersten Zeichens im String\$.

```
ASCII = Asc("!") ; 'ASCII' erhält den Wert '33'
```

Chr(ASCII-Wert)

Text\$ = *Chr(ASCII-Wert)*

Gibt das zum angegebenen ASCII Wert gehörende Zeichen zurück.

```
a$ = Chr(33) ; a$ enthält das Zeichen "!"
```

FindString(String\$, SuchString\$, StartPosition)

Position = *FindString(String\$, SuchString\$, StartPosition)*

Versucht den 'Suchstring\$' im angegebenen 'String\$' - beginnend an der

angegebenen Position (StartPosition.w) - zu finden. Wurde der String gefunden, wird dessen Position (in Anzahl der Zeichen, beginnend bei 1) zurückgegeben. Wurde der String nicht gefunden, gibt die Funktion 0 zurück. Position = FindString("PureBasic", "Bas", 1) ; 'Position' erhält den Wert '5'

Hex(Wert)

Ergebnis\$ = Hex(Wert)

Konvertiert eine numerische Zahl in einen String im Hexadezimal-Format.

Beispiel

```
a$ = Hex(12) ; a$ ergibt "C"
```

LCase(String\$)

Ergebnis\$ = LCase(String\$)

Gibt den originalen String - konvertiert in Kleinbuchstaben (wenn möglich) - zurück. Diese Funktion unterstützt auch Buchstaben mit Betonung; wenn also ein großes 'É' gefunden wurde, wird dieses in ein kleines 'é' konvertiert.

Left(String\$, Länge)

Ergebnis\$ = Left(String\$, Länge)

Gibt die angegebene Anzahl an Zeichen ('Länge') vom linken Rand des 'String\$' zurück. Diese Funktion ist auch sicher zu benutzen, wenn ein falscher Wert für den Längen-Parameter angegeben wurde, und gibt immer das am 'besten' passende Resultat zurück.

Len(String\$)

Länge = Len(String\$)

Ermittelt die Anzahl Zeichen im angegebenen 'String\$'.

LTrim(String\$)

`Ergebnis$ = LTrim(String$)`

Entfernt alle Leerzeichen ('space') vom Anfang des Strings.

Mid(String\$, StartPosition, Länge)

`Ergebnis$ = Mid(String$, StartPosition, Länge)`

Ermittelt einen String der angegebenen 'Länge', beginnend bei 'StartPosition', aus dem vorgegebenen 'String\$'. Der Wert von 'StartPosition' beginnt bei 1, was das erste Zeichen kennzeichnet. 'Länge' ist die benötigte Länge des Ausgabestrings 'Ergebnis\$'.

Beispiel

```
Mid("Hello", 2, 3) ; Resultat wird "ell" sein.
```

ReplaceString(String\$, SuchString\$, ErsatzString\$ [, Modus])

`Ergebnis$ = ReplaceString(String$, SuchString$, ErsatzString$ [, Modus])`

Versucht alle Vorkommen des 'SuchString\$' im angegebenen 'String\$' zu finden und ersetzt diese mit dem 'ErsatzString\$'.

'Modus' ist optional und kann eine Kombination der folgenden Werte sein:

- 1: Suche unabhängig von Groß-/Kleinschreibung (A=a). Standardmäßig beachtet die Suche die Groß-/Kleinschreibung.
- 2: "In place" (an Ort und Stelle) ersetzen. Dies bedeutet, dass der String direkt im Speicher ersetzt wird. Der 'SuchString\$' und der 'ErsatzString\$' müssen die gleiche Länge haben ! Dies ist eine gefährliche Option, nur für fortgeschrittene Anwender. Der Vorteil ist die sehr hohe Geschwindigkeit beim Ersetzen.

Right(String\$, Länge)

`Ergebnis$ = Right(String$, Länge)`

Gibt die angegebene Anzahl an Zeichen ('Länge') vom rechten Rand des 'String\$' zurück. Diese Funktion ist auch sicher zu benutzen, wenn ein falscher Wert für den Längen-Parameter angegeben wurde, und gibt immer das am 'besten' passende Resultat zurück.

RTrim(String\$)

`Ergebnis$ = RTrim(String$)`

Entfernt alle Leerzeichen ('space') vom Ende des Strings.

Space(Länge)

`Ergebnis$ = Space(Länge)`

Erstellt einen leeren String in der angegebenen 'Länge', der mit Leerzeichen ('Space') gefüllt wird.

`Ergebnis$ = Str(Wert)`

Konvertiert eine Zahl in einen String.

`Ergebnis$ = StrF(Wert.f[, NbDecimal])`

Konvertiert eine Fließkomma-Zahl in einen String. Eine maximale Zahl an Dezimalstellen kann angegeben werden. Die Zahl wird nicht gerundet, sondern abgeschnitten.

StrU(Wert, Typ)

`Ergebnis$ = StrU(Wert, Typ)`

Konvertiert eine vorzeichenlose Zahl in einen String.

Die folgenden 'Typen' sind möglich:

#Byte : Der Wert ist eine Byte-Zahl, im Bereich von 0 bis 255

#Word : Der Wert ist eine Word-Zahl, im Bereich von 0 bis 65536

#Long : Der Wert ist eine Long-Zahl, im Bereich von 0 bis 4294967296

Trim(String\$)

`Ergebnis$ = Trim(String$)`

Entfernt alle Leerzeichen ('Space') vom Anfang und vom Ende des 'String\$'.

UCase(String\$)

`Ergebnis$ = UCase(String$)`

Gibt den originalen String - konvertiert in Großbuchstaben (wenn mög-

lich) - zurück. Diese Funktion unterstützt auch Buchstaben mit Betonung; wenn also ein 'é' gefunden wurde, wird dieses in ein großes 'É' konvertiert.

Val(String\$)

Ergebnis.l = *Val(String\$)*

Wandelt einen String in einen numerischen Wert um. Der String muss ein Integer-Wert im Dezimalformat sein.

Beispiel

```
Ergebnis = Val("1024") ; Ergebnis wird mit 1024 gefüllt.
```

ValF(String\$)

Ergebnis.f = *ValF(String\$)*

Wandelt einen String in eine Fließkomma-Zahl um. Der String muss ein Fließkomma-Wert im Dezimalformat sein.

Beispiel

```
Ergebnis.f = ValF("10.24")  
; Ergebnis wird mit 10.24 gefüllt.
```

16.4 Sort

Oftmals müssen Elemente sortiert sein, um einfacher oder überhaupt verwendbar zu sein. PureBasic bietet hoch-optimierte Befehle zum Sortieren von Arrays, entweder in aufsteigender oder absteigender Reihenfolge.

Arrays selbst werden mit dem Schlüsselwort *Dim()* deklariert, sehen Sie hierzu im Kapitel 7.6.

SortArray(ArrayName(), Option [, Start, Ende])

SortArray(ArrayName(), Option [, Start, Ende])

Sortiert das angegebene Array 'ArrayName()' entsprechend den übergebenen 'Optionen'. Das Array kann aus den folgenden Standard-Typen bestehen: Byte, Word, Long, String. Ein optionaler 'Start' - 'Ende' Bereich kann angegeben werden.

„Option“ kann folgende Werte annehmen:

0 : Sortiert das Array in aufsteigender Richtung (kleine Zahlen

zuerst)

1 : Sortiert das Array in absteigender Richtung (große Zahlen zuerst)

2 : Sortiert ein String-Array ohne Berücksichtigung der Groß-/Kleinschreibung (a=A, b=B etc..) in aufsteigender Richtung

3 : Sortiert ein String-Array ohne Berücksichtigung der Groß-/Kleinschreibung (a=A, b=B etc..) in absteigender Richtung

Hinweis: Ist ein Array nicht vollständig gefüllt, dann werden 'Null'-Elemente an den Anfang des Arrays (bei aufsteigender Sortier-Richtung) bzw. an das Ende des Arrays (bei absteigender Sortier-Richtung) sortiert.

16.5 LinkedList

Die verknüpften Listen (,Linked Lists') sind Objekte, welche dynamisch entsprechend Ihrem Bedarf reserviert werden. Sie sind eine Liste von Elementen, wobei jedes Element vollkommen unabhängig von einem anderen ist. Sie können beliebige Elemente hinzufügen, Elemente an der von Ihnen gewünschten Stelle einfügen, einige andere löschen und vieles mehr. Diese Art des Datenmanagements wird sehr oft benutzt, da sie die beste Möglichkeit darstellt, mit Daten - deren Anzahl und Größe nicht bekannt ist - umzugehen.

Definiert werden die verknüpften Listen mit dem Schlüsselwort *NewList*, siehe hierzu Kapitel 7.7 dieses Handbuchs.

AddElement(LinkedList())

Fügt ein neues Listenelement nach der aktuellen Position ein. Dieses neue Element wird auch das aktuelle Element der Liste.

ChangeCurrentElement(LinkedList(), *NewElement)

Ändert das aktuelle Element der angegebenen Liste auf das angegebene *NewElement. Das *NewElement muss ein Zeiger (Pointer) auf ein anderes in der Liste existierendes Element sein. Diese Funktion ist sehr nützlich, um sich ein Element zu merken und es nach der Verarbeitung anderer Befehle wieder aufzurufen. Die Funktion sollte mit Vorsicht und nur von fortgeschrittenen Usern benutzt werden.

Beispiel

```
*Old_Element = @mylist() ; Ermittelt die Adresse des aktuellen Elements
ResetList(mylist()) ; Durchführen einer Suchschleife nach allen
While NextItem(mylist()) ; Elementen mit Name "John" und Änderung dieser in "J"
If mylist() = "John"
mylist() = "J"
EndIf
Wend
ChangeCurrentElement(mylist(), *Old_Element) ; Wiederherstellen unseres Elements vor der Suche
```

ClearList(LinkedList())

Löscht alle Elemente in dieser Liste und gibt deren Speicherplatz frei. Nach diesem Aufruf ist die Liste noch benutzbar, es befinden sich aber keine Elemente mehr in der Liste.

CountList(LinkedList())

Anzahl = *CountList(LinkedList())*

Zählt, wieviele Elemente in der verknüpften Liste vorhanden sind. Es ändert nicht das aktuelle Listenelement.

DeleteElement(LinkedList())

Entfernt das aktuelle Element aus der Liste. Nach dem Aufruf dieser Funktion wird das - auf das gelöschte Element folgende - Element das neue aktuelle Element. Haben Sie das letzte Element in der Liste gelöscht, wird das aktuelle Element auf 0 gesetzt.

FirstElement(LinkedList())

Ändert das aktuelle Listenelement auf das erste Listenelement.

InsertElement(LinkedList())

Fügt ein neues Element vor der aktuellen Position ein. Dieses neue Element wird das aktuelle Element der Liste.

LastElement(LinkedList())

Ändert das aktuelle Listenelement auf das letzte Listenelement.

ListIndex(LinkedList())

Position = *ListIndex(LinkedList())*

Gibt die Position des aktuellen Listenelements zurück, das erste Element in der Liste befindet sich dabei an Position 1.

NextElement(LinkedList())

Ergebnis = *NextElement(LinkedList())*

Ändert das aktuelle Element auf das nächste Listenelement und gibt dessen Adresse zurück, oder 0 wenn keine weiteren Elemente existieren.

PreviousElement(LinkedList())

Ergebnis = *PreviousElement(LinkedList())*

Ändert das aktuelle Element auf das vorhergehende Element und gibt dessen Adresse zurück, oder 0 wenn das aktuelle Element bereits das erste Element war oder gar keine Elemente in der Liste existieren.

ResetList(LinkedList())

Veranlasst ein Rücksetzen der "Linked List" vor das erste Element. Das bedeutet, dass kein Element mehr gültig ist. Dies ist sehr nützlich, um anschließend alle Elemente mittels *NextElement()* abzuarbeiten.

Beispiel

```
ResetList(Friends())  
While NextElement(Friends())  
    ...  
Wend
```

SelectElement(LinkedList(), Position)

Wechselt das aktuelle Listen-Element mit dem Element an der angege-

benen 'Position'. Das erste Element befindet sich an Position 0.

16.6 Memory

Manchmal ist es sehr nützlich, Zugriff auf den Systemspeicher (RAM) zu haben, um einige zeitintensive Operationen zu verarbeiten und sie zu beschleunigen. Diese Library ermöglicht das Reservieren einer beliebigen Anzahl an Speicherbereichen (Memory-Buffer) und deren direkte Benutzung in PureBasic.

AllocateMemory(#Memory, Größe, Flags)

Ergebnis = *AllocateMemory(#Memory, Größe, Flags)*

Reserviert einen zusammenhängenden Speicherbereich mit der angegebenen Größe. Wenn der angeforderte Speicher verfügbar ist, enthält ‚Ergebnis‘ die Startadresse des Speicherbereichs, andernfalls ist ‚Ergebnis‘ gleich 0. ‚Flags‘ müssen zurzeit 0 sein, diese werden später unterstützt. Wenn vorher eine weitere Speicherbank mit der gleichen #Memory Nummer reserviert war, wird diese automatisch freigegeben. Hinweis: Alle reservierten Speicherbereiche werden automatisch freigegeben, wenn das Programm beendet wird.

CompareMemory(MemoryID1, MemoryID2, Länge)

Ergebnis = *CompareMemory(MemoryID1, MemoryID2, Länge)*

Vergleicht zwei Speicherbereiche und gibt 1 zurück, wenn diese gleich sind. Andernfalls 0, wenn sie nicht übereinstimmen. Um eine gültige MemoryID zu erhalten, benutzen Sie einfach *UseMemory()* oder *MemoryID()*.

CompareMemoryString(*String1, *String2 [, Modus [, Länge]])

Ergebnis = *CompareMemoryString(*String1, *String2 [, Modus [, Länge]])*

Nur für fortgeschrittene Anwender. Vergleicht zwei Strings an den angegebenen Speicheradressen. Die Strings müssen entweder "null-terminiert" sein oder der Parameter 'Länge' muss die Anzahl der zu vergleichenden Zeichen angeben. Diese Funktion kann sehr nützlich sein, um aus Performance-Gründen die Strings im Speicherbuffer zu verglei-

chen.

„Modus“ kann einer der folgenden Werte sein:

0 : String Vergleich ist "case-sensitive" ($a \neq A$)

1 : String Vergleich ist "case-insensitive" ($a = A$)

„Ergebnis“ kann einer der folgenden Werte sein:

0 : wenn String1 gleich String2 ist

-1 : wenn String1 kleiner als String2 ist

1 : wenn String1 größer als String2 ist

CopyMemory(SourceMemoryID, DestinationMemoryID, Länge)

Kopiert einen Speicherbereich von 'SourceMemoryID' (der Ausgangs-Speicheradresse) mit der angegebenen Länge zur 'DestinationMemoryID' (der Ziel-Speicheradresse).

CopyMemoryString(*String [, @*DestinationMemoryID])

Nur für fortgeschrittene Anwender. Kopiert den String an der angegebenen Adresse zur optionalen Zieladresse '@*DestinationMemoryID', oder - falls diese weggelassen wurde - an das Ende des aktuellen Speicherbuffers. Wenn '*DestinationMemoryID' angegeben wird, wird der interne Buffer Zeiger auf den neuen Wert zurückgesetzt. Der interne Zeiger wird nach einem Kopieren automatisch aktualisiert. Damit ist es sehr gut möglich, eine ganze Menge Strings zu kopieren und währenddessen die Kontrolle über den Zeiger zu behalten. Dieser Befehl ist speziell optimiert, um extrem schnell einen Text-Speicherbuffer zu manipulieren.

Beispiel

```
*Pointer = AllocateMemory(0, 1000, 0)
CopyMemoryString("Hello", @*Pointer)
CopyMemoryString(" World")
; Dieser String wird einfach nach "Hello" im
; Speicherbuffer eingefügt.

*Pointer-2
; Setzt den Zeiger um 2 Zeichen zurück
; (auf das 'l' von 'World') ...
```

FreeMemory(#Memory)

Gibt den zuvor mittels *AllocateMemory()* reservierten Speicher frei. Wird als Parameter '#Memory' gleich -1 übergeben, werden alle zuvor reservierten Speicherbereiche freigegeben.

MemoryID()

ID = *MemoryID()*

Ermittelt die ID der aktuellen Speicherbank. Dies ist die Startadresse des Speicherbereichs.

MemoryStringLength(*String)

Länge = *MemoryStringLength(*String)*

Nur für fortgeschrittene Anwender. Gibt die Länge des angegebenen null-terminierten Strings zurück.

UseMemory(#Memory)

Ergebnis = *UseMemory(#Memory)*

Ändert die aktuelle Speicherbank in die angegebene neue. Es wird die Startadresse der neuen Speicherbank zurückgegeben.

ReAllocateMemory(#Memory, Größe)

Ergebnis = *ReAllocateMemory(#Memory, Größe)*

Reserviert (erneut) einen zusammenhängenden Speicherbereich '#Memory' entsprechend der angegebenen 'Größe'. Der Inhalt des vorherigen Speichers '#Memory' wird behalten.

Hinweis: Alle reservierten Speicherbanken werden am Programmende automatisch freigegeben.

16.7 File

Die Dateien sind die gebräuchlichste Art des Speicherns auf heutigen Computern. Mit PureBasic verwalten Sie diese auf sehr einfachem und optimiertem Weg. Eine beliebige Anzahl Dateien kann gleichzeitig verwendet werden. Diese Bibliothek benutzt gepufferte Funktionen, um die Schreib- und Lesegeschwindigkeit zu erhöhen.

CloseFile(#Datei)

Schließt die angegebene '#Datei', welche damit nicht weiter benutzt werden kann. Beim Schließen einer Datei wird der Buffer effektiv auf Disk abgespeichert. Hinweis: Am Programmende ist PureBasic "klug" genug, um alle offenen Dateien automatisch zu schließen. Deshalb müssen Sie dies nicht unbedingt selbst erledigen.

CreateFile(#Datei, DateiName\$)

Ergebnis = *CreateFile*(#Datei, DateiName\$)

Öffnet eine leere Datei. Existierte die angegebene Datei bereits, wird diese geöffnet und durch eine leere ersetzt! Seien Sie vorsichtig! Ergibt 'Ergebnis' nicht 0, konnte die Datei angelegt werden, andernfalls ist die Erstellung fehlgeschlagen. Dies muss immer getestet werden, da das Ausführen von Operationen auf nicht erstellte Dateien zu schlimmen Abstürzen führt.

Eof(#Datei)

Ergebnis = *Eof*(#Datei)

EOF steht für 'End Of File' (Ende der Datei). Die Funktion gibt einen Wert ungleich 0 zurück, wenn das Ende der angegebenen '#Datei' erreicht wurde, andernfalls wird 0 zurückgegeben.

FileSeek(NeuePosition)

Ändert den Lese/Schreib - Zeiger innerhalb der Datei auf die angegebene 'NeuePosition'.

Loc()

Position = *Loc*()

Gibt die aktuelle Zeiger-Position innerhalb der Datei zurück.

Lof()

Länge = *Lof*()

LOF steht für 'Length Of File' (Länge der Datei). Es gibt die Länge der aktuellen Datei zurück.

OpenFile(#Datei, Dateiname\$)

Ergebnis = OpenFile(#Datei, Dateiname\$)

Öffnet die angegebene Datei oder erstellt eine, falls noch keine existiert. Sie können nun Lese- und Schreibfunktionen in dieser Datei ausführen. Ist ‚Ergebnis‘ nicht 0, wurde die Datei erfolgreich geöffnet, andernfalls konnte die Datei nicht geöffnet werden. Dies muss immer getestet werden, da das Ausführen von Operationen auf nicht erstellte Dateien zu schlimmen Abstürzen führt.

ReadByte()

Zahl.b = ReadByte()

Liest ein Byte aus der aktuell geöffneten Datei ein.

ReadData(*MemoryBuffer, LengthToRead)

Liest den Inhalt der aktuellen Datei mit der angegebenen Länge 'LengthToRead' in den angegebenen Speicherbereich ‚*MemoryBuffer‘.

ReadFile(#Datei, Dateiname\$)

Ergebnis = ReadFile(#Datei, Dateiname\$)

Öffnet eine existierende Datei 'Dateiname\$' ausschließlich für Lese-Operationen. Ist ‚Ergebnis‘ nicht 0, wurde die Datei erfolgreich geöffnet, andernfalls konnte die Datei nicht geöffnet werden. Dies muss immer getestet werden, da das Ausführen von Operationen auf nicht erstellte Dateien zu schlimmen Abstürzen führt.

ReadLong()

Zahl.l = ReadLong()

Liest einen Long-Wert (4 Bytes) aus der aktuell geöffneten Datei ein.

ReadString()

Text\$ = *ReadString()*

Liest einen String aus der aktuell geöffneten Datei ein.

ReadWord()

Zahl.w = *ReadWord()*

Liest einen Word-Wert (2 Byte) aus der aktuell geöffneten Datei ein.

UseFile(#Datei)

Ändert die aktuell zu benutzende Datei auf die angegebene ,#Datei'.

WriteByte(Number.b)

Schreibt einen Byte-Wert in die aktuelle Datei. Die Datei muss mit Schreib-Unterstützung geöffnet worden sein (d.h. nicht mit *ReadFile()*).

WriteLong(Number.w)

Schreibt einen Long-Wert in die aktuelle Datei. Die Datei muss mit Schreib-Unterstützung geöffnet worden sein (d.h. nicht mit *ReadFile()*).

WriteData(*MemoryBuffer, LengthToWrite)

Schreibt den Inhalt des angegebenen Speicherbereichs '*MemoryBuffer' mit einer Länge von 'LengthToWrite' in die aktuelle Datei.

WriteString(Text\$)

Schreibt einen String in die aktuelle Datei. Die Datei muss mit Schreib-Unterstützung geöffnet worden sein (d.h. nicht mit *ReadFile()*).

WriteStringN(Text\$)

Schreibt einen String in die aktuelle Datei und fügt ein 'end of line' Zeichen (für Zeilenumbruch) hinzu. Die Datei muss mit Schreib-Unterstützung geöffnet worden sein (d.h. nicht mit *ReadFile()*).

WriteWord(Number.w)

Schreibt einen Word-Wert in die aktuelle Datei. Die Datei muss mit Schreib-Unterstützung geöffnet worden sein (d.h. nicht mit *ReadFile()*).

16.8 FileSystem

FileSystem ist ein Oberbegriff, der alle fortgeschrittenen Funktionen rund um die Dateien beinhaltet. Zum Beispiel ist es möglich, einen Verzeichnisinhalt einzulesen, ein neues Verzeichnis zu erstellen und vieles mehr...

CopyFile(Ausgangsdatei\$, Zieldatei\$)

Ergebnis = *CopyFile(Ausgangsdatei\$, Zieldatei\$)*

Kopiert die 'Ausgangsdatei\$' über die 'Zieldatei\$'. Warnung: Wenn die Zieldatei bereits existiert, wird sie automatisch gelöscht. Ergibt 'Ergebnis' gleich 0, konnte die Datei nicht kopiert werden.

CreateDirectory(VerzeichnisName\$)

Ergebnis = *CreateDirectory(VerzeichnisName\$)*

Erstellt ein neues Verzeichnis. Ergibt 'Ergebnis' gleich 0, konnte das Verzeichnis nicht erstellt werden.

DeleteFile(DateiName\$)

Ergebnis = *DeleteFile(DateiName\$)*

Löscht die angegebene Datei. Ergibt 'Ergebnis' gleich 0, konnte die Datei nicht gelöscht werden.

DirectoryEntryAttributes()

Attribute = *DirectoryEntryAttributes()*

Gibt die Dateiattribute des aktuellen Eintrags im Verzeichnis zurück, welches mit den Befehlen *ExamineDirectory()* und *NextDirectoryEntry()* aufgelistet wird. Die Attribute sind eine Kombination der folgenden Werte:

#PB_FileSystem_Hidden	Datei ist versteckt
#PB_FileSystem_Archive	Datei wurde archiviert und seit dem letzten Mal nicht geändert
#PB_FileSystem_Compressed	Datei ist komprimiert
#PB_FileSystem_Normal	Normale Attribute
#PB_FileSystem_ReadOnly	Datei ist im "ReadOnly" Modus (schreibgeschützt)
#PB_FileSystem_System	Datei ist eine Systemdatei

Um zu testen, ob ein Attribut gesetzt ist, benutzen Sie einfach '&' (logisches 'AND', zu deutsch: und) und die Attribut-Konstanten:

```
[...]  
FileAttributes = DirectoryEntryAttributes()  
If FileAttributes & #PB_FileSystem_Hidden  
    Debug "Diese Datei ist versteckt !"  
EndIf
```

DirectoryEntryName()

DateiName\$ = *DirectoryEntryName()*

Gibt den Namen des aktuellen Eintrags im mittels der Befehle *ExamineDirectory()* und *NextDirectoryEntry()* aufgelisteten Verzeichnis zurück.

DirectoryEntrySize()

Größe = *DirectoryEntrySize()*

Gibt die Größe des aktuellen Eintrags im mittels der Befehle *ExamineDirectory()* und *NextDirectoryEntry()* aufgelisteten Verzeichnis zurück.

ExamineDirectory(#Verzeichnis, VerzeichnisName\$, Pattern\$)

Ergebnis = *ExamineDirectory*(#Verzeichnis, VerzeichnisName\$, Pattern\$)

Beginnt die Prüfung des angegebenen Verzeichnisses zur weiteren Bearbeitung mittels der Befehle *NextDirectoryEntry()* und *DirectoryEntryName()*. Der 'Pattern\$' gibt an, welche Dateien aufgelistet werden sollen. ‚#Verzeichnis‘ ist die numerische ID und kann mittels *UseDirectory()* verwendet werden, um das Verzeichnis zu wechseln. Zum Beispiel: „*.“ oder „“ werden alle Dateien im Verzeichnis auflisten. „*.exe“ wird nur .exe (ausführbare) Dateien auflisten. Ergibt ‚Ergebnis‘ gleich 0, kann das Verzeichnis nicht untersucht werden.

FileSize(Dateiname\$)

Größe = *FileSize*(Dateiname\$)

Gibt die Größe (in Bytes) der angegebenen Datei zurück.

Besondere Rückgabewerte-Werte:

- 1 : Datei wurde nicht gefunden.
- 2 : Datei ist ein Verzeichnis.

NextDirectoryEntry()

Ergebnis = *NextDirectoryEntry()*

Dieser Befehl muss nach einem *ExamineDirectory()* aufgerufen werden. Er wird Schritt für Schritt durch das Verzeichnis gehen und dessen Inhalt auflisten. Der jeweilige Name des aktuellen Eintrags kann mittels dem *DirectoryEntryName()* Befehl ermittelt werden. ‚Ergebnis‘ kann die folgenden Werte annehmen:

- 0 : keine weiteren Einträge im Verzeichnis
- 1 : dieser Eintrag ist eine Datei
- 2 : dieser Eintrag ist ein Verzeichnis

RenameFile(AlterDateiName\$, NeuerDateiName\$)

Ergebnis = *RenameFile(AlterDateiName\$, NeuerDateiName\$)*

Nennt die alte Datei in die neue Datei um. Ergibt , Ergebnis' gleich 0, ist das Umbenennen fehlgeschlagen.

UseDirectory(#Verzeichnis)

Wechselt das aktuelle Verzeichnis auf das neu angegebene '#Verzeichnis'. Dieser Befehl ist nur nützlich in Verbindung mit *ExamineDirectory()*.

16.9 Misc

Verschiedene Befehle, die sich keinem bestimmten Thema zuordnen lassen.

Delay(Zeit)

Wartet die angegebene Zeit. Diese wird in Millisekunden angegeben. Das Programm wird komplett angehalten, bis die angegebene Zeit abgelaufen ist.

Red(Farbe)

Ergebnis = *Red(Farbe)*

Gibt den Rot-Wert einer 24 Bit RGB-Farbe zurück (zwischen 0 und 255).

Green(Farbe)

Ergebnis = *Green(Farbe)*

Gibt den Grün-Wert einer 24 Bit RGB-Farbe zurück (zwischen 0 und 255).

Blue(Farbe)

Ergebnis = Blue(Farbe)

Gibt den Blau-Wert einer 24 Bit RGB-Farbe zurück (zwischen 0 und 255).

RGB(Rot, Grün, Blau)

Farbe = RGB(Rot, Grün, Blau)

Gibt den 24 Bit Farb-Wert entsprechend den Rot-, Grün- und Blau-Werten zurück.

GetExtensionPart(DateiPfad\$)

Endung\$ = GetExtensionPart(DateiPfad\$)

Ermittelt die Dateiendung aus einem kompletten Dateipfad. Zum Beispiel, wenn der volle Pfad "C:.exe" lautet, wird das Resultat "exe" ergeben.

GetFilePart(DateiPfad\$)

DateiName\$ = GetFilePart(DateiPfad\$)

Ermittelt den Dateinamen aus einem kompletten Dateipfad. Zum Beispiel, wenn der volle Pfad "C:.exe" lautet, wird das Resultat "PB.exe" ergeben.

GetPathPart(DateiPfad\$)

Pfad\$ = GetPathPart(DateiPfad\$)

Ermittelt das Verzeichnis aus einem kompletten Dateipfad. Zum Beispiel, wenn der volle Pfad "C:.exe" lautet, wird das Resultat "C:" ergeben.

ProgramParameter()

Parameter\$ = ProgramParameter()

Ermittelt den nächsten Parameter-String, der dem Executable beim Start

übergeben wurde. Zum Beispiel

```
MyProgram.exe MyText.txt /FAST "Special Mode"
```

Das erste Mal, wenn `ProgramParameter()` aufgerufen wird, gibt es "MyText.txt" zurück, beim zweiten Mal "/FAST" und beim dritten Mal "Special Mode". Sind keine (weiteren) Parameter vorhanden, wird ein leerer String zurückgegeben.

RunProgram(DateiName\$, Parameter\$, Flags)

`Ergebnis = RunProgram(DateiName$, Parameter$, Flags)`

Startet ein externes Programm asynchron (die laufende Programmausführung wird nicht angehalten). Der 'DateiName\$' sollte den kompletten Pfad enthalten. Wenn der Rückgabewert 'Ergebnis' gleich 0 ergibt, konnte das Programm nicht gestartet werden.

Mögliche Werte als Flags:

- 0 : Keine Flags
- 1 : Wartet bis das gestartete Programm beendet wird
- 2 : Startet das Programm im unsichtbaren Modus

Um mehrere Optionen gleichzeitig zu benutzen, müssen Sie den ',' (OR) Operator verwenden. Ein Beispiel für ein unsichtbares Programm und Warten bis dieses beendet ist:

```
RunProgram(DateiName$, Parameter$, 1 | 2)
```

Random(Maximum)

`Ergebnis = Random(Maximum)`

Ergibt eine Zufallszahl zwischen 0 und dem angegebenen Maximalwert. Zusätzlich kann `RandomSeed()` benutzt werden, um den aktuellen Ausgangswert der Zufallszahl zu ändern. Hinweis: Bei jedem neuen Programmstart wird automatisch einer neuer Ausgangswert generiert. `RandomSeed()` ist daher nur nützlich, wenn der Programmierer immer die gleichen Zufallszahlen in der gleichen Reihenfolge wünscht.

RandomSeed(Wert)

Ändert den aktuellen Ausgangswert für mit `Random()` zurückgegebene

Zufallszahlen. Hinweis: Bei jedem neuen Programmstart wird automatisch ein neuer Ausgangswert generiert. *RandomSeed()* ist daher nur nützlich, wenn der Programmierer immer die gleichen Zufallszahlen in der gleichen Reihenfolge wünscht.

PeekBWL(*MemoryBuffer)

Wert.l = *PeekBWL(*MemoryBuffer)*

Nur für fortgeschrittene Programmierer. *PeekB()*, *PeekW()*, *PeekL()* lesen jeweils einen Byte-Wert (1 Byte), einen Word-Wert (2 Bytes) oder einen Long-Wert (4 Bytes) von der angegebenen Speicheradresse aus.

PeekF(*MemoryBuffer)

Wert.f = *PeekF(*MemoryBuffer)*

Nur für fortgeschrittene Programmierer. *PeekF()* liest einen Float-Wert (4 Bytes) von der angegebenen Speicheradresse.

PeekS(*MemoryBuffer [, Länge])

Text\$ = *PeekS(*MemoryBuffer [, Länge])*

Nur für fortgeschrittene Programmierer. Sehr nützlich, um einen String an der angegebenen Speicheradresse auszulesen. Der String muss mit einem '0' Zeichen enden, andernfalls wird der Speicher solange ausgelesen, bis eine '0' auftritt. Ein optionaler Parameter 'Länge' kann angegeben werden.

PokeBWL(*MemoryBuffer, Number.L)

Nur für fortgeschrittene Programmierer. *PokeB()*, *PokeW()*, *PokeL()* schreiben jeweils einen Byte-Wert (1 Byte), einen Word-Wert (2 Bytes) oder einen Long-Wert (4 Bytes) an die angegebene Speicheradresse.

PokeF(*MemoryBuffer, Number.f)

Nur für fortgeschrittene Programmierer. *PokeF()* schreibt eine Float-Zahl (4 Bytes) an die angegebene Speicheradresse.

PokeS(*MemoryBuffer, Text\$ [, Länge])

Nur für fortgeschrittene Programmierer. Schreibt einen String (einschließlich einer abschließenden '0') an die angegebene Speicheradresse. Ein optionaler Parameter 'Länge' kann angegeben werden.

16.10 Palette

Paletten können nur auf Bildschirmen mit 256 Farben (oder weniger) genutzt werden und erlauben, die angezeigten Farben zu 'mappen' (einzuteilen). Es ist möglich, jeder Farbe (benannt mit 0 bis 255) eine bestimmte RGB-Farbe zuzuweisen, welche aus einer 24 Bit-Palette (16 Millionen Farben) ausgewählt wurde. Der Vorteil hierbei ist groß, da alle Pixel einer Farbe einfach durch mit Änderung ihrer zugewiesenen Farbe verändert werden können. Dies erlaubt sehr schnelles Paletten-Cycling (Farbwechsel), Fade-In (Einblenden), Fade-Out (Ausblenden) und andere wohlbekannte Effekte.

InitPalette(#NumPaletteMax)

Ergebnis = *InitPalette(#NumPaletteMax)*

Initialisiert die gesamte Paletten-Programmierungsumgebung zur späteren Benutzung. Sie müssen diese Funktion am Anfang Ihres Sourcecodes aufrufen, wenn Sie die Paletten-Befehle benutzen möchten. Ist 'Ergebnis' gleich 0, ist das Initialisieren fehlgeschlagen und keiner der Paletten-Befehle kann benutzt werden. Falls unter Windows das Initialisieren fehlschlägt, liegt dies möglicherweise am nicht verfügbaren DirectX 7 (oder DirectX 3.0 mit NT4.0 Kompatibilität).

CreatePalette(#Palette)

Ergebnis = *CreatePalette(#Palette)*

Erstellt eine neue, leere 256 Farben Palette. Die Palette wird mit der Farbe 0 initialisiert. Ist 'Ergebnis' gleich 0, ist die Erstellung der #Palette fehlgeschlagen. Ein vorher bestehendes Palettenobjekt mit der ID '#Palette' wird automatisch freigegeben.

DisplayPalette(#Palette)

Zeigt die angegebene '#Palette', welche zuvor mit *CreatePalette()* oder *LoadPalette()* erstellt wurde, auf dem aktuellen Bildschirm an.

FreePalette(#Palette)

Gibt den von der angegebenen #Palette reservierten Speicher frei.

GetPaletteColor(Index)

Farbe = *GetPaletteColor(Index)*

Gibt die RGB-Farbe am angegebenen Index der aktuellen Palette zurück. Um die Rot-, Grün- und Blau-Bestandteile der Farbe zu erhalten, benutzen Sie die Befehle *Red()*, *Green()* und *Blue()*.

LoadPalette(#Palette, Dateiname\$)

Ergebnis = *LoadPalette(#Palette, Dateiname\$)*

Lädt eine Palette aus einer Standard BMP Datei (ausschließlich 256 Farben-Bilder). Die Palette wird mit den in der Datei gefundenen Werten initialisiert. Wurde die Palette korrekt geladen, ist der Rückgabewert ungleich 0, andernfalls trat ein Fehler auf.

SetPaletteColor(Index, Farbe)

SetPaletteColor(Index, Farbe)

Ändert die RGB-Farbe am angegebenen Index der aktuellen Palette. Um eine gültige RGB-Farbe zu erhalten, benutzen Sie den *RGB()* Befehl oder die folgende Formel (schneller):

```
RGBColorValue = Rot + Grün << 8 + Blau << 16
```

UsePalette(#Palette)

Ändert die aktuell zu benutzende #Palette.

16.11 Image

Images (Bilder) sind grafische Objekte, welche in einem Fenster oder in

verschiedenen Gadgets dargestellt werden können. PureBasic unterstützt derzeit die BMP und Icon (.ico) Bild-Typen.

CatchImage(#Image, Speicheradresse)

Ergebnis = *CatchImage*(#Image, Speicheradresse)

Lädt das angegebene Bild '#Image' von der angegebenen Speicheradresse. Das Bild muss im BMP-Format vorliegen (Icons werden nicht unterstützt). Ist das Laden des Bildes nicht möglich, wird als Ergebnis 0 zurückgegeben. Ein geladenes Bild kann mit Hilfe des *FreeImage*() Befehls freigegeben werden. Dieser Befehl ist nützlich in Verbindung mit dem 'IncludeBinary' Schlüsselwort. Damit können Bilder mit in das Executable gepackt werden. Nichtsdestotrotz, benutzen Sie diese Option mit Bedacht, da hiermit mehr Speicher benötigt wird, als wenn die Bild-datei in einer externen Datei gespeichert wird (die Datei befindet sich im Speicher des Executable und im physikalischen Speicher).

Beispiel

```
CatchImage(0, ?Logo)
End
Logo: IncludeBinary "Logo.bmp"
```

CopyImage(#Image1, #Image2)

Erstellt ein neues Bild '#Image2', welches identisch zur Quelle '#Image1' ist.

CreateImage(#Image, Breite, Höhe)

Erstellt ein leeres Bild (Image), welches zum darauf Zeichnen benutzt werden kann. Das Bild-Format wird vom aktuellen Bildschirm-Format abgeleitet. Zum Beispiel kann *DrawImage*() benutzt werden, um dieses Bild auf einem Fenster anzuzeigen.

FreeImage(#Image)

Gibt das angegebene Bild '#Image' und dessen zugehörigen Speicher frei.

GrabImage(#Image1, #Image2, x, y, Breite, Höhe)

Erstellt ein neues '#Image2' aus dem angegebenen Bereich (x, y, Breite, Höhe) der Quelle '#Image1'.

ImageDepth()

Tiefe = *ImageDepth()*

Ermittelt die Tiefe des aktuellen Image. Tiefe gibt die Anzahl an Bitplanes an, die das Bild enthält. Mit anderen Worten, es repräsentiert die Anzahl Farben des Bildes. Beispiel ein 24 Bit Bild (16 Millionen Farben) wird '24' zurückgeben. Ein 256 Farben-Bild wird '8' (8 Bit) zurückgeben. Ein Monochrom-Bild wird '1' zurückgeben.

ImageHeight()

Höhe = *ImageHeight()*

Gibt die Höhe des aktuellen Bildes in Pixel zurück.

ImageID()

ID = *ImageID()*

Gibt die ImageID des aktuellen Bildes zurück.

ImageOutput()

Ermittelt die 'OutputID' des aktuell benutzten Images, um darauf 2D-Zeichen-Operationen ausführen zu können. Hierzu verwenden Sie die PureBasic 2DDrawing Library (siehe *StartDrawing()*).

ImageWidth()

Breite = *ImageWidth()*

Gibt die Breite des aktuellen Bildes in Pixel zurück.

LoadImage(#Image, Dateiname\$)

Ergebnis = *LoadImage(#Image, Dateiname\$)*

Lädt das angegebene Bild. Das Bildformat kann entweder eine BMP oder eine ICON (.ico) Datei sein. Wenn die Funktion fehlschlägt, wird 0 zurückgegeben, andernfalls ist alles in Ordnung.

ResizeImage(#Image, Breite, Höhe)

Verändert die Größe des Bildes '#Image' auf die angegebene Dimension (Breite, Höhe).

SaveImage(#Image, Dateiname\$)

Ergebnis = *SaveImage(#Image, Dateiname\$)*

Speichert das angegebene Bild '#Image' in die angegebene Datei 'Dateiname\$'. Wenn die Funktion fehlschlägt, wird 0 zurückgegeben, andernfalls ist alles in Ordnung.

UseImage(#Image)

Ergebnis = *UseImage(#Image)*

Ändert das aktuelle Bild in das angegebene neue Bild '#Image'. 'Ergebnis' enthält die ImageID des neuen aktuellen Bildes.

16.12 2DDrawing

Das 2D Zeichnen beinhaltet alle 2D Zeichenfunktionen, die Sie in einem sichtbaren Bereich ausführen können. Das Zeichnen einer Linie, einer Box, eines Kreises oder einfach eines Textes gehört zu den 2D Zeichenoperationen.

StartDrawing(OutputID)

Ändert die aktuelle Ausgabe auf den angegebenen Ausgabekanal. Nach dessen Festlegung werden alle Zeichenoperationen auf diesem ausgegeben. Sobald alle Zeichenoperationen abgeschlossen wurden, muss *StopDrawing()* aufgerufen werden.

Eine gültige 'OutputID' kann mit den folgenden Befehlen ermittelt werden:

WindowOutput() Grafiken werden direkt auf dem Fenster gerendert

ScreenOutput() Grafiken werden direkt auf dem Bildschirm gerendert (für Spiele)

SpriteOutput() Grafiken werden direkt auf dem Sprite gerendert (für Spiele)

ImageOutput() Grafiken werden direkt in die Bilddaten gerendert (siehe *CreateImage()*)

PrinterOutput() Grafiken werden direkt auf die Druckerausgabe gerendert

BackColour(Rot, Grün, Blau)

Definiert die standardmäßige Hintergrundfarbe für alle Grafikfunktionen und die Textanzeige. Jede Farbe wird mit ihren Rot-, Grün- und Blau-Anteilen unterstützt. Wie immer liegen diese Werte jeweils zwischen 0 und 255, was bis zu 16,7 Millionen möglicher Farben ergibt.

Box(x, y, Breite, Höhe [, Farbe])

Zeichnet einen ausgefüllten Kasten in der angegebenen Größe in den aktuellen Ausgabekanal. Wenn der optionale Parameter 'Farbe' nicht verwendet wird, benutzt PureBasic die mittels *FrontColour()* festgelegte Zeichenfarbe. *RGB()* kann zum Ermitteln eines gültigen Farbwertes benutzt werden. Der aktuelle Ausgabekanal wird mittels *StartDrawing()* festgelegt.

Circle(x, y, Radius [, Farbe])

Zeichnet im aktuellen Ausgabekanal einen ausgefüllten Kreis an der Position x,y mit der Größe des angegebenen Radius. Wenn der optionale Parameter 'Farbe' nicht verwendet wird, benutzt PureBasic die mittels *FrontColour()* festgelegte Zeichenfarbe. *RGB()* kann zum Ermitteln eines gültigen Farbwertes benutzt werden. Der aktuelle Ausgabekanal wird mittels *StartDrawing()* festgelegt.

DrawImage(ImageID, x, y)

Zeichnet das angegebene Bild 'ImageID' an die Position x,y. Die 'ImageID' kann einfach mittels dem *UseImage()* Befehl aus der Image

Library ermittelt werden.

DrawingFont(FontID())

Ändert den aktuellen Zeichensatz auf die angegebene FontID. Alle neuen Texte werden mit dem neu eingestellten Zeichensatz dargestellt. Die 'FontID' erhalten Sie einfach mit dem *FontID()* Befehl aus der Font Bibliothek. Der Zeichensatz muss vor dieser Funktion mittels *LoadFont()* geladen werden.

DrawingMode(Modus)

Ändert den Zeichenmodus für die Text- und Grafikausgabe.
'Modus' kann eine Kombination der folgenden Werte sein:
0: Standard-Modus: Text wird mit Hintergrund angezeigt, Grafikflächen werden ausgefüllt.
1: Setzt den Text-Hintergrund auf transparent
2: Einschalten des XOR Modus (alle Grafiken werden mit dem aktuellen Hintergrund nach dem 'XOR'-Modus verschmolzen)
4: Einschalten des 'Outline' Modus. Kreise, Boxen etc... werden nur mit Umrissen gezeichnet und nicht mehr ausgefüllt.
Um mehrere Modi gleichzeitig nutzen können, benutzen Sie den '|' (OR) Operator.
Ein Beispiel für 'Outline' Flächen im XoR-Modus: *DrawingMode(2 | 4)*

DrawText(Text\$)

Stellt den angegebenen String auf dem aktuellen Ausgabekanal dar. Der Text kann beliebig mittels *Locate()* positioniert werden. *DrawingMode()* kann benutzt werden, um einen transparenten Hintergrund einzustellen. Dieser Befehl benutzt die mittels *FrontColour()* festgelegte Standard-Farbe. Der aktuelle Ausgabekanal wird mittels *StartDrawing()* definiert.

Ellipse(x, y, RadiusX, RadiusY [, Farbe])

Zeichnet eine ausgefüllte Ellipse an der Position x, y mit der Größe von RadiusX und RadiusY. Wenn der optionale Parameter 'Farbe' nicht verwendet wird, benutzt PureBasic die mittels *FrontColour()* festgelegte Zeichenfarbe. *RGB()* kann zum Ermitteln eines gültigen Farbwertes

benutzt werden. Der aktuelle Ausgabekanal wird mittels *StartDrawing()* festgelegt.

FrontColour(Rot, Grün, Blau)

Definiert die standardmäßige Zeichenfarbe für alle Grafikfunktionen und die Textanzeige. Jede Farbe wird mit ihren Rot-, Grün- und Blau-Anteilen unterstützt. Wie immer liegen diese Werte jeweils zwischen 0 und 255, was bis zu 16,7 Millionen möglicher Farben ergibt.

Line(x, y, Breite, Höhe [, Farbe])

Zeichnet eine Linie mit den angegebenen Dimensionen auf dem aktuellen Ausgabekanal. Wenn der optionale Parameter 'Farbe' nicht verwendet wird, benutzt PureBasic die mittels *FrontColour()* festgelegte Zeichenfarbe. *RGB()* kann zum Ermitteln eines gültigen Farbwertes benutzt werden. Der aktuelle Ausgabekanal wird mittels *StartDrawing()* festgelegt.

LineXY(x1, y1, x2, y2 [, Farbe])

Zeichnet eine Linie von Position x1, y1 nach x2, y2 auf dem aktuellen Ausgabekanal. Wenn der optionale Parameter 'Farbe' nicht verwendet wird, benutzt PureBasic die mittels *FrontColour()* festgelegte Zeichenfarbe. *RGB()* kann zum Ermitteln eines gültigen Farbwertes benutzt werden. Der aktuelle Ausgabekanal wird mittels *StartDrawing()* festgelegt.

Locate(x, y)

Platziert den Text-Cursor an der angegebenen Position. Dieser Befehl ist nur nützlich in Verbindung mit einem nachfolgenden *DrawText()*.

Plot(x, y [, Farbe])

Zeichnet einen Punkt an den angegebenen Koordinaten auf dem aktuellen Ausgabekanal. Wenn der optionale Parameter 'Farbe' nicht verwendet wird, benutzt PureBasic die mittels *FrontColour()* festgelegte Zeichenfarbe. *RGB()* kann zum Ermitteln eines gültigen Farbwertes benutzt

werden. Der aktuelle Ausgabekanal wird mittels *StartDrawing()* festgelegt.

Point(x, y)

Farbe = *Point(x, y)*

Ermittelt die im aktuellen Ausgabekanal an der Position (x, y) benutzte Farbe. Der zurückgegebene Farbwert ist ein 24 Bit - Wert, der kombiniert die Rot-, Grün- und Blau-Werte der Farbe enthält. Die entsprechenden Einzelwerte können mit den Funktionen *Red()*, *Green()* und *Blue()* ausgelesen werden.

StopDrawing()

Sobald alle benötigten Grafik-Operationen durchgeführt wurden, muss dieser Befehl aufgerufen werden, um anderen laufenden Applikationen wieder das Darstellen eigener Grafiken zu ermöglichen. Eine typische Sequenz von Zeichenbefehlen sieht wie folgt aus:

```
StartDrawing(WindowID())
Box(10,10,20,20)
Line(30,50,100,100)
....
StopDrawing()
```

TextLength(Text\$)

Länge = *TextLength(Text\$)*

Ermittelt die Länge (in Pixel) des angegebenen Strings, der auf dem aktuellen Ausgabekanal mittels des aktuellen Zeichensatzes dargestellt wird. Diese Funktion ist sehr nützlich, um die tatsächliche Länge eines mit einem nicht-proportionalen Zeichensatz dargestellten Strings zu ermitteln.

16.13 Sound

Das PureBasic Soundsystem ermöglicht auf einfachem Weg Sound innerhalb von Applikationen oder Spielen zu integrieren. Es benutzt spezielle Funktionen, um maximale Geschwindigkeit auf der verfügbaren Hardware zu erhalten. Es benötigt mindestens DirectX 7 (Windows

9x/2000) oder DirectX 3 bei Benutzung des Windows NT 4.0 Kompatibilitätsmodus.

InitSound()

Ergebnis = *InitSound()*

Initialisiert die Sound-Programmierungsumgebung. Ergibt der Rückgabewert ‚Ergebnis‘ gleich 0, kann kein Sound auf diesem Computer abgespielt werden oder die Soundausgabe (Sound-Device) ist beschäftigt. Diese Funktion muss immer vor allen anderen Soundbefehlen aufgerufen und ihr Ergebnis überprüft werden. Wenn die Initialisierung der Sound-Umgebung fehlschlägt, ist es absolut notwendig, alle weiteren Aufrufe von Soundbefehlen zu deaktivieren.

Mögliche Fehlerursachen unter MS Windows: DirectX 7 kann nicht aufgerufen werden oder es ist keine Soundkarte verfügbar. Wenn Ihr Programm unter NT4.0 laufen soll, markieren sie unbedingt 'NT4.0 kompatibles Executable' in den Compiler-Optionen.

CatchSound(#Sound, Speicheradresse)

Ergebnis = *CatchSound(#Sound, Speicheradresse)*

Lädt einen WAV Sound, der sich an der angegebenen Speicheradresse befindet. Ist das ‚Ergebnis‘ ungleich 0, wurde der Sound korrekt geladen und ist bereit zum Abspielen mit *PlaySound()*, andernfalls ist das Laden fehlgeschlagen. Der Sound muss im Standard WAVE Format (.wav) vorliegen. Sounds werden in 16 Bit und in 8 Bit sowie in jeder Frequenz unterstützt.

FreeSound(#Sound)

Hält den zuvor mit *LoadSound()* oder *CatchSound()* geladenen #Sound an und entfernt ihn aus dem Speicher. Sobald ein Sound freigegeben wurde, kann er nicht mehr abgespielt werden.

LoadSound(#Sound, Dateiname\$)

Ergebnis = *LoadSound(#Sound, Dateiname\$)*

Lädt den angegebenen Sound in den Speicher. Ist ‚Ergebnis‘ ungleich 0,

wurde der Sound korrekt geladen und bereit zum Abspielen mittels *PlaySound()*. Andernfalls ist das Laden des Sounds fehlgeschlagen. Der Sound muss im Standard WAVE Format (.wav) vorliegen. Sounds werden in 16 Bit und in 8 Bit sowie in jeder Frequenz unterstützt.

PlaySound(#Sound [, Modus])

Startet das Abspielen des angegebenen #Sounds. 'Modus' ist optional und kann folgenden Wert annehmen:

- 0 : Spielt den Sound nur einmal (Standard-Wert, wenn 'Modus' weggelassen wird)
- 1 : Spielt den Sound fortwährend (Schleife)

SoundFrequency(#Sound, Frequenz)

Legt in Echtzeit die neue Frequenz (in Hz) für den #Sound fest. Der Frequenz-Wert wird für den #Sound gespeichert, deshalb ist es nicht nötig, diesen jedesmal aufzurufen. Die gültigen Werte reichen von 1000 Hz bis 100000 Hz.

SoundPan(#Sound, Pan)

Legt in Echtzeit die Bilanz des #Sounds fest. Der Bilanz-Wert 'Pan' wird für den #Sound gespeichert, deshalb ist es nicht nötig, diesen jedesmal aufzurufen. Die Bilanz ist eine Möglichkeit, das Abspielen eines Sounds auf einem Stereo-Equipment zu beeinflussen. Die gültigen Werte reichen von -100 (voll links) bis 100 (voll rechts). Ist die Bilanz gleich 0, wird der Sound gleichmäßig auf linkem und rechtem Lautsprecher abgespielt.

SoundVolume(#Sound, Volume)

Legt in Echtzeit den neuen Lautstärke-Wert 'Volume' fest. Der Lautstärke-Wert wird für den #Sound gespeichert, deshalb ist es nicht nötig, diesen jedesmal aufzurufen. Die gültigen Werte reichen von 0 (keine Lautstärke) bis 100 (volle Lautstärke).

StopSound(#Sound)

Stoppt den angegebenen #Sound (sofern einer abgespielt wird). Ist der #Sound Wert gleich -1, dann werden alle gerade abgespielten Sounds gestoppt.

16.14 Module

PureBasic kann Standard Musik-Module abspielen, um während eines Spiels oder einer Applikation eine hübsche Hintergrundmusik zu haben. Die Module sind wohlbekannt von den Demo-Makern, da sie eine rationelle Möglichkeit darstellen, Musik auf dem Computer zu erstellen. Die zum Erstellen der Module benutzten Werkzeuge werden 'Tracker' (ProTracker, FastTracker, ImpulseTracker...) genannt. Der Vorteil von Modulen gegenüber .wav/mp3 Dateien sind ihre sehr geringe Größe, eine prinzipiell endlose Länge, die sehr schnellen Abspielroutinen, der mögliche Sprung zu einem bestimmten Musikabschnitt - passend zur gerade laufenden Action auf dem Bildschirm, ect... Es ist natürlich möglich, Standard-Sound und Modul-Sound zu mischen, um beide gleichzeitig abspielen zu können.

Unter Windows wird das MIDAS Soundsystem benutzt, um ein perfektes Abspielen und Unterstützung für verschiedene Tracker sicherzustellen. Vergessen Sie nicht, die midas11.dll in ihr endgültiges Programmpaket zu integrieren, da sie keine Standard DLL ist. Falls vorhanden wird DirectX benutzt, andernfalls die Standard Soundausgabe.

InitModule(#Module)

Ergebnis = *InitModule(#Module)*

Initialisiert die gesamte Modul Programmierungsumgebung. Wenn 'Result' 0 ergibt, kann kein Modul auf diesem Computer abgespielt werden oder die Soundausgabe (sound device) ist gerade beschäftigt. Diese Funktion muss immer aufgerufen werden, bevor irgendein anderer Sound-Befehl benutzt wird, und es sollte auch das zurückgegebene Ergebnis überprüft werden. Wenn das Initialisieren der Modul-Umgebung fehlschlägt, ist es absolut notwendig, alle weiteren Aufrufe von Modul-Befehlen zu deaktivieren.

Mögliche Fehlerursachen unter MS Windows: Midas11.dll wurde nicht gefunden oder es ist keine Soundkarte verfügbar. Wenn Ihr Programm unter NT4.0 laufen soll, markieren sie unbedingt 'NT4.0 compatible Executable' in den Compiler-Optionen. WICHTIG: Der Befehl *InitSound()* muss vor dem Benutzen von Befehlen aus der Module Library

aufgerufen werden.

FreeModule(#Module)

Das angegebene Modul, welches zuvor mittels dem *LoadModule()* Befehl geladen wurde, wird angehalten und aus dem Speicher entfernt. Sobald ein Modul freigegeben wurde, kann es nicht mehr abgespielt werden.

GetModulePosition()

Position = *GetModulePosition()*

Gibt die aktuelle Pattern-Position des laufenden Modules zurück. Der erste Pattern beginnt bei 0.

GetModuleRow()

Reihe = *GetModuleRow()*

Gibt die Reihen-Position in dem geraden abgespielten Pattern zurück. Die erste Reihe (row) beginnt bei 0.

LoadModule(#Module, "DateiName")

Ergebnis = *LoadModule(#Module, "DateiName")*

Lädt das angegebene Modul in den Speicher. Ergibt ‚Ergebnis‘ nicht 0, wurde das Modul korrekt geladen und ist bereit zum Abspielen mittels *PlayModule()*. Andernfalls ist das Laden des Modules fehlgeschlagen. Das Modul kann in den folgenden Formaten vorliegen: ProTracker (4 und 8 Kanäle), FastTracker (bis zu 32 Kanäle, 16 Bit-Qualität) und ImpulseTracker.

PlayModule(#Module)

Beginnt das Abspielen des angegebenen Modules, welches zuvor mittels dem *LoadModule()* Befehl geladen wurde.

SetModulePosition(Position)

Ändert die aktuelle Pattern-Position des laufenden Modules auf die neu angegebene Position. Der erste Pattern beginnt bei 0.

StopModule(#Module)

Stoppt das angegebene Modul (sofern es gerade abgespielt wird).

16.15 CDAudio

Das CD-Audio Format ist eine gute Möglichkeit, um sehr qualitative Musik während eines Spiels oder einer Applikation abzuspielen, die eine Menge Systemressourcen benötigt. Diese Library bietet Zugriff auf alle notwendigen Befehle, die für einfaches Abspielen von CD-Audio in PureBasic Programmen benötigt werden.

InitCDAudio()

Ergebnis = *InitCDAudio()*

Versucht die für die Benutzung der CD-Audio Befehle benötigten Ressourcen zu initialisieren. Dieser Befehl muss vor allen anderen CD-Audio Befehlen aufgerufen werden. Ist 'Ergebnis' gleich 0, ist die Initialisierung fehlgeschlagen oder es sind keine CD-Laufwerke verfügbar. Andernfalls beinhaltet 'Ergebnis' die Anzahl im System verfügbarer CD-Laufwerke.

CDAudioLength()

Länge = *CDAudioLength()*

Gibt die komplette Länge einer Audio-CD in Sekunden zurück.

CDAudioName()

Name\$ = *CDAudioName()*

Gibt den systemabhängigen Namen des Laufwerks zurück, in dem sich die aktuelle Audio-CD befindet. Zum Beispiel wird auf dem Windows-System "D:" zurückgegeben, wenn das Audio-CD Laufwerk dem System unter dem Buchstaben "D:" zugeordnet ist.

CDAudioStatus()

Status = *CDAudioStatus()*

Gibt den aktuellen Status des benutzten Audio-CD Laufwerks zurück.
Der Rückgabewert kann folgende Werte annehmen:

- 1 : CD-Laufwerk nicht bereit (zur Zeit leer oder geöffnet)
- 0 : CD-Laufwerk angehalten (aber die CD ist eingelegt und erkannt)
- >0 : Nummer des Audio-Tracks, der gerade abgespielt wird.

CDAudioTrackLength(TrackNummer)

Länge = *CDAudioTrackLength(TrackNummer)*

Gibt die Länge (in Sekunden) des angegebenen CD-Tracks zurück.

CDAudioTrackSeconds()

Sekunden = *CDAudioTrackSeconds()*

Gibt die Anzahl vergangener Sekunden zurück, seitdem das Abspielen des aktuellen Tracks begonnen hat.

CDAudioTracks()

Tracks = *CDAudioTracks()*

Gibt die Gesamtzahl der Tracks auf der CD zurück, die zum Abspielen verfügbar sind.

EjectCDAudio(Status)

Öffnet oder schließt das aktuelle CD-Laufwerk. Ist Status = 1, wird das CD-Laufwerk geöffnet, andernfalls wird es geschlossen.

PauseCDAudio()

Hält ("pausiert") das Abspielen der Audio-CD an. Das Abspielen kann mittels *ResumeCDAudio()* fortgesetzt werden.

PlayCDAudio(StartTrack, EndTrack)

Beginnt das Abspielen der Audio-CD beim Track 'StartTrack' und beendet es mit dem Ende des Tracks 'EndTrack'. Der erste Track ist mit 1 nummeriert.

ResumeCDAudio()

Setzt das Abspielen der Audio-CD fort, welche vorher mit dem Befehl *PauseCDAudio()* angehalten wurde.

StopCDAudio()

(Endgültiges) Stoppen der laufenden Audio-CD.

UseCDAudio(CDAudioDrive)

Da der Befehl *InitCDAudio()* erlaubt, die Anzahl der im System verfügbaren Audio-CD Laufwerke zu ermitteln, wird *UseCDAudio()* benutzt, um das aktuelle CD-Laufwerk zu wechseln. So ist das gleichzeitige Abspielen mehrerer CD's möglich.

16.16 Movie

PureBasic bietet sehr einfache aber trotzdem mächtige Befehle, um das Abspielen eines Films (Movie) innerhalb einer Applikation oder eines Spiels zu integrieren.

Nur für Windows: da es die DirectX Technologie benutzt, kann jede Art von Movies (oder besser von Medien) mittels dieser Library abgespielt werden: AVI, MPG, DivX, Mp3 etc..

InitMovie()

Ergebnis = *InitMovie()*

Initialisiert die Programmumgebung für Movies zur späteren Benutzung. Sie müssen diese Funktion vor allen anderen Befehlen aus dieser Library aufrufen. Dieser Befehl versucht, DirectX (v3.0 mit NT4.0 Kompatibilität oder sonst v7.0) zu öffnen. Wenn dies also fehlschlägt (Rückgabewert gleich 0), liegt es möglicherweise an einer zu alten oder ganz

fehlenden Version von DirectX.

FreeMovie(#Movie)

Gibt das angegebene #Movie frei und entfernt es aus dem Speicher. Das #Movie ist nicht mehr gültig.

LoadMovie(#Movie, Dateiname\$)

Ergebnis = *LoadMovie(#Movie, Dateiname\$)*

Versucht ein Movie zu öffnen und es für das spätere Abspielen vorzubereiten. Ist das ‚Ergebnis‘ gleich 0, konnte das Movie nicht geöffnet werden (Format wird nicht unterstützt oder Datei wurde nicht gefunden), andernfalls ist alles ok. Dieses Movie wird zum aktuell benutzten Movie. *PlayMovie()* kann benutzt werden, um das Abspielen zu starten.

MovieAudio(Volume, Balance)

Kontrolliert den Audio-Stream des aktuellen Movies. Volume (Lautstärke) und Balance können während des Abspielens verändert werden. Veränderungen werden sofort wirksam. Der Volume Bereich geht von 0 bis 100 (100 ist am lautesten). Balance reicht von -100 bis 100 (-100 ist vollständig rechts, 0 ist Mittel (normaler Modus) und 100 ist vollständig links).

MovieHeight()

Höhe = *MovieHeight()*

Gibt die Höhe (in Pixel) des aktuellen Movies zurück. Ist das Ergebnis gleich -1, wurde kein (kompatibler) Video-Stream gefunden. Der Audio-Stream kann trotzdem abgespielt werden. Die ist besonders nützlich für das Abspielen von MP3-Dateien mit den Movie-Befehlen (!).

MovieInfo(Flags)

Ergebnis = *MovieInfo(Flags)*

Gibt zusätzliche Informationen über das aktuelle Movie zurück.

Derzeit werden folgende Werte als 'Flags' unterstützt:
0: gibt die Anzahl der "Frames pro Sekunde" (*1000) zurück.

MovieLength()

Länge = *MovieLength()*

Gibt die Länge (in Frames) des aktuellen Movies zurück.

MovieSeek(Frame)

Ändert die aktuelle Movie-Position auf das angegebene 'Frame'.

MovieStatus()

Ergebnis = *MovieStatus()*

Ermittelt den Status des aktuellen Movies.

„Ergebnis“ kann einer der folgenden Werte sein:

-1 : Movie ist unterbrochen ("paused").

0 : Movie ist angehalten ("stopped").

>0 : Movie wird abgespielt. Der zurückgegebene Wert ist die Nummer des gerade angezeigten Frames.

MovieWidth()

Breite = *MovieWidth()*

Gibt die Breite (in Pixel) des aktuellen Movies zurück. Ist das Ergebnis gleich -1, wurde kein (kompatibler) Video-Stream gefunden. Der Audio-Stream kann trotzdem abgespielt werden. Die ist besonders nützlich für das Abspielen von MP3-Dateien mit den Movie-Befehlen (!).

PauseMovie()

Hält das Abspielen des aktuellen Movies an. Das Abspielen kann mittels dem *ResumeMovie()* Befehl fortgesetzt werden.

PlayMovie(#Movie, WindowID)

Startet das Abspielen eines zuvor mittels *LoadMovie()* geladenen Movies auf dem angegebenen Fenster 'WindowID'. Die 'WindowID' kann einfach mittels dem *WindowID()*Befehl aus der Windows Library ermittelt werden. Der Befehl *ResizeMovie()* kann benutzt werden, um die Größe und die Position des Movies auf diesem Fenster zu verändern (um z.B. nicht die volle Fenstergröße zu benutzen).

ResizeMovie(x, y, Breite, Höhe)

Verändert die Größe und die Position des Anzeigebereichs vom aktuellen Movie auf dem Movie-Fenster. Dies ist besonders nützlich vor dem Abspielen mittels *PlayMovie()*.

ResumeMovie()

Setzt das Abspielen des aktuellen Movies fort, nach einem *PauseMovie()* Befehl.

StopMovie()

Stoppt (endgültig) das Abspielen des aktuellen Movies. Wird das Movie erneut abgespielt, startet es am Anfang.

UseMovie(#Movie)

Macht das angegebene #Movie zum aktuellen Movie.

16.17 Sprite

'Sprites' sind von Computerspielen wohlbekannt. Dies sind kleine Bilder, manchmal auch 'Brushes' (Pinsel) genannt, welche an jeder Position des Bildschirms angezeigt werden können. Die Sprites können mittels einem transparenten Layer (Schicht) über Grafiken bewegt werden. Noch besser: PureBasic erlaubt das Ausführen von Echtzeit-Effekten wie Schatten, Alpha-Blending, RGB-Filter, Alpha-Kanal, ... und all dies im Fenster- oder 'Fullscreen' (Bildschirm) Modus. DirectX 7 wird für das Sprite-Handling benutzt, was die Benutzung eines ggf. vorhandenen Hardware Blitter-Chips ermöglicht.

Da die Sprites unter Microsoft Windows eng mit dem Bildschirm ('Screen') verknüpft sind, wurden die Screen-relevanten Befehle in die

Sprite Library integriert (dies kann sich später ändern).

InitSprite()

Ergebnis = *InitSprite()*

Initialisiert die gesamte Sprite-Umgebung zur späteren Benutzung. Sie müssen diese Funktion am Anfang Ihres Sourcecodes einfügen, wenn Sie die Sprite-Funktionen benutzen möchten. Sie können das ‚Ergebnis‘ testen, um die korrekte Initialisierung der Sprite-Umgebung zu überprüfen. War diese nicht erfolgreich, beenden Sie das Programm oder deaktivieren Sie alle Aufrufe von Sprite-relevanten Befehlen. Dieser Befehl versucht, DirectX 7 zu initialisieren. Wenn er also fehlschlägt, liegt dies möglicherweise an einem fehlenden oder zu alten DirectX.

CatchSprite(#Sprite, SpeicherAdresse, Modus)

Ergebnis = *CatchSprite(#Sprite, SpeicherAdresse, Modus)*

Lädt das angegebene Sprite aus dem angegebenen Speicherbereich 'SpeicherAdresse'. Das Sprite muss im unkomprimierten BMP-Format vorliegen. Vor dem Laden eines Sprites sollte ein Bildschirm mit *OpenScreen()* oder *OpenWindowedScreen()* geöffnet sein. Ging etwas beim Laden des Sprites schief, wird als Ergebnis 'Result' 0 zurückgegeben. Das Sprite kann in einer beliebigen Dimension und in 256 Farben, 16 Bit, 24 Bit oder im 32 Bit-Format vorliegen. Ein geladenes Sprite kann mittels dem *FreeSprite()* Befehl freigegeben werden. Dieser Befehl ist nützlich im Zusammenhang mit dem 'IncludeBinary' PureBasic Schlüsselwort. Damit können Bilder mit in das Executable gepackt werden. Verwenden Sie diese Option trotzdem mit Bedacht, da mehr Speicher als beim Speichern des Sprites in einer externen Datei benötigt wird (das Sprite befindet sich im Speicher des Executable UND wird in den physikalischen Speicher geladen).

Der 'Modus' Parameter kann folgende Werte annehmen:

0 : Normaler Modus, das Sprite befindet sich im Video-Speicher (wenn möglich)

#PB_Sprite_Memory: das Sprite wird in den PC-Hauptspeicher geladen (für SpecialFX)

#PB_Sprite_Alpha: das Sprite ist in 8 Bit, grau und wird mit *DisplayAlphaSprite()* oder *DisplayShadowSprite()* benutzt

#PB_Sprite_Texture: das Sprite wird mit 3D Unterstützung erstellt, nützlich für den *CreateSprite3D()* Befehl der Sprite3D Library.

CatchJPEGSprite(#Sprite, SpeicherAdresse, JPEGLength, Modus)

Ergebnis = *CatchJPEGSprite(#Sprite, SpeicherAdresse, JPEGLength, Modus)*

Lädt das angegebene Sprite aus dem angegebenen Speicherbereich 'SpeicherAdresse'. Das Sprite muss im JPG-Format (mit Ausnahme der progressiven Form) vorliegen. Vor dem Laden eines Sprites sollte ein Bildschirm mit *OpenScreen()* oder *OpenWindowedScreen()* geöffnet sein. Ging etwas beim Laden des Sprites schief, wird als 'Result' 0 zurückgegeben. Das Sprite kann in einer beliebigen Dimension und in 256 Farben, 16 Bit, 24 Bit oder im 32 Bit-Format vorliegen. Ein geladenes Sprite kann mittels dem *FreeSprite()* Befehl freigegeben werden. Das JPEG Sprite benutzt eine interne JPEG Dekomprimierungs-Routine, es werden keine externen DLLs benötigt. Dieser Befehl ist nützlich im Zusammenhang mit dem 'IncludeBinary' PureBasic Schlüsselwort. Damit können Bilder mit in das Executable gepackt werden. Verwenden Sie diese Option trotzdem mit Bedacht, da mehr Speicher als beim Speichern des Sprites in einer externen Datei benötigt wird (das Sprite befindet sich im Speicher des Executable UND wird in den physikalischen Speicher geladen).

Der 'Modus' Parameter kann die folgenden Werte annehmen:

0 : Normaler Modus, das Sprite befindet sich im Video-Speicher (wenn möglich)

#PB_Sprite_Memory: das Sprite wird in den PC-Hauptspeicher geladen (für SpecialFX)

#PB_Sprite_Alpha: das Sprite ist in 8 Bit, grau und wird mit *DisplayAlphaSprite()* oder *DisplayShadowSprite()* benutzt

#PB_Sprite_Texture: das Sprite wird mit 3D Unterstützung erstellt, nützlich für den *CreateSprite3D()* Befehl der Sprite3D Library.

ChangeAlphaIntensity(R, G, B)

Ändert die Alpha-Intensität unabhängig von jedem der Rot, Grün und Blau-Kanäle. Dies wird zusammen mit dem *DisplayAlphaSprite()* Befehl benutzt. Das bedeutet, dass ein AlphaSprite in rot, grün oder jeder

Farbe anstelle der transparenten gerendert werden kann. Dies ermöglicht hübsche Echtzeit-Effekte.

ChangeGamma(R, G, B, Flags)

Ändert den Gamma-Wert für den aktuellen Bildschirm. Die Funktion arbeitet nur im 'FullScreen' Modus (nicht im Fenster-Modus). Die Rot, Grün und Blau-Kanäle können individuell geändert werden. Dieser Befehl kann zum Fading/Fadeout, Color-Splashing etc... des gesamten Bildschirms ('FullScreen') genutzt werden. Wenn dieser Befehl nichts tut, dann unterstützt die Hardware diese Funktion nicht (es wird keine Emulation angeboten, wegen der erforderlichen Tonnen von Operationen).

ClearScreen(R, G, B)

Löscht den gesamten Bildschirm mit der angegebenen RGB-Farbe.

ClipSprite(#Sprite, x, y, Breite, Höhe)

Ergebnis = *ClipSprite(#Sprite, x, y, Breite, Höhe)*

Fügt einen Clip-Bereich zum angegebenen Sprite hinzu. Zum Beispiel, wenn ein Sprite 100 x 100 (Breite x Höhe) groß ist und das Clipping mit (x=10, y=10, Breite=20, Höhe=20) angegeben wird, dann wird beim Anzeigen des Sprites nur der rechteckige Bereich beginnend bei x=10, y=10 mit einer Breite von 20 und einer Höhe von 20 angezeigt.

CloseScreen()

Schließt den aktuellen Bildschirm (egal ob "Windowed" oder "Fullscreen" Modus). Nach dem Schließen eines Bildschirms, müssen alle Sprites erneut geladen werden, da das Bildschirmformat verlorengeht und der Videospeicher freigegeben wird. Ein(e) Applikation/Spiel kann problemlos während des Programmablaufs zwischen "Fullscreen" und "Windowed" Modus umschalten.

CopySprite(#Sprite1, #Sprite2, Modus)

Kopiert das #Sprite1 in das #Sprite2. Wenn das #Sprite2 nicht existiert, wird es neu erstellt.

Der 'Modus' Parameter kann folgende Werte annehmen:

0 : Normaler Modus, das Sprite befindet sich im Video-Speicher (wenn möglich)

#PB_Sprite_Memory: das Sprite wird in den Hauptspeicher des PC geladen (für SpecialFX)

#PB_Sprite_Alpha: das Sprite ist in 8 Bit, grau und wird mit *DisplayAlphaSprite()* oder *DisplayShadowSprite()* benutzt

#PB_Sprite_Texture: das Sprite wird mit 3D Unterstützung erstellt, nützlich für den *CreateSprite3D()* Befehl der Sprite3D Library.

DisplayAlphaSprite(#Sprite, x, y)

Zeigt das #Sprite an der angegebenen Position auf dem aktuellen Bildschirm an. Die Farbe 128,128,128 (grau) wird als transparente Farbe verwendet (diese Farbe wird nicht dargestellt). Das Sprite emuliert einen 'Alpha-Kanal', wodurch alle mit so einem Kanal möglichen Tricks auch mit dem Sprite möglich sind. Das Sprite muss ein 8 Bit (256 Farben) Sprite sein, mit nur grauen Farben. Auf diesem Sprite ergibt jede Grauintensität einen dunkleren oder helleren Schatten. Dies ermöglicht sehr coole Effekte in Echtzeit, wie Soft-Shadow etc... Die Möglichkeiten sind gewaltig. *ChangeAlphaIntensity()* kann benutzt werden, um jede der R,G,B Komponenten des Alpha-Kanals zu kontrollieren. Das Sprite muss mit dem *LoadSprite()* Befehl und der #PB_AlphaSprite Option geladen werden. Dieser Befehl wird ge'clipped', womit es möglich ist, das Sprite mit negativen Werten ausserhalb des Bildschirms darzustellen.

Dieser Befehl arbeitet viel schneller, wenn *StartSpecialFX()* genutzt wird.

DisplayRGBFilter(x, y, Breite, Höhe, R, G, B)

Stellt einen Farb-Filter ('translucide') über dem Bildschirm dar. Dies ermöglicht sehr schnelles Darstellen eines rechteckigen Bereichs in einer beliebigen Farbe.

Dieser Befehl arbeitet viel schneller, wenn *StartSpecialFX()* genutzt wird.

DisplayShadowSprite(#Sprite, x, y)

Zeigt das #Sprite an der angegebenen Position auf dem aktuellen Bildschirm an. Die Farbe 0 wird als transparente Farbe angesehen (diese Farbe wird nicht dargestellt). Das Sprite wird wie ein Schatten dargestellt, unter Benutzung eines schnelleren Algorithmus als bei *DisplayAlphaSprite()*, aber mit nur einer Intensität. Das Sprite muss ein 8 Bit (256 Farben) Sprite sein. Das Sprite muss mit dem *LoadSprite()* Befehl und der #PB_AlphaSprite Option geladen werden. Dieser Befehl wird ge'clipped', womit es möglich ist, das Sprite mit negativen Werten ausserhalb des Bildschirms darzustellen.

Dieser Befehl arbeitet viel schneller, wenn *StartSpecialFX()* genutzt wird.

DisplaySolidSprite(#Sprite, x, y, R, G, B)

Zeigt das #Sprite an der angegebenen Position auf dem aktuellen Bildschirm an. Die Farbe 0 wird als transparente Farbe angesehen (diese Farbe wird nicht dargestellt). Das Sprite wird in einer Farbe - beschrieben durch die R,G,B Argumente - dargestellt. Das Sprite muss ein 8 Bit (256 Farben) Sprite sein. Es muss mit dem *LoadSprite()* Befehl und der #PB_AlphaSprite Option geladen werden. Dieser Befehl wird ge'clipped', womit es möglich ist, das Sprite mit negativen Werten ausserhalb des Bildschirms darzustellen.

Dieser Befehl arbeitet viel schneller, wenn *StartSpecialFX()* genutzt wird.

DisplaySprite(#Sprite, x, y)

Zeigt das #Sprite an der angegebenen Position (x, y) auf dem aktuellen Sprite-Buffer an. Es gibt keine transparente Farbe. Dieser Befehl ist ge'clipped' (wird automatisch auf den Bildschirm zugeschnitten), daher ist die Darstellung eines Sprites ausserhalb des Bildschirms mit negativen Werten möglich.

DisplayTranslucideSprite(#Sprite, x, y, Intensität)

Zeigt das #Sprite an der angegebenen Position auf dem aktuellen Bild-

schirm an. Die Farbe 0 wird als transparente Farbe angesehen (diese Farbe wird nicht dargestellt). Die 'Intensität' kontrolliert die Transparenz des Sprites über dem Hintergrund. Die Intensität-Werte reichen von 0 (völlig transparent) bis 255 (undurchsichtig). Dieser Effekt ist auch unter dem Namen 'Alpha-Blending' bekannt. Dieser Befehl wird ge'clipped', womit es möglich ist, das Sprite mit negativen Werten ausserhalb des Bildschirms darzustellen.

Dieser Befehl arbeitet viel schneller, wenn *StartSpecialFX()* genutzt wird.

DisplayTransparentSprite(#Sprite, x, y)

Zeigt das #Sprite an der angegebenen Position (x, y) auf dem aktuellen Bildschirm an. Standardmäßig wird die Farbe 0 (schwarz) als transparente Farbe betrachtet (diese Farbe wird nicht angezeigt). Es ist möglich, die transparente Farbe mittels *TransparentSpriteColor()* zu verändern. Dieser Befehl ist ge'clipped' (wird automatisch auf den Bildschirm zugeschnitten), daher ist die Darstellung eines Sprites außerhalb des Bildschirms mit negativen Werten möglich.

FlipBuffers()

Vertauscht den hinteren mit dem vorderen Buffer des aktuellen Bildschirms. So wird der unsichtbare Bereich nun sichtbar, welches einen 'Double-Buffering' Effekt (flickerfrei) ermöglicht. Ein Bildschirm muss mittels *OpenScreen()* geöffnet worden sein.

FreeSprite(#Sprite)

Entfernt das angegebene #Sprite aus dem Speicher. Sie können es nun nicht mehr benutzen.

GrabSprite(#Sprite, x, y, Breite, Höhe)

Fotografiert ("grab") den Bildschirminhalt im Bereich von (x, y, Breite, Höhe) und erstellt daraus ein neues #Sprite.

IsScreenActive()

Ergebnis = *IsScreenActive()*

Spiele und 'FullScreen' (den ganzen Bildschirm nutzende) Applikationen nutzen PureBasic Befehle, welche unter einer Multitasking Umgebung (AmigaOS, Windows oder Linux) laufen. Dies bedeutet, dass der User vom Bildschirm zurück auf den normalen Desktop wechseln kann. Diese Veränderung kann mit diesem Befehl registriert werden und sollte entsprechende Aktionen einleiten, wie *ReleaseMouse()*, Anhalten des Spiels, Stoppen der Sounds etc... Ist das 'Ergebnis' gleich 0, dann ist der Bildschirm nicht mehr aktiv, andernfalls ist der Bildschirm aktiv. Dieser Befehl muss nach einem *FlipBuffers()* aufgerufen werden, da die Ereignisse innerhalb von *FlipBuffers()* verwaltet werden.

LoadSprite(#Sprite, Dateiname\$, Modus)

Ergebnis = *LoadSprite(#Sprite, Dateiname\$, Modus)*

Lädt das angegebene Sprite in den Speicher zur sofortigen Verwendung. Das Sprite muss im BMP Format (in unkomprimierter Form) vorliegen. Vor dem Laden eines Sprites sollte ein Bildschirm mit *OpenScreen()* oder *OpenWindowedScreen()* geöffnet sein. Ging etwas beim Laden des Sprites schief, wird der Wert 0 zurückgegeben. Andernfalls ist alles in Ordnung... Die Sprites können in beliebigen Dimensionen und in 256 Farben, in 16 Bit, 24 Bit oder 32 Bit-Format vorliegen. Ein geladenes Sprite kann mit dem *FreeSprite()* Befehl freigegeben werden.

Der 'Modus' Parameter kann folgende Werte annehmen:

0 : Normaler Modus, das Sprite befindet sich im Video-Speicher (wenn möglich)

#PB_Sprite_Memory: das Sprite wird in den PC-Hauptspeicher geladen (für SpecialFX)

#PB_Sprite_Alpha: das Sprite ist in 8 Bit, grau und wird mit *DisplayAlphaSprite()* oder *DisplayShadowSprite()* benutzt

#PB_Sprite_Texture: das Sprite wird mit 3D Unterstützung erstellt, nützlich für den *CreateSprite3D()* Befehl der Sprite3D Library.

LoadJPEGSprite(#Sprite, Dateiname\$, Modus)

Ergebnis = *LoadJPEGSprite(#Sprite, Dateiname\$, Modus)*

Lädt das angegebene Sprite in den Speicher zur sofortigen Verwendung.

Das Sprite muss im JPG Format (mit Ausnahme der progressiven Form) vorliegen. Vor dem Laden eines Sprites sollte ein Bildschirm mit *OpenScreen()* oder *OpenWindowedScreen()* geöffnet sein. Ging etwas beim Laden des Sprites schief, wird der Wert 0 zurückgegeben. Andernfalls ist alles in Ordnung... Die Sprites können in beliebigen Dimensionen und in 256 Farben, in 16 Bit, 24 Bit oder 32 Bit-Format vorliegen. Ein geladenes Sprite kann mit dem *FreeSprite()* Befehl freigegeben werden. Das JPEG Sprite benutzt eine interne JPEG Dekompressions-Routine, es werden keine externen DLLs benötigt.

Der 'Modus' Parameter kann folgende Werte annehmen:

0 : Normaler Modus, das Sprite befindet sich im Video-Speicher (wenn möglich)

#PB_Sprite_Memory: das Sprite wird in den PC-Hauptspeicher geladen (für SpecialFX)

#PB_Sprite_Alpha: das Sprite ist in 8 Bit, grau und wird mit *DisplayAlphaSprite()* oder *DisplayShadowSprite()* benutzt

#PB_Sprite_Texture: das Sprite wird mit 3D Unterstützung erstellt, nützlich für den *CreateSprite3D()* Befehl der Sprite3D Library.

OpenScreen(Breite, Höhe, Tiefe, Titel\$)

Ergebnis = *OpenScreen(Breite, Höhe, Tiefe, Titel\$)*

Öffnet einen neuen Bildschirm entsprechend der angegebenen 'Breite', 'Höhe' und 'Tiefe'. Der geöffnete Bildschirm wird mit 2 Videobuffern erstellt, um das 'Double-Buffering' zu ermöglichen, welches speziell für Spiele entwickelt wurde. Die Buffer können mittels dem *FlipBuffers()* Befehl manipuliert werden. Wenn das Öffnen des Bildschirms fehlschlägt, ist 'Ergebnis' gleich 0.

Breite und Höhe sollten Standard-Kombinationen sein: 640*480, 800*600, 1024*768...

Tiefe kann die folgenden Werte annehmen:

- | | |
|----|--|
| 4 | : 16 Farben, unabhängige Palette möglich |
| 8 | : 256 Farben, unabhängige Palette möglich |
| 16 | : 65.000 Farben, feste Palette |
| 24 | : 16 Mio. Farben, feste Palette |
| 32 | : 16 Mio. Farben, schneller als der 24 Bit Modus |

OpenWindowedScreen(WindowID, x, y, Breite, Höhe, AutoStretch, RightOffset, BottomOffset)

Ergebnis = *OpenWindowedScreen(WindowID, x, y, Breite, Höhe, AutoStretch, RightOffset, BottomOffset)*

Öffnet einen neuen Bildschirm entsprechend den angegebenen Parametern auf dem angegebenen Fenster 'WindowID'. Ist 'AutoStretch' gleich 1, dann wird die Größe des Bildschirmbereichs automatisch angepasst, wenn sich die Fenstergröße ändert. 'RightOffset' und 'BottomOffset' werden benutzt, um einen rechten und einen unteren Rand des Fensters zu definieren (für eine StatusBar zum Beispiel). Der geöffnete Bildschirm wird mit 2 Videobuffern erstellt, um das speziell für Spiele entwickelte DoubleBuffering zu ermöglichen. Die Buffer können mittels dem *FlipBuffers()* Befehl manipuliert werden. Wenn das Öffnen des Bildschirms fehlschlägt, ist 'Ergebnis' gleich 0.

SaveSprite(#Sprite, Dateiname\$)

Speichert das angegebene #Sprite im BMP-Format auf Disk. Sehr nützlich für "Screenshots" (Schnappschüsse vom Bildschirminhalt) in Zusammenspiel mit dem *GrabSprite()* Befehl.

ScreenID()

ID = *ScreenID()*

Gibt die ScreenID des OS zurück. Unter Windows ist dies die normale WindowID. Damit kann jeder Befehl diesen Wert verwenden, der eine solche ID benutzt (wie zum Beispiel *PlayMovie()*).

ScreenOutput()

OutputID = *ScreenOutput()*

Gibt die 'OutputID' des aktuell benutzten Bildschirms zurück, um darauf 2D-Zeichenoperationen auszuführen. Hierfür wird die PureBasic 2DDrawing Library benutzt (siehe *StartDrawing()*).

SetFrameRate(FrameRate)

Legt die Frame-Rate (in Frames pro Sekunde) für den aktuellen Bildschirm fest. Dies ist besonders nützlich für den mit *OpenWindowedScreen()* festgelegten 'windowed' Bildschirmmodus.

SetRefreshRate(RefreshRate)

Legt die Refresh-Rate (in Hz) für den nächsten zu öffnenden Bildschirm fest. Wenn *OpenScreen()* die angegebene Rate nicht verarbeiten kann, schlägt diese Funktion fehl.

SpriteCollision(#Sprite1, x1, y1, #Sprite2, x2, y2)

Ergebnis = *SpriteCollision*(#Sprite1, x1, y1, #Sprite2, x2, y2)

Testet, ob sich die zwei Sprites überlappen. Wenn nicht, ist der Rückgabewert 'Ergebnis' gleich 0. Diese Routine eine rechteckige Prüfung, ergibt also eine sehr schnelle aber nicht sehr genaue Funktion. Sehr nützlich für schnelle Arcade-Spiele.

SpriteDepth(#Sprite)

Tiefe = *SpriteDepth*(#Sprite)

Gibt die Tiefe des angegebenen #Sprites zurück.

SpriteHeight(#Sprite)

Höhe = *SpriteHeight*(#Sprite)

Gibt die Höhe in Pixel des angegebenen #Sprite zurück.

SpriteOutput(#Sprite)

OutputID = *SpriteOutput*(#Sprite)

Ermittelt die OutputID des angegebenen #Sprite, um darauf 2D Zeichenoperationen durchzuführen. Es wird hierfür die PureBasic 2DDrawing Library (siehe *StartDrawing()*) benutzt.

SpritePixelCollision(#Sprite1, x1, y1, #Sprite2, x2, y2)

Ergebnis = SpritePixelCollision(#Sprite1, x1, y1, #Sprite2, x2, y2)

Überprüft, ob sich zwei Sprites überlappen. Wenn nein, ist das ‚Ergebnis‘ gleich 0. Diese Routine basiert auf einer exakten Kollisionsüberprüfung anhand transparenter Pixel, welches eine langsamere aber sehr genaue Routine darstellt. Um die Überprüfung zu optimieren, entfernen Sie so viele transparente Pixelfarben wie möglich, um die genaue Sprite-Größe zu erhalten.

SpriteWidth(#Sprite)

Breite = SpriteWidth(#Sprite)

Gibt die Breite in Pixel des angegebenen #Sprite zurück.

StartSpecialFX()

PureBasic erlaubt das Ausführen von Echtzeit-Effekten wie Alpha-Blending, Shadow, Color-Filter etc... Deren Darstellung ist sehr CPU-intensiv (auch wenn die Befehle sehr optimiert sind) und muss deshalb zur schnellen Ausführung einen Trick benutzen. *StartSpecialFX()* erstellt einen Speicherbuffer, worin das gesamte Rendering durchgeführt wird. Das Ergebnis wird zurück in den Video-Speicher kopiert, sobald *StopSpecialFX()* aufgerufen wird. Dies bedeutet, dass während dieser Periode keine Hardware-Beschleunigung (z.B. Grafikkarten-Chips) benutzt werden kann. Alle Operationen werden durch den Hauptprozessor ausgeführt. Es ist wichtig, dies zu verstehen, da andernfalls die Spiele-Performance gewaltig sinkt.

Zum Beispiel sollte der gesamte SpecialFX()-Code zusammengefasst und mit einmal gerendert werden (Shadow, Translucide Sprite etc...). Standard-Sprites sollten nach einem *StopSpecialFX()* dargestellt werden, um die Hardwarebeschleunigung auszunutzen. Befehle welche den SpezialFX Modus nutzen: *DisplayAlphaSprite()*, *DisplaySolidSprite()*, *DisplayShadowSprite()*, *DisplayRGBFilters()*, *DisplayTranslucideSprite()*

StopSpecialFX()

Sobald alle Rendering-Operationen ausgeführt wurden (siehe *StartSpecialFX()*), kopiert dieser Befehl den Buffer zurück in den Video-

Speicher und die Hardwarebeschleunigung ist wieder verfügbar.

TransparentSpriteColor(#Sprite, Rot, Grün, Blau)

Ändert die transparente Farbe des Sprites (wenn es mittels *DisplayTransparentSprite()* angezeigt wird). Ist der #Sprite Parameter gleich -1, dann wird die Standardfarbe (normalerweise schwarz (0,0,0)) auf die neu angegebene Farbe geändert und alle zukünftig geladenen Sprites (mittels *LoadSprite()*) benutzen diese Farbe als transparente Farbe.

16.18 Sprite3D

Die Sprite3D Library ist eine kleine 3D-Engine, die zum Anzeigen von 2D Sprites mit den neuen 3D Hardware-Möglichkeiten wie Echtzeit-Zooming, Transforming, Blending (Transparenz-Effekte) und mehr benutzt wird. Um mehr über die Sprites zu erfahren, lesen Sie einfach die Dokumentation zur 'Sprite' Library. Eine neuere Grafikkarte ist dringend zu empfehlen, um die volle Power dieser Funktionen (mit 3D-Beschleunigung) ausnutzen zu können. 3D-Sprites werden angeboten, um damit einfach ungewöhnliche Effekte erstellen zu können. Beachten Sie aber bitte, dass sie langsamer als normale Sprites sind und größeren Einschränkungen (z.B. in der Größe) unterliegen. DirectX 7 wird für das Sprite-Handling benutzt, was die Benutzung eines ggf. vorhandenen Hardware Blitter-Chips ermöglicht.

InitSprite3D()

Ergebnis = *InitSprite3D()*

Versucht, die gesamte 3D-Sprite Programmumgebung zu initialisieren. Ist das ‚Ergebnis‘ gleich 0, dann ist die 3D-Hardware (oder eine Emulation) nicht verfügbar. *InitSprite()* muss vor dieser Funktion aufgerufen werden.

CreateSprite3D(#Sprite3D, #Sprite)

Ergebnis = *CreateSprite3D(#Sprite3D, #Sprite)*

Erstellt ein 3D-Sprite '#Sprite3D' aus dem angegebenen 2D-Sprite '#Sprite'. Ein 3D-Sprite umfasst nur 4 Bildpunkte (ein Rechteck), worauf eine Textur gelegt ('mapped') wird. Die Textur ist in Wahrheit ein regu-

läres Sprite, dass mittels *LoadSprite()* im *#PB_Sprite_Texture* Modus geladen wurde. Die Textur muss ein Quadrat in den folgenden Grössen sein: 16*16, 32*32, 64*64, 128*128 oder 256*256. Andere Größen funktionieren möglicherweise auf einigen Grafikkarten, aber nicht auf allen. Ist das ‚Ergebnis‘ gleich 0, konnte das 3D-Sprite nicht erstellt werden, andernfalls ist alles ok.

Ein zum Erstellen eines 3D-Sprites benutztes 2D-Sprite muss nicht gelöscht werden, wenn ein 3D-Sprite es benutzt. Ein einzelnes 2D-Sprite kann von einer beliebigen Anzahl 3D-Sprites verwendet werden.

DisplaySprite3D(#Sprite3D, x, y, Transparenz)

Stellt das angegebene '#Sprite3D' an den angegebenen Koordinaten (x,y) dar. Ein Transparenz-Wert wird benutzt und sollte zwischen 0 und 255 liegen. 0 bedeutet unsichtbar (durchsichtig), 255 bedeutet undurchsichtig. Beliebige Werte in diesem Bereich können benutzt werden. Für eine bessere Darstellung ist dringend ein 32Bit-Bildschirmmodus angeraten, um weiches Ein- und Ausblenden zu ermöglichen.

Start3D() muss vor dieser Funktion aufgerufen werden. Wenn mehrere 3D-Sprites dargestellt werden sollen, ist es viel besser, diese alle mit einmal anzuzeigen.

Beispiel

```
Start3D()
DisplaySprite3D(1, 100, 100, 128)
DisplaySprite3D(2, 100, 150, 100)
...
Stop3D()
```

FreeSprite3D(#Sprite)

Entfernt das angegebene Sprite '#Sprite' aus dem Speicher. Das damit verknüpfte 2D-Sprite wird hierdurch jedoch nicht freigegeben.

RotateSprite3D(#Sprite3D, Winkel, Modus)

Dreht (rotiert) das angegebene #Sprite3D um den angegebenen 'Winkel'. Der Wert von 'Winkel' kann zwischen 0 und 360 liegen.

'Modus' kann folgende Werte annehmen:

0 : Alle Dimensionen werden auf ihre Original-Werte zurückge-

setzt (damit werden alle vorherigen "Zoom" oder "Rotate" Operationen rückgängig gemacht). Dies ist sehr nützlich, wenn mehrere Sprites zur gleichen Zeit aus einem #Sprite3D generiert werden. Dadurch kann jedes Sprite korrekt, mit seinem eigenen Winkel, angezeigt werden.

1 : Die vorherigen Sprite Dimensionen/Verformungen werden nicht verändert. Dies ermöglicht die Benutzung des *ZoomSprite3D()* Befehls vor dem "Rotate" Befehl, macht aber die gemeinsame Benutzung des #Sprite3D schwieriger.

Sprite3DBlendingMode(Ausgangsmodus, Zielmodus)

Verändert die Art, wie das Sprite3D (beim Benutzen von *DisplaySprite3D()* und einem Transparenz Wert) grafisch mit dem Hintergrund verbunden ("blended") wird. Beide Modi (Ausgangs- und Zielmodus) können einen Wert zwischen 0 und 24 annehmen. Dieser Befehl ist nur für fortgeschrittene Programmierer.

Hier die DirectX Liste der zugehörigen Werte:

```
#D3DTOP_SELECTARG1 = 2
#D3DTOP_SELECTARG2 = 3
#D3DTOP_MODULATE    = 4
#D3DTOP_MODULATE2X  = 5
#D3DTOP_MODULATE4X  = 6
#D3DTOP_ADD         = 7
#D3DTOP_ADDSIGNED   = 8
#D3DTOP_ADDSIGNED2X = 9
#D3DTOP_SUBTRACT    = 10
#D3DTOP_ADDSMOOTH   = 11
#D3DTOP_BLENDDIFFUSEALPHA = 12
#D3DTOP_BLENDTEXTUREALPHA = 13
#D3DTOP_BLENDFACTORALPHA = 14
#D3DTOP_BLENDTEXTUREALPHAM = 15
#D3DTOP_BLENDCURRENTALPHA = 16
#D3DTOP_PREMODULATE = 17
#D3DTOP_MODULATEALPHA_ADDCOLOR = 18
#D3DTOP_MODULATECOLOR_ADDALPHA = 19
#D3DTOP_MODULATEINVALPHA_ADDCOLOR = 20
#D3DTOP_MODULATEINVCOLOR_ADDALPHA = 21
#D3DTOP_BUMPENVMAP = 22
#D3DTOP_BUMPENVMAPLUMINANCE = 23
#D3DTOP_DOTPRODUCT3 = 24
```

Sprite3DQuality(Qualität)

Ändert die Art der Sprite3D Darstellung ("Rendering").

0 : Keine Filterung (schnell, aber "hässlich" beim Zoomen/Rotieren)

gebenen Dimensionen. Wird eine Drehung benötigt, muss *RotateSprite3D()* nach *ZoomSprite3D()* aufgerufen werden.

16.19 Mouse

PureBasic bietet vollständigen Zugriff auf die am Computer angeschlossenen Mäuse. Es unterstützt Standard-Mäuse mit bis zu 3 Knöpfen. Diese Library ist optimiert und benutzt "low-level" (hardware-nahe) Funktionen, speziell zur Benutzung in Spielen. Benutzen Sie diese Library nicht in einer regulären Applikation. Es wird die DirectX Technologie verwendet.

InitMouse()

Initialisiert die Programmumgebung zur weiteren Benutzung der Maus-Befehle. Sie müssen diese Funktion vor allen anderen Befehlen dieser Library aufrufen. Ergibt 'Result' gleich 0, ist keine Maus verfügbar... Dieser Befehl versucht, DirectX (v3.0 mit NT4.0 Kompatibilität oder sonst v7.0) zu öffnen. Wenn dies also fehlschlägt, liegt es möglicherweise an einer zu alten oder ganz fehlenden Version von DirectX.

ExamineMouse()

Ergebnis = *ExamineMouse()*

Aktualisiert den Maus-Status. Wird 0 zurückgegeben, kann die Maus nicht benutzt werden. Dieser Befehl sollte vor *MouseDeltaX()*, *MouseDeltaY()*, *MouseX()*, *MouseY()* oder *MouseButton()* benutzt werden. Zurzeit wird nur die Standard-Maus unterstützt.

MouseButton(ButtonNumber)

Ergebnis = *MouseButton(ButtonNumber)*

Gibt 0 zurück, wenn der angegebene Mausknopf 'ButtonNumber' nicht gedrückt wird, andernfalls ist der Knopf gedrückt. Eine beliebige Anzahl an Knöpfen kann zur gleichen Zeit gedrückt sein. *ExamineMouse()* muss vor dieser Funktion aufgerufen werden, um den aktuellen Status der Mausknöpfe zu ermitteln.

MouseDeltaX()

Ergebnis = *MouseDeltaX()*

Gibt die Mausbewegung auf der X-Achse (in Pixel) seit dem letzten Aufruf dieser Funktion zurück. Dies bedeutet, dass das Ergebnis entweder ein negativer oder ein positiver Wert sein kann, je nachdem ob die Maus seit dem letzten Aufruf nach links oder rechts bewegt wurde. *ExamineMouse()* muss vor dieser Funktion aufgerufen werden, um die aktuelle Maus-Position zu ermitteln.

MouseDeltaY()

Ergebnis = *MouseDeltaY()*

Gibt die Mausbewegung auf der Y-Achse (in Pixel) seit dem letzten Aufruf dieser Funktion zurück. Dies bedeutet, dass das Ergebnis entweder ein negativer oder ein positiver Wert sein kann, je nachdem ob die Maus seit dem letzten Aufruf nach oben oder unten bewegt wurde. *ExamineMouse()* muss vor dieser Funktion aufgerufen werden, um die aktuelle Maus-Position zu ermitteln.

MouseWheel()

Ergebnis = *MouseWheel()*

Gibt die Anzahl der seit dem letzten Aufruf erfolgten "Ticks" des Mausekkrads zurück. Der Wert ist positiv, wenn das Rad vorwärts gedreht wurde und negativ, wenn das Rad rückwärts gedreht wurde. Der Befehl *ExamineMouse()* muss vor dieser Funktion aufgerufen werden, um die Mausinformationen zu aktualisieren.

MouseX()

Ergebnis = *MouseX()*

Gibt die aktuelle Maus-Position (in Pixel) auf der X-Achse des aktuellen Bildschirms zurück. *ExamineMouse()* muss vor dieser Funktion aufgerufen werden, um die aktuelle Maus-Position zu ermitteln.

MouseY()

Ergebnis = *MouseY()*

Gibt die aktuelle Maus-Position (in Pixel) auf der Y-Achse des aktuellen Bildschirms zurück. *ExamineMouse()* muss vor dieser Funktion aufgerufen werden, um die aktuelle Maus-Position zu ermitteln.

ReleaseMouse()

ReleaseMouse()

Gibt ("unlock") die Maus zur Benutzung unter dem Standard-OS frei. Dies geschieht typischerweise nach Überprüfung des Ergebnisses von *IsScreenActive()*.

16.20 Keyboard

PureBasic bietet einfachen und schnellen Zugriff auf die Tastatur und sollte nur in Spielen oder Anwendungen benutzt werden, die einen extrem schnellen Zugriff auf die Tastatur im "Roh-Format" (raw) benötigen. Es wird die DirectX Technologie benutzt. Für die Abfrage von Tastaturkürzeln bei "normalen" Windows-Applikationen sehen Sie in die Dokumentation zur "Windows" Library unter "AddKeyboardShortcut".

InitKeyboard()

Initialisiert die Programmumgebung zur späteren Benutzung der Keyboard Befehle. Sie müssen diese Funktion vor allen anderen Befehlen aus dieser Bibliothek aufrufen. Dieser Befehl versucht, DirectX (v3.0 mit NT4.0 Kompatibilität oder sonst v7.0) zu öffnen. Wenn dies also fehlschlägt, liegt es möglicherweise an einer zu alten oder ganz fehlenden Version von DirectX.

ExamineKeyboard()

Aktualisiert den Tastatur-Status. Dieser Befehl muss vor den *KeyboardPushed()* oder *KeyboardReleased()* Befehlen aufgerufen werden.

KeyboardPushed(KeyID)

Ergebnis = *KeyboardPushed(KeyID)*

Gibt 0 zurück, wenn die angegebene Taste 'KeyID' nicht gedrückt wird,

andernfalls wird ein Wert ungleich 0 zurückgegeben. Eine beliebige Anzahl an Tasten kann zur gleichen Zeit gedrückt werden. Der Befehl *ExamineKeyboard()* muss vor dieser Funktion aufgerufen werden, um den Tastatur-Status zu aktualisieren.

KeyID : eine Konstante, welche die zu überprüfende Taste repräsentiert.
Liste gültiger 'KeyID' Konstanten:

#PB_Key_All ; Alle Tasten werden überprüft. Sehr nützlich zum Abfragen beliebiger Tastendrucke.

#PB_Key_1 ; alle Zahlen-Tasten	#PB_Key_A ; alle Buchstaben-Tasten
#PB_Key_2	#PB_Key_B
#PB_Key_3	#PB_Key_C
#PB_Key_4	...
#PB_Key_5	#PB_Key_X
#PB_Key_6	#PB_Key_Y
#PB_Key_7	#PB_Key_Z
#PB_Key_8	
#PB_Key_9	
#PB_Key_0	
#PB_Key_Escape	#PB_Key_Grave
#PB_Key_Minus	#PB_Key_LeftShift
#PB_Key_Equals	#PB_Key_BackSlash
#PB_Key_Back	#PB_Key_Comma
#PB_Key_Tab	#PB_Key_Period
#PB_Key_LeftBracket	#PB_Key_Slash
#PB_Key_RightBracket	#PB_Key_RightShift
#PB_Key_Return	#PB_Key_Multiply
#PB_Key_LeftControl	#PB_Key_LeftAlt
#PB_Key_SemiColon	#PB_Key_Space
#PB_Key_Apostrophe	#PB_Key_Capital
#PB_Key_F1 ; Funktions-Tasten	#PB_Key_NumLock
#PB_Key_F2	#PB_Key_Scroll
#PB_Key_F3	#PB_Key_Pad0
#PB_Key_F4	#PB_Key_Pad1
#PB_Key_F5	#PB_Key_Pad2
#PB_Key_F6	#PB_Key_Pad3
	#PB_Key_Pad4

#PB_Key_F7	#PB_Key_Pad5
#PB_Key_F8	#PB_Key_Pad6
#PB_Key_F9	#PB_Key_Pad7
#PB_Key_F10	#PB_Key_Pad8
#PB_Key_F11	#PB_Key_Pad9
#PB_Key_F12	
#PB_Key_Add	#PB_Key_Up ; Cursor-Tasten
#PB_Key_Subtract	#PB_Key_Down
#PB_Key_Decimal	#PB_Key_Left
#PB_Key_PadEnter	#PB_Key_Right
#PB_Key_RightControl	#PB_Key_End
#PB_Key_PadComma	#PB_Key_PageUp
#PB_Key_Divide	#PB_Key_PageDown
#PB_Key_RightAlt	#PB_Key_Insert
#PB_Key_Pause	#PB_Key_Delete
#PB_Key_Home	

KeyboardReleased(KeyID)

Ergebnis = *KeyboardReleased(KeyID)*

Gibt 1 zurück, wenn die angegebene Taste 'KeyID' gedrückt und wieder losgelassen wurde, andernfalls wird 0 zurückgegeben. Dieser Befehl ist sehr nützlich z.B. für eine "Pause" Taste in einem Spiel (einmal wird das Spiel angehalten, beim nächsten Mal wird es fortgesetzt). Der Befehl *ExamineKeyboard()* muss vor dieser Funktion aufgerufen werden, um den Tastatur-Status zu aktualisieren. Für eine komplette Liste der gültigen 'KeyID' Werte, sehen Sie unter *KeyboardPushed()*.

16.21 Joystick

PureBasic bietet vollen Zugriff auf am Computer angeschlossene Joysticks. Es unterstützt Standard-Joysticks mit bis zu 8 Knöpfen. Sie benutzt die DirectX Technologie.

InitJoystick()

Ergebnis = *InitJoystick()*

Initialisiert die gesamte Programmumgebung zur späteren Benutzung der Joystick-Befehle. Sie müssen diese Funktion vor allen anderen Be-

fehlen aus dieser Bibliothek aufrufen. Dieser Befehl versucht, DirectX (v3.0 mit NT4.0 Kompatibilität oder sonst v7.0) zu öffnen. Wenn dies also fehlschlägt, liegt es möglicherweise an einer zu alten oder ganz fehlenden Version von DirectX.

ExamineJoystick()

Status = *ExamineJoystick()*

Diese Funktion muss vor den nachfolgenden Befehlen aufgerufen werden, um den aktuellen Status des Joysticks wiederzugeben: *JoystickButton()*, *JoystickAxisX()*, *JoystickAxisY()*. Zur Zeit wird nur der Standard-Joystick unterstützt.

JoystickAxisX()

Ergebnis = *JoystickAxisX()*

Gibt einen der folgenden Werte zurück:

- 1 : Bewegung nach links
- 0 : Keine Bewegung auf der X-Achse
- 1 : Bewegung nach rechts

ExamineJoystick() muss vor dieser Funktion aufgerufen werden, um den aktuellen Joystick-Status zu ermitteln.

JoystickAxisY()

Ergebnis = *JoystickAxisY()*

Gibt einen der folgenden Werte zurück:

- 1 : Bewegung nach oben (vorn)
- 0 : Keine Bewegung auf der Y-Achse
- 1 : Bewegung nach unten (zurück)

ExamineJoystick() muss vor dieser Funktion aufgerufen werden, um den aktuellen Joystick-Status zu ermitteln.

JoystickButton(ButtonNumber)

Ergebnis = *JoystickButton(ButtonNumber)*

Gibt 0 zurück, wenn der angegebene Joystick-Knopf 'ButtonNumber'

nicht gedrückt wird, andernfalls wird ein Wert ungleich 0 zurückgegeben. Eine beliebige Anzahl an Knöpfen kann zur gleichen Zeit gedrückt sein. *ExamineJoystick()* muss vor dieser Funktion aufgerufen werden, um den aktuellen Status der Joystick-Knöpfe zu ermitteln.

16.22 Window

Fenster (Windows) sind unerlässliche Bestandteile moderner Benutzeroberflächen. PureBasic gewährt Ihnen vollen Zugriff darauf.

ActivateWindow()

Aktiviert das aktuelle Fenster und zeigt dieses im Vordergrund an.

AddKeyboardShortcut(#Window, Shortcut, EventID)

Fügt einen Tastatur-Shortcut ("Tastenkürzel") zum angegebenen Fenster '#Window' hinzu. Ein Shortcut generiert ein Menü-Ereignis (wie ein Menü-Eintrag), da sie meistens im Zusammenhang mit Menüs benutzt werden. Der 'EventID' ist der Wert, welcher vom *EventMenuID()* Befehl zurückgegeben wird. Standardmäßig hat ein Fenster bereits die `#PB_Shortcut_Tab` und `#PB_Shortcut_Tab|#PB_Shortcut_Shift` Shortcuts, um die Tabulator und Shift-Tabulator Tasten(kombinationen) korrekt zu handhaben. Ein Shortcut kann mittels *RemoveKeyboardShortcut()* entfernt werden.

Der 'Shortcut' Parameter kann eine der folgenden Konstanten sein:

#PB_Shortcut_Back	#PB_Shortcut_0	#PB_Shortcut_Pad0
#PB_Shortcut_Tab	#PB_Shortcut_1	#PB_Shortcut_Pad1
#PB_Shortcut_Clear	#PB_Shortcut_2	#PB_Shortcut_Pad2
#PB_Shortcut_Return	#PB_Shortcut_3	#PB_Shortcut_Pad3
#PB_Shortcut_Menu	#PB_Shortcut_4	#PB_Shortcut_Pad4
#PB_Shortcut_Pause	#PB_Shortcut_5	#PB_Shortcut_Pad5
#PB_Shortcut_Print	#PB_Shortcut_6	#PB_Shortcut_Pad6
#PB_Shortcut_Capital	#PB_Shortcut_7	#PB_Shortcut_Pad7
#PB_Shortcut_Escape	#PB_Shortcut_8	#PB_Shortcut_Pad8
#PB_Shortcut_Space	#PB_Shortcut_9	#PB_Shortcut_Pad9
#PB_Shortcut_Prior	#PB_Shortcut_A	#PB_Shortcut_F1
#PB_Shortcut_Next	#PB_Shortcut_B	#PB_Shortcut_F2
#PB_Shortcut_End	#PB_Shortcut_C	#PB_Shortcut_F3
#PB_Shortcut_Home	#PB_Shortcut_D	#PB_Shortcut_F4
#PB_Shortcut_Left	#PB_Shortcut_E	#PB_Shortcut_F5
#PB_Shortcut_Up	#PB_Shortcut_F	#PB_Shortcut_F6
#PB_Shortcut_Right	#PB_Shortcut_G	#PB_Shortcut_F7
#PB_Shortcut_Down	#PB_Shortcut_H	#PB_Shortcut_F8
#PB_Shortcut_Select	#PB_Shortcut_I	#PB_Shortcut_F9
#PB_Shortcut_Execute	#PB_Shortcut_J	#PB_Shortcut_F10
#PB_Shortcut_Snapshot	#PB_Shortcut_K	#PB_Shortcut_F11
#PB_Shortcut_Insert	#PB_Shortcut_L	#PB_Shortcut_F12
#PB_Shortcut_Delete	#PB_Shortcut_M	#PB_Shortcut_F13
#PB_Shortcut_Help	#PB_Shortcut_N	#PB_Shortcut_F14
#PB_Shortcut_LeftWindows	#PB_Shortcut_O	#PB_Shortcut_F15
#PB_Shortcut_RightWindows	#PB_Shortcut_P	#PB_Shortcut_F16
#PB_Shortcut_Apps	#PB_Shortcut_Q	#PB_Shortcut_F17
#PB_Shortcut_Numlock	#PB_Shortcut_R	#PB_Shortcut_F18
#PB_Shortcut_Scroll	#PB_Shortcut_S	#PB_Shortcut_F19
#PB_Shortcut_Multiply	#PB_Shortcut_T	#PB_Shortcut_F20
#PB_Shortcut_Add	#PB_Shortcut_U	#PB_Shortcut_F21
#PB_Shortcut_Separator	#PB_Shortcut_V	#PB_Shortcut_F22
#PB_Shortcut_Subtract	#PB_Shortcut_W	#PB_Shortcut_F23
#PB_Shortcut_Decimal	#PB_Shortcut_X	#PB_Shortcut_F24
#PB_Shortcut_Divide	#PB_Shortcut_Y	
	#PB_Shortcut_Z	

Die oben angegebene Taste kann mit jeder der folgenden Konstanten kombiniert werden:

#PB_Shortcut_Shift

#PB_Shortcut_Control
#PB_Shortcut_Alt

CloseWindow(#Window)

Schließt das angegebene Fenster.

DetachMenu()

Entfernt das Menü vom aktuell benutzten Fenster. Dies wird oft benutzt, um anschließend das Menü-Layout zu ändern.

EventGadgetID()

#Gadget = *EventGadgetID()*

Benutzen Sie diese Funktion nach einem *WindowEvent()* oder *WaitWindowEvent()* Befehl, um den gedrückten Schalter bestimmen zu können (gibt die #Gadget Nummer zurück).

EventMenuID()

#Menu = *EventMenuID()*

Benutzen Sie diese Funktion nach einem *WindowEvent()* oder *WaitWindowEvent()* Befehl, um den ausgewählten Menü-Eintrag bestimmen zu können (gibt die #MenuItem Nummer zurück).

EventType()

Ereignis = *EventType()*

Nach einem *WindowEvent()* oder *WaitWindowEvent()* Befehl benutzen Sie diese Funktion, um den Typ des letzten Ereignisses festzustellen. *EventType()* ist jetzt eng verknüpft mit den Gadget und Systray Libraries und kann nach dem Auftreten eines Ereignisses folgende Werte zurückgeben:

#PB_EventType_LeftClick

.....

Maustaste

#PB_EventType_RightClick	: Klick mit der rechten Maustaste
--------------------------	-----------------------------------

#PB_EventType_LeftDoubleClick	: Doppelklick mit der linken Maustaste
-------------------------------	--

#PB_EventType_RightDoubleClick	: Doppelklick mit der rechten Maustaste
--------------------------------	---

#PB_EventType_ReturnKey	: Eingabe von 'Return' in einem <i>StringGadget()</i> .
-------------------------	---

#PB_EventType_Focus	: wird zurückgegeben, wenn ein <i>StringGadget()</i> den Focus erhält.
---------------------	--

#PB_EventType_LostFocus	: wird zurückgegeben, wenn ein <i>StringGadget()</i> den Focus verliert.
-------------------------	--

#PB_EventType_Change	: wird zurückgegeben, wenn sich der Inhalt eines <i>StringGadget()</i> ändert.
----------------------	--

#Window = *EventWindowID()*

Benutzen Sie diese Funktion nach einem *WindowEvent()* oder *WaitWindowEvent()* Befehl, um das Fenster bestimmen zu können, in dem ein Ereignis stattfand.

HideWindow(#Window, Status)

HideWindow(#Window, Status)

Versteckt oder zeigt das angegebene Fenster '#Window'.

'Status' kann folgende Werte annehmen:

- 1 : das Fenster #Window wird versteckt.
- 0 : das Fenster #Window wird angezeigt.

MoveWindow(x, y)

Verschiebt das Fenster an die angegebenen Koordinaten. Die Koordinaten beziehen sich auf die linke obere Bildschirm-Ecke, beginnend mit (0,0).

OpenWindow(#Window, x, y, InnereBreite, InnereHöhe, Flags, Titel\$)

WindowID = *OpenWindow(#Window, x, y, InnereBreite, InnereHöhe, Flags, Titel\$)*

Öffnet ein neues Fenster entsprechend den übergebenen Parametern. Das neue Fenster wird auch automatisch zum aktiven Fenster. Sie müssen dafür also nicht erst den *UseWindow()* Befehl benutzen. Ergibt die 'WindowID' gleich 0, konnte das Fenster nicht geöffnet werden. 'InnereBreite' und 'InnereHöhe' definieren die innere Fenstergröße (ohne Ränder und andere Fensterdekorationen), um die Windows XP, Amiga-OS und Linux "Skinning" Kompatibilität zu gewährleisten.

Mögliche Flags sind:

#PB_Window_SystemMenu	Schaltet das System-Menü in der Fenster-Titelzeile ein.
#PB_Window_MinimizeGadget	Fügt das Minimieren-Gadget der Fenster-Titelzeile hinzu. #PB_Window_System wird automatisch hinzugefügt.
#PB_Window_MaximizeGadget	Fügt das Maximieren-Gadget der

	#PB_Window_System wird automatisch hinzugefügt.
#PB_Window_SizeGadget	Fügt das Größenänderungs-Gadget zum Fenster hinzu.
#PB_Window_Invisible	Erstellt ein Fenster, zeigt es aber nicht an.
#PB_Window_TitleBar	Erstellt ein Fenster mit einer Titelleiste.
#PB_Window_BorderLess	Erstellt ein Fenster ohne jegliche Ränder.

ReSizeWindow(Breite, Höhe)

Ändert die Größe des Fensters auf die angegebenen Werte (Breite, Höhe).

RemoveKeyboardShortcut(#Window, Shortcut)

Entfernt ein Tastatur-Shortcut ("Tastenkürzel"), welches zuvor mit *AddKeyboardShortcut()* definiert wurde. Für eine vollständige Liste der verfügbaren Shortcuts sehen Sie einfach beim *AddKeyboardShortcut()* Befehl nach. Wird als 'Shortcut' #PB_Shortcut_All angegeben, werden alle Shortcuts entfernt.

SetWindowCallback(@ProcedureName())

Nur für erfahrene Programmierer. Diese Funktion wird nur unter dem Microsoft Windows OS unterstützt. Normale Ereignisse (Events) sollten mit den regulären Befehlen *WaitWindowEvent()* oder *WindowEvent()* verarbeitet werden. Dieser Befehl assoziiert einen Callback ("Rückruf") um die Events des aktuellen Fensters zu verarbeiten. Alle Ereignisse werden durch diesen Callback abgefangen und können hier verarbeitet

werden. Eine Warnung - dieser Weg ist "low-level" und der Debugger kann während dieser Periode nicht benutzt werden. Die Callback Prozedur muss 4 Parameter haben.

Hier ist ein Beispiel-Code, wie ein Callback richtig benutzt wird:

```
Procedure MyWindowCallback(WindowID, Message, wParam, lParam)
    Result = #PB_ProcessPureBasicEvents
    ;
    ; Ihr Programmcode hier
    ;
    ProcedureReturn Result
EndProcedure
```

UseWindow(#Window)

Macht das angegebene Fenster zum aktuell-benutzten ('currently-used') Fenster.

WaitWindowEvent()

Ereignis = *WaitWindowEvent()*

Wartet bis ein Ereignis auftritt. Es ist dieselbe Funktion wie *WindowEvent()*, hält aber die Programmausführung an, was sehr wichtig in einer Multitasking Umgebung ist. Eine Applikation sollte möglichst immer diesen Befehl anstelle von *WindowEvent()* benutzen. Für weitere Informationen, sehen Sie die Dokumentation zu *WindowEvent()*

WindowEvent()

Ereignis = *WindowEvent()*

Überprüft, ob in irgendeinem der geöffneten Fenster ein Ereignis stattfand. Diese Funktion kehrt umgehend zum Programmablauf zurück, seien Sie also vorsichtig, um nicht die gesamte CPU-Power mit einer Schleife ohne Wartefunktion ('non temporized loop') zu verschwenden... Um die Programmausführung bis zum Stattfinden eines Ereignisses anzuhalten, benutzen Sie einfach den *WaitWindowEvent()* Befehl. Um die Fenster-Nummer, in dem das Ereignis stattfand, bestimmen zu können, müssen Sie die *EventWindowID()* Funktion benutzen.

Mögliche Ereignisse sind:

#PB_EventMenu	ein Menü wurde ausgewählt
#PB_EventGadget	ein Gadget wurde gedrückt
#PB_EventCloseWindow	das Schließgadget vom Fenster wurde gedrückt
#PB_EventRepaint	der Fensterinhalt wurde zerstört und muss neu gezeichnet werden (nützlich für 2D Grafik-Operationen)
#PB_EventMoveWindow	das Fenster wurde verschoben

WindowID([#Window])

WindowID = *WindowID([#Window])*

Ermittelt die einmalige ID (Handle), welche das aktuelle Fenster im Betriebssystem (OS) identifiziert. Diese Funktion ist sehr nützlich, wenn eine andere Library einen Bezug zu einem Fenster benötigt. Eine optionales '#Window' kann angegeben werden.

WindowHeight()

Höhe = *WindowHeight()*

Ermittelt die Höhe, in Pixel, des aktuellen Fensters.

WindowWidth()

Breite = *WindowWidth()*

Ermittelt die Breite, in Pixel, des aktuellen Fensters.

WindowX()

`x = WindowX()`

Ermittelt die Position, in Pixel, des linken Randes des aktuellen Fensters.

WindowY()

`y = WindowY()`

Ermittelt die Position, in Pixel, des oberen Randes des aktuellen Fensters.

WindowMouseX()

`x = WindowMouseX()`

Ermittelt die Maus-Position relativ zum linken Rand des aktuellen Fensters. Der Wert kann negativ sein, wenn sich der Mauszeiger außerhalb des Fensters befindet.

WindowMouseY()

`y = WindowMouseY()`

Ermittelt die Maus-Position relativ zum oberen Rand des aktuellen Fensters. Der Wert kann negativ sein, wenn sich der Mauszeiger außerhalb des Fensters befindet.

WindowOutput()

`OutputID = WindowOutput()`

Gibt die 'OutputID' des aktuell benutzten Fensters zurück, um darauf 2D-Zeichenoperationen auszuführen. Hierfür wird die PureBasic 2DDrawing Library (siehe *StartDrawing()*) benutzt.

16.23 Font

Zeichensätze (Fonts) sind auf Computern weit verbreitet, da sie die einzige Möglichkeit darstellen, Text in verschiedenen Größen und Formen auszugeben.

CloseFont(#Font)

Schließt den angegebenen Zeichensatz, der zuvor mittels *LoadFont()* initialisiert wurde.

FontID()

FontID.1 = *FontID()*

Gibt den System Font-Zeiger zurück.

LoadFont(#Font, Name\$, YSize)

FontID.1 = *LoadFont(#Font, Name\$, YSize)*

Versucht den angegebenen Zeichensatz (Name\$, YSize = Höhe) zu öffnen. Ergibt die zurückgegebene FontID gleich 0, wurde der Zeichensatz nicht gefunden. Wenn vorher ein anderer Zeichensatz mit der gleichen '#Font' Nummer geladen wurde, wird dieser automatisch freigegeben.

UseFont(#Font)

Ändert den aktuellen Zeichensatz auf den angegebenen Zeichensatz '#Font'.

16.24 Gadget

Die Gadgets in PureBasic stehen als allgemeiner Begriff für alle Oberflächen-Komponenten: Schalter, Häkchenboxen, Auswahlfenster, Schalttafeln,... Diese Library ist OS-unabhängig und verwendet die wirklichen OS Graphical User Interface (GUI) Komponenten.

ActivateGadget(#Gadget)

Aktiviert (setzt den Fokus) auf das angegebene Gadget. Dies wird hauptsächlich zusammen mit den *ComboBoxGadget()* und *StringGadget()* Gadgettypen benutzt.

AddGadgetColumn(#Gadget, Position, Titel\$, Breite)

Fügt eine Spalte zum angegebenen #Gadget hinzu. Der Gadget-Typ kann einer der folgenden sein:

ListIconGadget()

Die Position definiert den Spaltenindex, wo der neue Eintrag eingefügt werden soll. 'Breite' gibt die ursprüngliche Breite der neuen Spalte an.

AddGadgetItem(#Gadget, Position, Text\$ [, ImageID]))

Fügt einen Eintrag zum angegebenen #Gadget hinzu. Eine optionale *ImageID()* kann für Gadgets angegeben werden, die dies unterstützen (wie *ListIconGadget()*). Der Gadgettyp kann einer der folgenden sein:

ComboBoxGadget()

PanelGadget()

ListViewGadget()

ListIconGadget(): unterstützt die ImageID.

TreeGadget(): unterstützt die ImageID. Der Parameter 'Position' wird ignoriert.

Die 'Position' definiert den Eintrag-Index, wo der neue Eintrag eingefügt werden soll. Um diesen Eintrag am Ende der aktuellen Eintrag-Liste hinzuzufügen, benutzen Sie einen Wert von -1.

ButtonGadget(#Gadget, x, y, Breite, Höhe, Text\$ [, Flags])

Erstellt ein Schalter-Gadget innerhalb der Gadget-Liste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird.

'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_Button_Right	rechtsbündige Darstellung des Schalter-Textes
------------------	---

#PB_Button_Left	linksbündige Darstellung des Schalter-Textes
-----------------	--

#PB_Button_Default	legt das definierte Aussehen des Schalters als Standard-Schalter für das Fenster fest
#PB_Button_MultiLine	Ist der Text zu lang, wird er über mehrere Zeilen dargestellt
#PB_Button_Toggle	erstellt einen 'Toggle' Schalter, ein Klick und der Schalter bleibt gedrückt, ein weiterer Klick gibt ihn wieder frei

ButtonImageGadget(#Gadget, x, y, Breite, Höhe, ImageID)

Erstellt ein Schalter-Gadget innerhalb der aktuellen Gadgetliste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Die 'ImageID' repräsentiert eine BMP-Datei und kann einfach mittels der Befehle *ImageID()* bzw. *UseImage()* aus der Image Library ermittelt werden.

ChangeListIconGadgetDisplay(#Gadget, Modus)

Ein *ListIconGadget()* kann unter dem MS Windows Betriebssystem vier verschiedene Anzeigemodi annehmen: kleine Icons, große Icons, Listen und Reporte (Spalten).

'Modus' kann einen der folgenden Werte annehmen:

- 0 : großer Icon Modus
- 1 : kleiner Icon Modus
- 2 : List Modus
- 3 : Report Modus (Spalten, Standardmodus)

CheckBoxGadget(#Gadget, x, y, Breite, Höhe, Text\$ [, Flags])

Erstellt ein Checkbox-Gadget (Häkchen-Schalter) innerhalb der Gadget-Liste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Der 'Text\$' ist eine optionale Beschreibung der Checkbox, welche rechts vom Gadget dargestellt wird.

GetGadgetState() kann zum Ermitteln des aktuellen Gadget-Status ver-

wendet werden (markiert oder unmarkiert). *SetGadgetState()* kann zum Ändern des Gadget-Status verwendet werden (markiert oder unmarkiert).

'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_CheckBox_Right rechtsbündige Darstellung des Textes

#PB_CheckBox_Center zentrierte Darstellung des Textes

ClearGadgetItemList(#Gadget)

Löscht alle Einträge im angegebenen #Gadget. Das Gadget muss einer der folgenden Typen sein:

ComboBoxGadget()

ListViewGadget()

PanelGadget()

ListIconGadget()

TreeGadget()

ClosePanelGadget()

Beendet die Erstellung des aktuellen PanelGadget und geht zurück zur vorhergehenden Gadget-Liste. *PanelGadget()* muss vor dieser Funktion aufgerufen worden sein.

CloseTreeGadgetNode()

Schließt den aktuellen "Knoten" und verzweigt zurück zum übergeordneten Eintrag. Dies wird benutzt nach einem *OpenTreeGadgetNode()*.

ComboBoxGadget(#Gadget, x, y, Breite, Höhe [, Flags])

Erstellt ein ComboBox (Auswahl-) Gadget innerhalb der aktuellen Gadget-Liste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Sobald eine ComboBox erstellt wurde, ist ihre Liste mit Einträgen leer. Die folgenden Befehle können

benutzt werden, um auf den Listen-Inhalt zuzugreifen:

AddGadgetItem(): fügt einen Eintrag hinzu

RemoveGadgetItem(): entfernt einen Eintrag

ClearGadgetItemList(): entfernt alle Einträge

SetGadgetState() kann zum Ändern des ausgewählten Eintrags verwendet werden.

GetGadgetState() kann zum Ermitteln des Index vom aktuellen Element verwendet werden.

„Flags“ sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_ComboBox_Editable: Macht die ComboBox editierbar

#PB_ComboBox_LowerCase : Der gesamte in der ComboBox eingegebene Text wird in Kleinbuchstaben konvertiert.

#PB_ComboBox_UpperCase : Der gesamte in der ComboBox eingegebene Text wird in Großbuchstaben konvertiert.

CountGadgetItems(#Gadget)

Ergebnis = *CountGadgetItems(#Gadget)*

Gibt die Anzahl der Einträge im angegebenen '#Gadget' zurück. Der erste Eintrag beginnt bei 1. Wenn das Gadget keinerlei Einträge enthält, wird 0 zurückgegeben. Dies ist eine universelle Funktion, welche mit allen Gadgets arbeitet, die mehrere Einträge verwalten:

ListViewGadget()

ListIconGadget()

PanelGadget()

ComboGadget()

TreeGadget()

CreateGadgetList(WindowID)

Ergebnis = *CreateGadgetList(WindowID)*

Reserviert die Ressourcen für eine zukünftige Gadgetliste. Dieser Befehl muss vor allen Befehlen zur Gadget-Erstellung wie *ButtonGadget()*,

etc... aufgerufen werden. Falls der Rückgabewert 0 ergibt, ist die Erstellung einer Gadgetliste fehlgeschlagen und es können keine Gadgets definiert werden.

DisableGadget(#Gadget, Status)

Deaktiviert oder aktiviert ein Gadget. Ist der Status = 1 wird das Gadget deaktiviert, bei einem Status = 0 wird es aktiviert.

Frame3DGadget(#Gadget, x, y, Breite, Höhe, Text\$, Flags)

Erstellt ein Frame3D (Rahmen) Gadget innerhalb der Gadget-Liste. Diese Art von Gadget dient nur zur dekorativen Zwecken. Verschiedene Darstellungen des Rahmens sind möglich, diese können mit den folgenden 'Flag' Werten erreicht werden:

- 0 : Normal, 3D Rahmen
- 1 : Doppelt vertiefter Rahmen
- 2 : Einfach vertiefter Rahmen

FreeGadget(#Gadget)

Gibt das '#Gadget' frei und entfernt es vom Bildschirm und aus der Gadgetliste.

GadgetID(#Gadget)

GadgetID = GadgetID(#Gadget)

Gibt die einmalige 'GadgetID' des '#Gadget' zurück.

GadgetToolTip(#Gadget, Text\$)

GadgetToolTip(#Gadget, Text\$)

Verknüpft den angegebenen 'Text\$' mit dem '#Gadget'. Ein Tooltip-Text ist ein Text, der in einer kleinen gelben Box erscheint, wenn sich der Mauszeiger eine Weile über einem Gadget befindet.

GadgetX(#Gadget)

Position = *GadgetX(#Gadget)*

Gibt die X-Position des angegebenen '#Gadget' zurück.

GadgetY(#Gadget)

Position = *GadgetY(#Gadget)*

Gibt die Y-Position des angegebenen '#Gadget' zurück.

GadgetHeight(#Gadget)

Höhe = *GadgetHeight(#Gadget)*

Gibt die Höhe des angegebenen '#Gadget' zurück.

GadgetWidth(#Gadget)

Breite = *GadgetWidth(#Gadget)*

Gibt die Breite des angegebenen '#Gadget' zurück.

GetGadgetItemState(#Gadget, Eintrag)

Status = *GetGadgetItemState(#Gadget, Eintrag)*

Gibt den Status des 'Eintrag's vom angegebenen '#Gadget' zurück. Dies ist eine universelle Funktion, welche zusammen mit den meisten Gadgets arbeitet, die mehrere Einträge verwalten:

ListViewGadget(): gibt 1 zurück, wenn der Eintrag ausgewählt ist, andernfalls 0.

ListIconGadget(): gibt 1 zurück, wenn der Eintrag ausgewählt ist, andernfalls 0.

TreeGadget(): gibt eine Kombination der folgende Werte zurück:

1 : Der Eintrag ist ausgewählt, sonst 0.

2 : Der Eintrag ist ausgeklappt (ein Eintrag-"Baum" ist geöffnet), sonst 0.

Zum Beispiel: ist 'Result' gleich 3, ist der Eintrag ausgewählt und ausgeklappt.

GetGadgetItemText(#Gadget, Eintrag, Spalte)

Text\$ = GetGadgetItemText(#Gadget, Eintrag, Spalte)

Gibt den Textinhalt des 'Eintrag's vom angegebenen '#Gadget' zurück. Dies ist eine universelle Funktion, welche zusammen mit den meisten Gadgets arbeitet, die mehrere Einträge verwalten:

ListViewGadget()

ListIconGadget()

PanelGadget()

ComboGadget()

TreeGadget()

GetGadgetState(#Gadget)

Status = GetGadgetState(#Gadget)

Gibt den aktuellen Status des Gadgets zurück. Dies ist eine universelle Funktion, die mit den meisten der Gadgettypen arbeitet:

CheckBoxGadget() : gibt 1 zurück, wenn abgehakt, andernfalls 0.

ComboBoxGadget() : gibt den aktuell gewählten Eintrag-Index zurück, -1 wenn nichts markiert ist.

ListViewGadget() : gibt den aktuell gewählten Eintrag-Index zurück, -1 wenn nichts markiert ist.

OptionGadget() : gibt 1 zurück, wenn aktiviert, andernfalls 0.

PanelGadget() : gibt den aktuellen Panel-Index zurück, -1 falls kein Panel.

ListIconGadget() : gibt den aktuell gewählten Eintrag-Index zurück, -1 wenn nichts markiert ist.

TreeGadget() : gibt den aktuell gewählten Eintrag-Index zurück, -1 wenn nichts markiert ist.

GetGadgetText(#Gadget)

Text\$ = GetGadgetText(#Gadget)

Gibt den Inhalt des Gadget-Textes vom angegebenen #Gadget zurück. Dieser Befehl ist besonders nützlich für:

StringGadget()

ComboBoxGadget()

ListViewGadget()

TextGadget()

HideGadget(#Gadget, Status)

Versteckt oder zeigt ein Gadget. Ist der Status = 1, dann ist das Gadget versteckt. Ist der Status = 0, wird es angezeigt.

ImageGadget(#Gadget, x, y, Breite, Höhe, ImageID [, Flags])

Erstellt ein Image-Gadget (Bild-Gadget) innerhalb der aktuellen Gadgetliste. Die 'ImageID' repräsentiert eine BMP-Datei und kann einfach mittels der Befehle *ImageID()* bzw. *UseImage()* aus der Image Library ermittelt werden. Die Gadgetgröße passt sich automatisch der Bildgröße an. Das Gadget hat keinerlei Eingabefunktion, deshalb werden Benutzereingaben (Mausklicks etc...) nicht ausgewertet.

'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_Image_Border : stellt einen vertieften Rand rings um das Bild dar.

IPAddressGadget(#Gadget, x, y, Breite, Höhe)

Erstellt ein IPAddress Gadget innerhalb der aktuellen Gadgetliste. Dieses Gadget ermöglicht auf einfache Art und Weise die Eingabe einer kompletten IP-Adresse. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Die folgenden Befehle können in Zusammenhang mit diesem Gadget benutzt werden:

GetGadgetState(): Ermittelt die aktuelle IP-Adresse (benutzen Sie *IPAddressField()*, um den Wert jedes Felds zu ermitteln).

SetGadgetState(): Ändert die aktuelle IP-Adresse (benutzen Sie *MakeIPAddress()*, um eine gültige IP-Adresse zu erstellen).

GetGadgetText(): Gibt die aktuelle IP-Adresse in Textform (mit Punkten getrennt) zurück, zum Beispiel "127.0.0.1".

SetGadgetText(): Zur Zeit nur, um den Inhalt der IP-Adresse durch Übergabe eines Leerstrings zu löschen.

ListIconGadget(#Gadget, x, y, Breite, Höhe, Titel\$, TitelBreite [, Flags])

Erstellt ein ListIcon-Gadget innerhalb der aktuellen Gadgetliste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. 'Titel\$' ist der Titel der ersten Spalte und 'Breite' seine ursprüngliche Breite. Sobald ein ListIcon erstellt wurde, ist seine Liste mit Einträgen leer. Die folgenden Befehle können zum Zugriff auf den Listeninhalt benutzt werden:

AddGadgetItem() : Fügt einen Eintrag (mit optionalem Bild) hinzu

RemoveGadgetItem() : Entfernt einen Eintrag

ClearGadgetItemList() : Entfernt alle Einträge

CountGadgetItems() : Ermittelt die Anzahl der aktuellen Einträge im #Gadget.

GetGadgetItemState() : Ermittelt den aktuellen Status des angegebenen Eintrags.

SetGadgetItemState() : Ändert den aktuellen Status des angegebenen Eintrags.

GetGadgetItemText() : Ermittelt den aktuellen Text des angegebenen Eintrags.

SetGadgetItemText() : Ändert den aktuellen Text des angegebenen Eintrags.

Benutzen Sie *AddGadgetColumn()*, um eine Spalte zu diesem Gadget hinzufügen.

Benutzen Sie *ChangeListIconGadgetDisplay()*, um den Darstellungsmodus des ListIcons zu verändern, da vier verschiedenen Formen unterstützt werden (große Icons, kleine Icons, Listen und Reports).

'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_ListIcon_CheckBoxes	Zeigt Checkboxes in der ersten Spalte.
-------------------------	--

#PB_ListIcon_MultiSelect	Schaltet die Mehrfach-Selektion ein.
#PB_ListIcon_GridLines	Zeigt Separator-Linien an.
#PB_ListIcon_FullRowSelect	Die Auswahl erfolgt damit über die gesamte Zeile anstelle nur der ersten Spalte.
#PB_ListIcon_HeaderDragDrop	Die Reihenfolge der Spalten kann per Drag'n'Drop verändert werden.
#PB_ListIcon_AlwaysShowSelection	Auch wenn das Gadget nicht aktiv ist, die Auswahl ist weiterhin sichtbar.

ListViewGadget(#Gadget, x, y, Breite, Höhe)

Erstellt ein ListView Gadget (Auswahlliste) innerhalb der aktuellen Gadgetliste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Sobald ein ListView Gadget erstellt wurde, ist dessen Listeninhalt leer. Die folgenden Befehle können benutzt werden, um auf den Listen-Inhalt zuzugreifen:

AddGadgetItem(): fügt einen Eintrag hinzu

RemoveGadgetItem(): entfernt einen Eintrag

ClearGadgetItemList(): entfernt alle Einträge

GetGadgetState() kann zum Ermitteln des Indexes vom ausgewählten Eintrag verwendet werden.

SetGadgetState() kann zum Ändern des ausgewählten Eintrags verwendet werden.

Mit dem Befehl *EventType()* kann überprüft werden, ob der User beim Auswählen eines Listeneintrags einen Doppelklick ausgeführt hat.

OpenTreeGadgetNode()

Beim Einfügen von Einträgen zu einem *TreeGadget()* benutzen Sie *OpenTreeGadgetNode()*, um einen neuen "Node" (Knoten/Verzweigung) in der aktuellen Eintrags-Liste zu erstellen. Um diesen "Node" zu schließen und zum übergeordneten Eintrag zurückzukehren, benutzen Sie einfach *CloseTreeGadgetNode()*.

OptionGadget(#Gadget, x, y, Breite, Höhe, Text\$)

Erstellt ein Optionen-Gadget innerhalb der Gadgetliste. Beim ersten Aufruf dieser Funktion wird eine Gruppe erstellt und alle folgenden Aufrufe von *OptionGadget()* fügen dieser Gruppe ein Gadget hinzu. Um die Gruppe abzuschließen, rufen Sie einfach einen anderen Gadgettyp auf. Diese Art von Gadgets sind sehr nützlich, da zur gleichen Zeit immer nur ein Gadget der Gruppe selektiert sein kann.

PanelGadget(#Gadget, x, y, Breite, Höhe)

Erstellt ein Panel-Gadget innerhalb der Gadgetliste. Sobald ein Panel (Schalttafel) erstellt wurde, ist seine Liste mit Einträgen leer. Die folgenden Befehle können benutzt werden, um auf den Inhalt der Liste zuzugreifen:

AddGadgetItem(): fügt einen Eintrag hinzu

RemoveGadgetItem(): entfernt einen Eintrag

ClearGadgetItemList(): entfernt alle Einträge

SetGadgetState() kann zum Ändern des aktiven Panels verwendet werden.

GetGadgetState() kann zum Ermitteln des Indexes des aktuellen Panels verwendet werden.

Die nächsten Gadgets werden automatisch auf dem neuen Panel-Gadget Eintrag erstellt. Dies ist sehr bequem. Wenn das Panel-Gadget mit allen benötigten Gadgets gefüllt wurde, muss *ClosePanelGadget()* aufgerufen werden, um zur vorhergehenden Gadgetliste zurückzukehren. Dies bedeutet auch, dass ein Panel-Gadget auch auf einem anderen Panel-Gadget erstellt werden kann...

ProgressBarGadget(#Gadget, x, y, Breite, Höhe, Minimum, Maximum [, Flags])

Erstellt ein ProgressBar-Gadget (Fortschrittsanzeige) in der aktuellen Gadgetliste. Die Werte sind festgelegt zwischen 'Minimum' und 'Maximum'.

SetGadgetState() ändert den aktuellen Wert der Fortschrittsanzeige.

GetGadgetState() ermittelt den aktuellen Wert der Fortschrittsanzeige.

'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_ProgressBar_Smooth Die Fortschrittsanzeige erfolgt stufenlos anstelle der Benutzung von Blöcken.

#PB_ProgressBar_Vertical Die Fortschrittsanzeige erfolgt im vertikalen Modus.

RemoveGadgetItem(#Gadget, Position)

Entfernt einen Eintrag innerhalb des '#Gadget's an der angegebenen 'Position'. Der Positions-Index beginnt bei 0. Der Gadgettyp kann einer der folgenden sein:

ComboBoxGadget()

PanelGadget()

ListViewGadget()

ListIconGadget()

TreeGadget()

ResizeGadget(#Gadget, x, y, Breite, Höhe)

Ändert das angegebene #Gadget auf die neue Position und Größe. Um das Erstellen eines in Echtzeit größenveränderbaren ("realtime resizable") Graphical User Interface (GUI) zu vereinfachen, kann '-1' an jeden Parameter (x, y, Breite oder Höhe) übergeben werden und dieser Parameter wird dann nicht verändert. Dies ist sehr nützlich, denn oft-

mals soll ein Gadget nur in Breite oder Höhe verändert werden.

```
[...]  
ResizeGadget(0, -1, -1, 300, -1) ; Ändert nur die Gadget-  
breite.
```

SetGadgetFont(FontID)

Legt den Zeichensatz fest, der für neu erstellte Gadgets benutzt wird.

SetGadgetItemState(#Gadget, Eintrag, Status)

Ändert den Status des angegebenen 'Eintrag's im betreffenden #Gadget. Dies ist eine universelle Funktion, welche zusammen mit den meisten Gadgettypen, die mehrere Einträge verwalten, arbeitet. 'Status' kann folgende Werte annehmen:

ListViewGadget(): 1, wenn der Eintrag ausgewählt sein soll, sonst 0.

ListIconGadget(): 1, wenn der Eintrag ausgewählt sein soll, sonst 0.

TreeGadget(): Eine Kombination der folgenden Werte:

1: Der Eintrag ist ausgewählt, sonst 0.

2: Der Eintrag ist ausgeklappt (ein Baum-Pfad geöffnet), sonst 0.

Ist der 'Status' zum Beispiel 3, dann ist der Eintrag ausgewählt und ausgeklappt.

SetGadgetItemText(#Gadget, Eintrag, Text\$, Spalte)

Ändert den Text des angegebenen 'Eintrag's (in der 'Spalte') beim angegebenen '#Gadget'. Dies ist eine universelle Funktion, welche zusammen mit den meisten Gadgettypen, die mehrere Einträge verwalten, arbeitet:

ListViewGadget()

ListIconGadget()

PanelGadget()

ComboGadget()

TreeGadget()

SetGadgetState(#Gadget, Status)

Ändert den aktuellen Status des #Gadget. Dies ist eine universelle Funktion, welche mit den meisten der Gadgets arbeitet:

CheckBoxGadget(): 1 um es abzuheben, andernfalls 0.

ComboBoxGadget(): ändert den aktuell selektierten Eintrag.

ListViewGadget(): ändert den aktuell selektierten Eintrag.

OptionGadget(): 1 um es zu aktivieren, andernfalls 0.

PanelGadget(): ändert das aktuelle Panel.

ProgressBarGadget(): ändert den Status der Fortschrittsanzeige.

TrackBarGadget(): ändert die aktuelle Regler-Position.

SetGadgetText(#Gadget, Text\$)

Ändert den Inhalt des Gadget-Textes im angegebenen #Gadget. Dieser Befehl ist insbesondere nützlich bei:

StringGadget()

ComboBoxGadget()

TextGadget()

SpinGadget(#Gadget, x, y, Breite, Höhe, Minimum, Maximum)

Erstellt ein SpinGadget in der aktuellen Gadgetliste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Die folgenden Befehle können zum Verwalten des SpinGadget benutzt werden:

GetGadgetState(): Ermittelt den aktuellen Gadget-Wert.

SetGadgetState(): Ändert den aktuellen Gadget-Wert.

GetGadgetText(): Ermittelt den im Gadget enthaltenen Text.

SetGadgetText(): Ändert den im Gadget enthaltenen Text.

StringGadget(#Gadget, x, y, Breite, Höhe, Inhalt\$ [, Flags])

Erstellt ein String-Gadget innerhalb der aktuellen Gadget-Liste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurück-

gegeben wird. 'Inhalt\$' beinhaltet den anfänglichen Inhalt des String-Gadgets. Später kann der Inhalt mit den Befehlen *SetGadgetText()* und *GetGadgetText()* verändert werden.

'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_String_Password	Passwort-Modus, es werden nur '*' anstelle normaler Zeichen angezeigt.
#PB_String_ReadOnly	'Read only' Modus. Es kann kein Text eingegeben werden.
#PB_String_Numeric	Nur Zahlen werden akzeptiert.
#PB_String_LowerCase	Alle Zeichen werden automatisch in Kleinbuchstaben umgewandelt.
#PB_String_UpperCase	Alle Zeichen werden automatisch in Großbuchstaben umgewandelt.
#PB_String_BorderLess	Es werden keine Ränder rings um das Gadget gezeichnet.

TextGadget(#Gadget, x, y, Breite, Höhe, Text\$ [, Flags])

Erstellt ein Text-Gadget innerhalb der Gadget-Liste. Ein Text-Gadget ist ein einfacher Textbereich. Dessen Inhalt kann mit den Befehlen *SetGadgetText()* und *GetGadgetText()* geändert werden.

'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_Text_Center	Der Text wird im Gadget zentriert dargestellt.
#PB_Text_Right	Der Text wird rechtsbündig dargestellt.

#PB_Text_Border Ein vertiefter Rand wird rings um das Gadget gezeichnet.

TrackBarGadget(#Gadget, x, y, Breite, Höhe, Minimum, Maximum [, Flags])

Erstellt ein TrackBar Gadget in der aktuellen Gadgetliste. Dieses ermöglicht die Auswahl eines Wertebereichs mit Hilfe eines Schiebereglers, wie er in verschiedenen Multimedia-Playern zu finden ist. Der 'Minimum - Maximum' sollte zwischen 0 und 10.000 liegen. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Die folgenden Befehle können zum Zugriff auf dieses Gadget verwendet werden:

GetGadgetState(): Gibt die aktuelle Regler-Position (Wert innerhalb der Minimum-Maximum Spanne) zurück.

SetGadgetState(): Ändert die aktuelle Regler-Position.

'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:

#PB_TrackBar_Ticks Stellt einen 'Tick' Marker an jedem Schritt dar.

#PB_TrackBar_Vertical Das TrackBar ist jetzt vertikal.

TreeGadget(#Gadget, x, y, Breite, Höhe [, Flags])

Erstellt ein Tree-Gadget (Baum-Gadget) innerhalb der aktuellen Gadgetliste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Sobald ein Tree (Baum) erstellt wurde, ist seine Liste mit Einträgen leer. Die folgenden Befehle können benutzt werden, um auf den Listeninhalt zuzugreifen:

AddGadgetItem(): Fügt einen Eintrag (mit optionalem Bild) hinzu.

ClearGadgetItemList() : Entfernt alle Einträge.

<i>CountGadgetItems()</i>	: Ermittelt die Anzahl der aktuellen Einträge im #Gadget.
<i>GetGadgetItemState()</i>	: Ermittelt den aktuellen Status des angegebenen Eintrags.
<i>SetGadgetItemState()</i>	: Ändert den aktuellen Status des angegebenen Eintrags.
<i>GetGadgetItemText()</i>	: Ermittelt den aktuellen Textinhalt des angegebenen Eintrags.
<i>SetGadgetItemText()</i>	: Ändert den aktuellen Text des angegebenen Eintrags.
'Flags' sind optional und können sich aus einer oder mehreren der folgenden Konstanten zusammensetzen:	
#PB_Tree_AlwaysShowSelection	Auch wenn das Gadget nicht aktiviert ist, ist die Auswahl immer noch sichtbar.
#PB_Tree_NoLines	Versteckt die kleinen Linien zwischen allen "Nodes".
#PB_Tree_NoButtons	Versteckt die '+' "Node" Schalter.
#PB_Tree_CheckBoxes	Fügt eine Checkbox vor jedem Eintrag ein.

UseGadgetList(WindowID)

Macht das angegebene Fenster 'WindowID' zum aktuellen Fenster, auf welchem die Gadgetliste ihre Gadgets hinzufügt. Die WindowID kann einfach mit dem Befehl *WindowID()* ermittelt werden.

WebGadget(#Gadget, x, y, Breite, Höhe, URL\$)

Erstellt ein Web-Gadget in der aktuellen Gadgetliste. #Gadget ist die Nummer, die (später) von der *EventGadgetID()* Funktion zurückgegeben wird. Die folgenden Befehle können zum Zugriff auf das WebGad-

get benutzt werden:

SetGadgetText(): Ändert die aktuellen URL.

GetGadgetText(): Ermittelt die aktuelle URL.

SetGadgetState(): Führt einige Aktionen auf dem Gadget aus. Die folgenden Konstanten sind gültig:

#PB_Web_Back Ein Schritt zurück in der Navigation.

#PB_Web_Forward Ein Schritt vorwärts in der Navigation.

#PB_Web_Stop Stoppt das Laden der aktuellen Seite.

#PB_Web_Refresh Erneutes Laden ("Refresh") der aktuellen Seite.

Hinweis

Unter Microsoft Windows benutzt das WebGadget das Internet Explorer 4.0+ ActiveX Objekt. Dies bedeutet, dass IE installiert sein muss (standardmäßig installiert auf Win98/ME und Win2000/XP). Die ATL.dll (befindet sich im Verzeichnis „PureBasic\Compilers“) wird ebenfalls benötigt, und zwar im gleichen Verzeichnis wie das Executable.

16.25 Menu

Die Menüverwaltung in PureBasic ist sehr einfach. Natürlich können Sie auch alle möglichen Parameter einsetzen.

CloseSubMenu()

Schließt das aktuelle Unter-Menü und kehrt zum vorhergehenden zurück. *OpenSubMenu()* muss vor diesem Befehl aufgerufen worden sein!

CreateMenu(#Menu, WindowID)

Ergebnis = *CreateMenu*(#Menu, WindowID))

Erstellt ein neues leeres Menü auf dem angegebenen Fenster 'WindowID'. 'WindowID' kann einfach mit *WindowID()* Befehl aus der Window Library ermittelt werden. Unmittelbar nach dem Erstellen des Me-

nüs wird dieses das aktuelle Menü zum Einfügen weiterer Einträge. Es ist nun möglich, Befehle wie *MenuItem()*, *MenuBar()*, etc. zu benutzen. Ergibt der Rückgabewert ‚Ergebnis‘ gleich 0, dann ist das Erstellen des Menüs fehlgeschlagen und es können keine weitere Menübefehle benutzt werden.

CreatePopupMenu(#Menu)

Ergebnis = *CreatePopupMenu(#Menu)*

Erstellt ein neues leeres Popup-Menü. Es wird damit das aktuelle Menü zum Einfügen weiterer Einträge. Es ist nun möglich, Befehle wie *MenuItem()*, *MenuBar()*, etc. zu benutzen. *DisplayPopupMenu()* kann benutzt werden, um dieses Popup-Menü an einer beliebigen Position auf dem Bildschirm darzustellen. Ergibt der Rückgabewert ‚Ergebnis‘ gleich 0, dann ist das Erstellen des Menüs fehlgeschlagen und es können keine weitere Menübefehle benutzt werden.

DisableMenuItem(MenuItem, Status)

Deaktiviert oder aktiviert den Menüeintrag 'MenuItem'. Ist der angegebene Status gleich 1, wird der Menüeintrag deaktiviert. Ist der Status gleich 0, wird der Menüeintrag aktiviert.

DisplayPopupMenu(#Menu, WindowID() [, x, y])

Stellt ein zuvor erstelltes Popup-Menü unter der aktuellen Maus-Position dar. Optionale X,Y Positionen können angegeben werden. Die 'x' und 'y' Koordinaten werden in Pixel angegeben, gerechnet von der oberen linken Bildschirmecke.

FreeMenu(#Menu)

Entfernt das angegebene Menü aus dem Speicher. Nach dem Aufrufen dieser Funktion können Sie das Menü nirgends mehr anfügen, aber Sie können die #Menu Nummer für eine weitere Menülste nutzen.

GetMenuItemState(#Menu, MenuItem)

Ergebnis = GetMenuItemState(#Menu, MenuItem)

Gibt den aktuellen Status des 'MenuItem' zurück. Dies ist nur nützlich für Menüeinträge mit einer "Häkchen" Option. Ist der Menü-Eintrag "checked" (abgehakt), ergibt 'Ergebnis' gleich 1, andernfalls ist 'Ergebnis' gleich 0. Das Häkchen kann zu einem Menüeintrag mit dem *SetMenuItemState()* Befehl hinzugefügt werden.

MenuBar()

Erstellt einen Abgrenzungsbalken im aktuellen Menü.

MenuHeight()

Höhe = MenuHeight()

Gibt die Höhe (in Pixel) der Titelzeile vom Menü zurück. Die ist nützlich zur richtigen Berechnung der Fensterhöhe, wenn ein Menü benutzt wird.

MenuItem(MenuID, Text\$)

Erstellt einen Menü-Eintrag für das Menü. Im Text\$ können Sie das spezielle '&' Zeichen benutzen, um einen bestimmten Buchstaben zu unterstreichen:

"&Datei" ergibt in Wirklichkeit: Datei

MenuTitle(Title\$)

Erstellt einen neuen Titel-Eintrag für das Menü. Im Text\$ können Sie das spezielle '&' Zeichen benutzen, um einen bestimmten Buchstaben zu unterstreichen:

"&Datei" ergibt in Wirklichkeit: Datei

OpenSubMenu(Text\$)

Erstellt einen leeren Unter-Menüeintrag im aktuellen Menü. Im Text\$ können Sie das spezielle '&' Zeichen benutzen, um einen bestimmten Buchstaben zu unterstreichen:

"&Datei" ergibt in Wirklichkeit: Datei

SetMenuItemState(#Menu, MenuItem, Status)

Ändert den 'Status' des angegebenen Menüeintrag 'MenuItem'. Dies ist nützlich, um eine "Checkbox" nahe dem Text des Menüeintrag abzuhaben. Ist 'Status' gleich 1, dann wird das Häkchen dargestellt, andernfalls wird Häkchen entfernt. *GetMenuItemState()* kann benutzt werden, um den aktuellen Status des Menüeintrags zu ermitteln.

16.26 Requester

Computernutzer kennen alle die Requester, da fast alle Applikationen mit einer grafischen Benutzeroberfläche wenigstens einen davon benutzen. Sie sind sehr nützlich bei einigen grundlegenden Dingen (wie Öffnen einer Datei, Auswählen einer Farbe, Auswählen eines Zeichensatzes ect.), welche durch standardmäßige Auswahlfenster - genannt 'Requester' - sichergestellt werden.

ColorRequester()

Farbe = *ColorRequester()*

Öffnet einen Standard Requester zum Auswählen einer Farbe. Die ausgewählte Farbe wird als 24 Bit-Wert zurückgegeben, der wie üblich die Rot-, Grün- und Blau-Werte enthält. Um einfach die Werte der RGB-Komponenten zu ermitteln, benutzen Sie *Red()*, *Green()* und *Blue()*. Wenn der Benutzer den Requester abbricht, wird -1 zurückgegeben.

FontRequester(FontName\$, FontGröße, Flags)

Ergebnis = *FontRequester(FontName\$, FontGröße, Flags)*

Öffnet den Standard Requester zum Auswählen eines Zeichensatzes. Wenn der Benutzer den Requester abbricht, ist der Rückgabewert gleich 0, andernfalls beträgt dieser 1. 'FontName' und 'FontSize' können benutzt werden, um den Standard-Font festzulegen. 'Flags' werden noch nicht unterstützt und müssen 0 betragen (für zukünftige Kompatibilität). Die Befehle *SelectedFontColor()*, *SelectedFontName()* und *SelectedFontSize()* können nach einem erfolgreichen Aufruf benutzt werden, um die benötigten Informationen über den ausgewählten Zeichensatz zu erhalten.

MessageRequester(Titel\$, Text\$, Flags)

Ergebnis = *MessageRequester(Titel\$, Text\$, Flags)*

Öffnet einen ('blocking') Requester zum Anzeigen einiger Informationen. Die Programmausführung wird vollständig angehalten, bis der Benutzer den Requester schließt.

Als Flag kann einer der folgenden angegeben werden:

#PB_MessageRequester_YesNo	Um die 'Yes' (Ja) oder 'No' (Nein) Schalter zu erhalten
----------------------------	---

#PB_MessageRequester_YesNoCancel	Um die 'Yes' (Ja), 'No' (Nein) und 'Cancel' (Abbruch) Schalter zu erhalten
----------------------------------	--

#PB_MessageRequester_Ok	Um nur einen 'Ok' Schalter zu erhalten
-------------------------	--

OpenFileRequester(Titel\$, StandardDatei\$, Pattern\$, PatternPosition)

DateiName\$ = *OpenFileRequester(Titel\$, StandardDatei\$, Pattern\$, PatternPosition)*

Öffnet einen Standard Requester zum Auswählen einer Datei. Der 'Titel\$' kann angegeben werden, um den standardmäßigen Titel zu ersetzen. Der 'StandardDatei\$' ist nützlich, um den Requester mit dem richtigen Verzeichnis und dem richtigen Dateinamen zu initialisieren.

Der Pattern\$ ist ein Standard Filter, welcher nur die Anzeige von Dateien mit dieser oder jener Dateiendung erlaubt. Er muss in der folgenden Form angegeben werden: "Text-Dateien | *.txt | Musik-Dateien | *.mus;*.mod". Der Pattern arbeitet immer paarweise: Name (welcher wirklich im Filter erscheint) und Endung (z.B. *.txt). Mehrere für einen Typ mögliche Endungen können unter Benutzung des ; (Semikolon) angegeben werden.

Letztlich gibt die 'PatternPosition' an, welcher Pattern als Standard eingestellt sein soll. Mögliche Werte sind 0 bis zur Anzahl der Patterns.

PathRequester(Titel\$, UrsprungsPfad\$)

Pfad\$ = PathRequester(Titel\$, UrsprungsPfad\$)

Öffnet den Standard Verzeichnis-Requester zum Auswählen eines Verzeichnispfades. Der 'Titel\$' kann angegeben werden, um den standardmäßigen Titel zu ersetzen. Der 'UrsprungsPfad\$' ist nützlich, um den Requester mit dem richtigen Verzeichnis zu initialisieren. Der Verzeichnispfad wird mit dem abschließenden '\ ' zurückgegeben.

SaveFileRequester(Titel\$, StandardDatei\$, Pattern\$, PatternPosition)

DateiName\$ = SaveFileRequester(Titel\$, StandardDatei\$, Pattern\$, PatternPosition)

Öffnet einen Standard Requester zum Auswählen einer zu speichernden Datei. Der 'Titel\$' kann angegeben werden, um den standardmäßigen Titel zu ersetzen. Der 'StandardDatei\$' ist nützlich, um den Requester mit dem richtigen Verzeichnis und dem richtigen Dateinamen zu initialisieren.

Der Pattern\$ ist ein Standard Filter, welcher nur die Anzeige von Dateien mit dieser oder jener Dateiendung erlaubt. Er muss in der folgenden Form angegeben werden: "Text-Dateien | *.txt | Musik-Dateien | *.mus;*.mod". Der Pattern arbeitet immer paarweise: Name (welcher wirklich im Filter erscheint) und Endung (z.B. *.txt). Mehrere für einen Typ mögliche Endungen können unter Benutzung des ; (Semikolon) angegeben werden.

Letztlich gibt die 'PatternPosition' an, welcher Pattern als Standard eingestellt sein soll. Mögliche Werte sind 0 bis zur Anzahl der Patterns.

SelectedFontColor()

Farbe = SelectedFontColor()

Gibt die vom Benutzer im *FontRequester()* ausgewählte Farbe zurück. Die Farbe ist ein 24 Bit-Wert, welcher die Rot-, Grün- und Blau-Informationen enthält. Um die individuellen RGB-Farbwerte zu erhalten, benutzen Sie einfach *Red()*, *Green()* und *Blue()*.

SelectedFontName()

Name\$ = SelectedFontName()

Gibt den Namen des vom Benutzer im *FontRequester()* ausgewählten

Zeichensatzes zurück. Dieser Name kann direkt vom *LoadFont()* Befehl verwendet werden.

SelectedFontSize()

Größe = *SelectedFontSize()*

Gibt die Größe des vom Benutzer im *FontRequester()* ausgewählten Zeichensatzes zurück.

SelectedFontStyle()

Stil = *SelectedFontStyle()*

Gibt den Stil des vom Benutzer im *FontRequester()* ausgewählten Zeichensatzes zurück. 'Stil' kann einen oder mehrere der folgenden Werte enthalten:

#PB_Font_Bold : Zeichensatz ist "fett".

#PB_Font_Italic : Zeichensatz ist "kursiv".

Um zu testen, ob ein Wert vorhanden ist, benutzen Sie den '&' (bitweiser 'AND', zu deutsch: UND) Operator:

```
If SelectedFontStyle() & #PB_Font_Bold
    ; Der Zeichensatz ist fett
EndIf
```

16.27 ToolBar

Die Toolbars (Werkzeuggestreifen) sind sehr nützlich, um mit Hilfe von kleinen Icons schnellen Zugriff auf einige Funktionen der Applikation zu erhalten. Die Icons sind oftmals 'Shortcuts' von Menüpunkten. PureBasic ermöglicht die Erstellung einer beliebigen Zahl von Toolbars und deren Handhabung als wären es Menüs.

CreateToolBar(#ToolBar, WindowID)

Ergebnis = *CreateToolBar(#ToolBar, WindowID)*

Erstellt eine leere Toolbar auf dem angegebenen Fenster 'WindowID', welche über die Nummer '#ToolBar' identifiziert wird. Die 'WindowID' kann einfach mittels dem *WindowID()* Befehl ermittelt werden. Diese Toolbar wird die aktuelle Toolbar für die Erstellung, zu der mittels der

Befehle *ToolBarStandardButton()* und *ToolBarSeparator()* einige Elemente hinzugefügt werden können. Ist 'Ergebnis' gleich 0, konnte die Toolbar nicht erstellt werden, andernfalls ist alles in Ordnung.

DisableToolBarButton(#ButtonID, Status)

Aktiviert oder deaktiviert den angegebenen Toolbar-Schalter: Ist 'Status' gleich 0, wird der Schalter aktiviert. Ist 'Status' gleich 1, wird der Schalter deaktiviert.

FreeToolBar(#ToolBar)

Gibt die angegebene '#ToolBar' frei.

ToolBarImageButton(#ButtonID, ImageID)

Fügt einen Image-Schalter (Bild-Schalter) zur konstruierten Toolbar hinzu. *CreateToolBar()* muss vor Benutzung dieses Befehls aufgerufen werden. Die 'ImageID' kann einfach mittels *UseImage()* bzw. *ImageID()* aus der Image Library ermittelt werden. Sie kann entweder eine Icon oder BMP Datei sein.

ToolBarSeparator()

Fügt der zu konstruierenden Toolbar einen vertikalen Balken (Separator) hinzu. Vor diesem Befehl muss *CreateToolBar()* aufgerufen worden sein.

ToolBarStandardButton(#ButtonID, #ButtonIcon)

Fügt der zu konstruierenden Toolbar einen Standard-Schalter hinzu. Vor diesem Befehl muss *CreateToolBar()* aufgerufen worden sein. Ein Standard-Schalter ist ein Icon, welches direkt im OS verfügbar ist.

Der '#ButtonIcon' Parameter muss eine der folgenden Konstanten sein:

#PB_ToolBarIcon_New	#PB_ToolBarIcon_Copy
#PB_ToolBarIcon_Open	#PB_ToolBarIcon_Paste
#PB_ToolBarIcon_Save	#PB_ToolBarIcon_Undo
#PB_ToolBarIcon_Print	#PB_ToolBarIcon_Redo
#PB_ToolBarIcon_Find	#PB_ToolBarIcon_Delete
#PB_ToolBarIcon_Replace	#PB_ToolBarIcon_Properties
#PB_ToolBarIcon_Cut	#PB_ToolBarIcon_Help

ToolBarToolTip(#ButtonID, Text\$)

Verknüpft den angegebenen 'Text\$' mit der aktuellen ToolBar '#ButtonID'. Ein ToolTip-Text ist ein Text, der (in einer kleinen gelben Box) angezeigt wird, wenn der Mauszeiger eine Weile über einem Gadget verharret.

16.28 StatusBar

Eine Status-Leiste (status bar) ist die untere Leiste eines Fensters, worauf einige Informationen angezeigt werden. Diese Leiste ist immer sichtbar und kann in mehrere Abschnitte aufgeteilt werden.

AddStatusBarField(Breite)

Fügt ein Feld zur aktuellen Statusleiste hinzu, die zuvor mittels *CreateStatusBar()* erstellt wurde. 'Breite' ist die Breite des neuen Felds in Pixel. Jedes neue Feld wird nach dem alten Feld erstellt. *SetStatusBarText()* kann benutzt werden, um jedes Feld mit einem Text zu füllen.

CreateStatusBar(#StatusBar, WindowID)

Ergebnis = *CreateStatusBar*(#StatusBar, WindowID)

Erstellt und fügt eine leere Statusleiste zur angegebenen WindowID hinzu. Ergibt 'Ergebnis' gleich 0, ist die Erstellung der Statusleiste fehlgeschlagen, andernfalls ist alles in Ordnung. Sobald die Statusleiste erstellt wurde, kann *AddStatusBarField()* zum Definieren der verschiedenen Abschnitte der Leiste benutzt werden.

FreeStatusBar(#StatusBar)

Gibt die angegebene Statusleiste frei.

StatusBarIcon(#StatusBar, Feld, ImageID)

Fügt ein Icon zum angegebenen 'Feld' hinzu. Das Icon wird links vom Feldtext angezeigt. Das Icon muss im 16x16 Format vorliegen. 'ImageID' kann einfach mittels *LoadImage()* oder *UseImage()* ermittelt werden.

StatusBarText(#StatusBar, Feld, Text\$, Aussehen)

Ändert den Text des angegebenen 'Feld's der '#StatusBar'. 'Aussehen' wird benutzt, um das Aussehen des Felds zu verändern und kann folgende Werte annehmen:

- 0 : Standard, gedrückt mit Rändern
- 1 : ohne Ränder
- 2 : "erhobene" Ränder
- 4 : zentriert den Text im Feld
- 8 : richtet den Text am rechten Rand des Felds aus

Die Optionen können mittels dem '|' (OR) Operator kombiniert werden:

8 | 1 ergibt ein randloses Feld mit dem Text am rechten Rand ausgerichtet.

UpdateStatusBar(#StatusBar)

Passt die Größe der angegebenen Statusleiste an die (neuen) Dimensionen des zugehörigen Fensters an. Dieser Befehl wird typischerweise nach einem #PB_Event_SizeWindow aufgerufen.

16.29 Clipboard

Das Clipboard (die "Zwischenablage") ist der Standard-Weg, um Informationen und Daten zwischen gerade auf dem OS laufenden Applikationen auszutauschen. Zum Beispiel wird ein Text beim Ausschneiden im Text-Editor ins Clipboard kopiert und kann später von jedem anderen Text-Editor wieder aufgerufen und eingefügt werden. PureBasic ermöglicht auf einfache Art und Weise das Kopieren und Empfangen von jeder Art Daten über das Standard-Clipboard.

ClearClipboard()

Löscht das Clipboard. Das bedeutet, dass alle vorher kopierten Daten restlos gelöscht werden und nicht mehr verfügbar sind.

GetClipboardData(Datentyp)

Daten = *GetClipboardData(Datentyp)*

Gibt eine Kopie der gerade im Clipboard befindlichen Daten zurück.
Zurzeit wird nur ein Datentyp unterstützt:

#PB_ClipboardImage: welches die *ImageID()* eines zuvor kopierten Bildes zurückgibt.

GetClipboardText()

Text\$ = *GetClipboardText()*

Gibt den gerade im Clipboard befindlichen Text\$ zurück.

SetClipboardData(Datentyp, DataID)

Übermittelt Daten an das Clipboard. Die DataID muss eine gültige Objekt-ID sein.

Zurzeit wird nur ein Datentyp unterstützt:

#PB_ClipboardImage: welches das angegebene Bild in das Clipboard kopiert. Um eine ImageID zu erhalten, benutzen Sie einfach die *ImageID()* Funktion.

SetClipboardText(Text\$)

Kopiert einen Text\$ in das Clipboard. Enthält das Clipboard bereits einen Text, wird dieser überschrieben.

16.30 Preference

Preference ("Voreinstellungs-") Dateien beinhalten vom Anwender definierte Programmparameter, die auf Disk gespeichert und bei einem erneuten Programmstart wieder eingelesen werden (wie die '.INI' Dateien unter Windows zum Beispiel). PureBasic bietet die Möglichkeit, hierarchische Preference-Dateien zu erstellen, die sehr einfach auf allen Computerplattformen einsetzbar sind. Das Dateiformat ist in ASCII

gehalten, mit einer Einstellung pro Zeile (Benutzung des 'Schlüsselwort=Wert' Syntax). Gruppen können zur besseren Lesbarkeit erstellt werden.

ClosePreferences()

Schliesst eine zuvor mit *OpenPreferences()* geöffnete oder mit *CreatePreferences()* erstellte Preference-Datei.

CreatePreferences(DateiName\$)

Ergebnis = *CreatePreferences(DateiName\$)*

Erstellt eine neue leere Preference-Datei. Existiert diese Datei bereits, so wird diese gelöscht. Wird als ‚Ergebnis‘ gleich 0 zurückgegeben, konnte die Datei nicht erstellt werden. Sobald die Datei erfolgreich erstellt wurde, können folgende Befehle zum Schreiben von Daten genutzt werden: *WritePreferenceString()*, *WritePreferenceLong()*, *WritePreferenceFloat()*, *PreferenceGroup()* und *PreferenceComment()*.

Sobald alle Schreib-Operationen erfolgten, wird *ClosePreference()* zum Schließen der Preference-Datei genutzt.

OpenPreferences(DateiName\$)

Ergebnis = *OpenPreferences(DateiName\$)*

Öffnet eine bereits existierende Preference-Datei. Ist das ‚Ergebnis‘ gleich 0, dann konnte die Datei nicht geöffnet werden. Wenn auch die Datei nicht geöffnet werden konnte, kann der Programmierer trotzdem die Lese-Befehle nutzen, die dann jeweils die spezifizierten Standardwerte zurückgeben. Dies ist sehr nützlich, um in einem Schritt die Programmvariablen zu initialisieren.

Die folgenden Befehle können zum Lesen von Daten genutzt werden:

ReadPreferenceString(), *ReadPreferenceLong()*, *ReadPreferenceFloat()* und *PreferenceGroup()*.

Sobald alle Lese-Operationen erfolgten, wird *ClosePreferences()* zum Schließen der Preference-Datei genutzt.

PreferenceGroup(Name\$)

Ergebnis = *PreferenceGroup*(Name\$)

Befindet sich die Preference-Datei im Schreib-Modus (nach einem *CreatePreferences*() Befehl), dann wird eine neue Gruppe in der Form: [GruppenName\$] erstellt. Alle folgenden Schreib-Operationen beziehen sich dann auf diese Gruppe.

Befindet sich die Preference-Datei im Lese-Modus (nach einem *OpenPreferences*() Befehl), dann wird der aktuelle Lesezeiger auf den Start der angegebenen Gruppe verschoben. Alle folgenden Lese-Operationen sind dann auf diese Gruppe beschränkt.

PreferenceComment(Text\$)

Ergebnis = *PreferenceComment*(Text\$)

Schreibt eine neue Kommentar-Zeile in die gerade erstellte Preference-Datei.

ReadPreferenceFloat(Keyword\$, StandardWert.f)

Ergebnis.f = *ReadPreferenceFloat*(Keyword\$, StandardWert.f)

Versucht den zum Schlüsselwort 'Keyword\$' gehörenden Wert (im Float '.f' Format) zu lesen. Wenn *PreferenceGroup*() benutzt wurde, dann beschränkt sich die Suche auf die aktuelle Gruppe. Wurde das Schlüsselwort nicht gefunden oder die Preference-Datei nicht korrekt geöffnet (Datei fehlt zum Beispiel), dann wird der angegebene 'StandardWert' zurückgegeben.

ReadPreferenceLong(Keyword\$, StandardWert)

Ergebnis.l = *ReadPreferenceLong*(Keyword\$, StandardWert)

Versucht den zum Schlüsselwort 'Keyword\$' gehörenden Wert (im Long '.l' Format) zu lesen. Wenn *PreferenceGroup*() benutzt wurde, dann beschränkt sich die Suche auf die aktuelle Gruppe. Wurde das Schlüsselwort nicht gefunden oder die Preference-Datei nicht korrekt geöffnet (Datei fehlt zum Beispiel), dann wird der angegebene 'StandardWert' zurückgegeben.

ReadPreferenceString(Keyword\$, StandardWert\$)

`Ergebnis$ = ReadPreferenceString(Keyword$, StandardWert$)`

Versucht den zum Schlüsselwort 'Keyword\$' gehörenden String zu lesen. Wenn *PreferenceGroup()* benutzt wurde, dann beschränkt sich die Suche auf die aktuelle Gruppe. Wurde das Schlüsselwort nicht gefunden oder die Preference-Datei nicht korrekt geöffnet (Datei fehlt zum Beispiel), dann wird der angegebene 'StandardWert\$' zurückgegeben.

WritePreferenceFloat(Keyword\$, Wert.f)

Schreibt das angegebene Schlüsselwort 'Keyword\$' und dessen zugehörigen Wert (im Float '.f' Format) in der Form 'Keyword\$ = Wert' in die Preference-Datei, die zuvor mit *CreatePreferences()* erstellt wurde.

WritePreferenceLong(Keyword\$, Wert)

Schreibt das angegebene Schlüsselwort 'Keyword\$' und dessen zugehörigen Wert (im Long '.l' Format) in der Form 'Keyword\$ = Wert' in die Preference-Datei, die zuvor mit *CreatePreferences()* erstellt wurde.

WritePreferenceString(Keyword\$, Wert\$)

Schreibt das angegebene Schlüsselwort 'Keyword\$' und dessen zugehörigen Wert (im String '.s' Format) in der Form 'Keyword\$ = Wert' in die Preference-Datei, die zuvor mit *CreatePreferences()* erstellt wurde.

16.31 Printer

Drucker sind essentielle Schnittstellen, um virtuelle, numerische Daten auf Papier auszugeben. Viele Software benötigt einen Weg, Daten auf Papier auszudrucken, um wirklich nützlich zu sein. PureBasic erlaubt jede Art von Daten - von einfachem Text bis zu Bildern - in jeder Auflösung zu drucken.

NewPrinterPage()

Erstellt eine neue leere Seite ("ein virtuelles Blatt Papier"). Die vorherige Seite wird an den Drucker geschickt und kann nicht mehr verändert werden.

PrinterOutput()

OutputID = *PrinterOutput()*

Gibt die OutputID des aktuellen Druckers zurück, die zusammen mit dem *StartDrawing()* Befehl benutzt werden kann. Alle 2DDrawing Befehle sind möglich und können zum Ausdrucken benutzt werden. Der Drucker wird wie ein leeres Blatt Papier angesehen, genau wie ein Fenster oder ein Bild.

PrintRequester()

Ergebnis = *PrintRequester()*

Öffnet den regulären Drucker-Requester, um den Drucker auszuwählen und einige Einstellungen zu treffen. Dieser Befehl muss vor allen anderen Drucker-Befehlen aufgerufen werden. Ist das ‚Ergebnis‘ gleich 0, dann ist entweder kein Drucker verfügbar oder der Benutzer hat den Requester abgebrochen. Sobald *PrintRequester()* erfolgreich aufgerufen wurde, wird *StartPrinting()* benutzt, um das Ausdrucken zu beginnen.

PrinterPageHeight()

Höhe = *PrinterPageHeight()*

Gibt die Höhe (in Pixel) des Zeichen-Bereichs zurück. Der Wert ändert sich mit der Auflösung (DPI) des Druckers. Dies bedeutet, dass ein Dokument, das mit 75 DPI gedruckt wird, einen Zeichenbereich hat, der 4 mal kleiner ist als beim Druck eines Dokuments mit 150 DPI.

PrinterPageWidth()

Breite = *PrinterPageWidth()*

Gibt die Breite (in Pixel) des Zeichen-Bereichs zurück. Der Wert ändert sich mit der Auflösung (DPI) des Druckers. Dies bedeutet, dass ein Dokument, das mit 75 DPI gedruckt wird, einen Zeichenbereich hat, der 4-mal kleiner ist als beim Druck eines Dokuments mit 150 DPI.

StartPrinting(JobName\$)

Ergebnis = StartPrinting(JobName\$)

Initialisiert den Drucker und startet die Druck-Operation. 'JobName\$' ist der Name, welcher im Drucker-Spooler angezeigt wird und den Druck identifiziert. Ist das 'Ergebnis' gleich 0, konnte die Druck-Operation nicht gestartet werden.

StopPrinting()

Stoppt alle Druck-Operationen und sendet die Daten an den Drucker.

16.32 Network

Netzwerke sind auf der ganzen Welt weit verbreitet und erlauben die einfache Kommunikation zwischen Computern. PureBasic unterstützt das offizielle Internet-Protokoll zum Datenaustausch: TCP/IP. Diese Bibliothek erlaubt Ihnen das Erstellen von Applikationen und Spielen, die dieses Protokoll mittels des allseits bekannten "Client-Server-Modells" unterstützen. Mit diesen Befehlen ist es möglich, jede Art von Applikationen mit Internetanbindung (Browser, Web-Server, FTP-Client...) oder schnellen 'Multiplayer' Spielen zu schreiben.

InitNetwork()

Ergebnis = InitNetwork()

Dieser Befehl muss vor allen anderen Befehlen der Network Library aufgerufen werden. Ergibt 'Ergebnis' gleich 0, ist kein TCP/IP-Stack auf dem System verfügbar, andernfalls wurde alles korrekt initialisiert.

CloseNetworkConnection(ConnectionID)

Schließt die angegebene Client-Verbindung 'ConnectionID'.

CloseNetworkServer()

Schließt den aktuell laufenden Server. Alle mit diesem Server verbundenen Clients werden automatisch entfernt. Der Port wird freigegeben und kann wieder von einer anderen Applikation genutzt werden.

CreateNetworkServer(Port)

Ergebnis = CreateNetworkServer(Port)

Erstellt einen neuen Netzwerk-Server auf dem lokalen Computer an dem angegebenen Port. Die Port-Werte reichen von 6000 bis 7000 (dies ist der empfohlene Bereich, in Wirklichkeit können die Werte im Bereich von 0 bis 16000 liegen). Eine beliebige Anzahl Server kann gleichzeitig auf demselben Computer laufen, jedoch nicht mit derselben Port-Nummer. Ergibt 'Ergebnis' gleich 0, konnte der Server nicht eingerichtet werden (die Port-Nummer wird bereits benutzt). Andernfalls wurde der Server korrekt eingerichtet und ist bereit zur Benutzung.

Port: Port-Nummer für diesen Server

IPAddressField(IPAdresse, Feld)

Ergebnis = IPAddressField(IPAdresse, Feld)

Gibt den Wert des angegebenen Felds der angegebenen IP-Adresse zurück. 'IpAdresse' kann ein mittels *MakeIPAddress()* erstellter Wert sein. 'Feld' kann zwischen 0 und 3 (von rechts nach links) liegen. Zum Beispiel "127.0.0.1": das Feld 0 ist gleich '1' und das Feld 3 ist gleich '127'. Dieser Befehl ist insbesondere nützlich in Zusammenhang mit folgenden Funktionen:

IPAddressGadget() (siehe Gadget Library)

MakeIPAddress()

MakeIPAddress(Feld3, Feld2, Feld1, Feld0)

Ergebnis = MakeIPAddress(Feld3, Feld2, Feld1, Feld0)

Gibt den entsprechenden Wert der mit dieser Funktion spezifizierten IP-Adresse zurück. Jedes Feld kann einen Wert zwischen 0 und 255 haben. Die Befehl ist besonders nützlich zusammen mit:

IPAddressGadget()

NetworkClientID()

ClientID = NetworkClientID()

Dieser Befehl wird nur auf der Server-Seite benötigt. Er ermöglicht die Ermittlung des Clients, der die Daten gesendet hat.

NetworkClientEvent(ConnectionID)

Ergebnis = *NetworkClientEvent(ConnectionID)*

Gibt 0 zurück, wenn keine Informationen (oder Daten) über den angegebenen Client-Port 'ConnectionID' empfangen wurden, andernfalls liegen welche vor, die verarbeitet werden sollten.

Der Rückgabewert zeigt an, was gerade passierte:

0 : Nichts ist passiert

2 : Roh-Daten wurden empfangen (können gelesen werden mittels *ReceiveNetworkData()*)

3 : Eine Datei wurde empfangen (kann gelesen werden mittels *ReceiveNetworkFile()*)

NetworkServerEvent()

Ergebnis = *NetworkServerEvent()*

Gibt 0 zurück, wenn keine Informationen (oder Daten) über den Server-Port empfangen wurden, andernfalls liegen welche vor, die verarbeitet werden sollten.

Der Rückgabewert zeigt an, was gerade passierte:

0 : Nichts ist passiert

1 : Ein neuer Client wurde mit dem Server verbunden

2 : Roh-Daten wurden empfangen (können gelesen werden mittels *ReceiveNetworkData()*)

3 : Eine Datei wurde empfangen (kann gelesen werden mittels *ReceiveNetworkFile()*)

4 : Ein Client hat den Server verlassen (Verbindungstrennung)

OpenNetworkConnection(ServerName\$, Port)

ConnectionID = *OpenNetworkConnection(ServerName\$, Port)*

Versucht eine Verbindung mit dem angegebenen Server aufzubauen. 'ServerName\$' kann eine IP-Adresse oder ein voller Name sein (z.B.: "127.0.0.1" oder "ftp.home.net"). Wenn die Verbindung vom Server gewährt wurde, ist die 'ConnectionID' ungleich NULL, andernfalls ist der Verbindungsaufbau fehlgeschlagen. 'ConnectionID' wird von allen

Befehlen zum Empfangen und Senden benötigt.

ServerName\$: Name oder IP-Adresse des Computers, der den Server hostet, mit dem verbunden werden soll.

Port: Port-Nummer des laufenden Servers (siehe *CreateNetworkServer()*).

ReceiveNetworkData(ConnectionID, *DataBuffer, Länge)

Ergebnis = *ReceiveNetworkData(ConnectionID, *DataBuffer, Länge)*

Empfängt die Roh-Daten vom angegebenen Client. Dieser Befehl kann von beiden, Client- und Server-Applikationen, genutzt werden. Auf der Server-Seite ist 'ConnectionID' der Client, welcher die Daten gesendet hat (die ClientID kann einfach mittels *NetworkClientID()* ermittelt werden). Auf der Client-Seite wird die 'ConnectionID' von *OpenNetworkConnection()* zurückgegeben. Die Daten werden in den angegebenen Speicherbereich '*DataBuffer' gelesen. 'Ergebnis' gibt die Anzahl an bereits gelesenen Bytes an. Ist 'Ergebnis' gleich 'Länge', dann sind noch mehr Daten einzulesen.

ReceiveNetworkFile(ConnectionID, Dateiname\$)

Empfängt eine Datei vom angegebenen Client. Dieser Befehl kann von beiden, Client- und Server-Applikationen, genutzt werden. Auf der Server-Seite ist 'ConnectionID' der Client, welcher die Datei gesendet hat. Auf der Client-Seite benutzen Sie die von *OpenNetworkConnection()* zurückgegebene 'ConnectionID'. Die Datei muss mittels des speziellen *SendNetworkFile()* Befehls gesendet worden sein.

SendNetworkData(ConnectionID, *MemoryBuffer, Länge)

Sendet Roh-Daten zum angegebenen Client. Dieser Befehl kann von beiden, Client- und Server-Applikationen, genutzt werden. Auf der Server-Seite ist 'ConnectionID' der Client, welcher die Daten empfangen soll. Auf der Client-Seite benutzen Sie die von *OpenNetworkConnection()* zurückgegebene 'ConnectionID'.

SendNetworkFile(ConnectionID, Dateiname\$)

Sendet eine komplette Datei zum angegebenen Client. Dieser Befehl kann von beiden, Client- und Server-Applikationen, genutzt werden. Auf der Server-Seite ist 'ConnectionID' der Client, welcher die Daten empfangen soll. Auf der Client-Seite benutzen Sie die von *OpenNetworkConnection()* zurückgegebene 'ConnectionID'. Die Datei wird mittels sehr spezieller (und sicherer) Methoden gesendet. Sie muss mittels dem *RecieveNetworkFile()* Befehl empfangen werden. Dieser Befehl blockiert die Programmausführung bis die komplette Datei gesendet wurde.

SendNetworkString(ConnectionID, String\$)

Sendet einen String zum angegebenen Client. Dieser Befehl kann von beiden, Client- und Server-Applikationen, genutzt werden. Auf der Server-Seite ist 'ConnectionID' der Client, welcher die Daten empfangen soll. Auf der Client-Seite benutzen Sie die von *OpenNetworkConnection()* zurückgegebene 'ConnectionID'.

16.33 Help

Die Hilfe ist ein sehr bedeutender Bestandteil jeder Software. Sie erlaubt dem Anwender das einfache und effiziente Entdecken der angebotenen Software-Features. PureBasic ermöglicht das Anzeigen von Standard Hilfe-Dateien mit "Global" und "Kontext" Unterstützung.

Unter Microsoft Windows werden zwei Formate von Hilfe-Dateien unterstützt: .hlp (altes Format) und .chm (HTML-basierend, neues Format).

CloseHelp()

Schließt die zuvor mit *OpenHelp()* geöffnete Hilfe.

OpenHelp(Dateiname\$, Thema\$)

Öffnet ein Hilfe-Fenster und stellt dieses dar. Ist der 'Thema\$' keine Null, stellt die Hilfe direkt die entsprechende Seite dar (sehr nützlich für Online/kontext-sensitive Hilfe). Unter Microsoft Windows werden zwei Formate von Hilfe-Dateien unterstützt: .hlp (altes Format) und .chm (HTML-basierend, neues Format).

16.34 Database

Die Database Library ist eine einfache Sammlung von Befehlen, um auf jeden Datenbank-Typ (Oracle, MySQL, Access, etc...) zuzugreifen, der die bekannte ODBC API benutzt. Die Library basiert auf SQL-Abfragen, um Daten aus einer Datenbank zu erhalten oder hinein zu schreiben. Deshalb kann es interessant für Sie sein, einen kurzen Blick in ein Dokument mit einer SQL-Beschreibung zu schauen, falls dies nötig sein sollte.

InitDatabase()

Ergebnis = *InitDatabase()*

Initialisiert die Datenbank-Programmumgebung zur weiteren Benutzung. Es wird versucht, die ODBC Library zu laden und die benötigten Ressourcen zu reservieren. Ist das Ergebnis gleich 0, dann ist ODBC nicht verfügbar oder zu alt (ODBC 3.0 oder besser wird benötigt) und alle Datenbank Funktionsaufrufe müssen deaktiviert werden.

CloseDatabase(#Database)

Schließt die angegebene Datenbank '#Database' (und ggf. offene Verbindungen oder Transaktionen). Damit sind keine weiteren Operationen in dieser Datenbank erlaubt.

DatabaseColumnName(Spalte)

Text\$ = *DatabaseColumnName(Spalte)*

Gibt den Namen der angegebenen Spalte (Feld) in der aktuellen Datenbank zurück.

DatabaseColumnType(Spalte)

Typ = *DatabaseColumnType(Spalte)*

Gibt den Typ der angegebenen Spalte (Feld) in der aktuellen Datenbank zurück. Ist der Rückgabewert gleich 0, ist der Typ nicht definiert oder der Befehl ist fehlgeschlagen (es war unmöglich den Typ zu ermitteln). Typen-Werte können sein:

- 1 Numerisches Format: Long (.l) unter PureBasic
- 2 String-Format: String (.s) unter PureBasic
- 3 Numerisches Fließkomma-Format:Float (.f) unter PureBasic

DatabaseColumns()

Anzahl = *DatabaseColumns()*

Ermittelt die Anzahl an Spalten (Felder) in der gerade geöffneten Datenbank.

DatabaseDriverDescription()

Text\$ = *DatabaseDriverDescription()*

Gibt die Beschreibung des aktuellen Datenbank-Treibers zurück. Die Treiber werden aufgelistet mit den *ExamineDatabaseDrivers()* und *NextDatabaseDriver()*Befehlen.

DatabaseDriverName()

Name\$ = *DatabaseDriverName()*

Gibt den Namen des aktuellen Datenbank-Treibers zurück. Die Treiber werden aufgelistet mit den *ExamineDatabaseDrivers()* und *NextDatabaseDriver()*Befehlen.

DatabaseQuery(Request\$)

Ergebnis = *DatabaseQuery(Request\$)*

Führt die SQL-Abfrage 'Request\$' in der aktuellen Datenbank aus. Ist das 'Ergebnis' gleich 0, konnte die Abfrage nicht korrekt ausgeführt werden (SQL-Fehler, falsch formatierte Abfrage etc...), andernfalls war die Abfrage erfolgreich und *NextDatabaseRow()* kann zur Auflistung der ermittelten Einträge benutzt werden.

ExamineDatabaseDrivers()

Ergebnis = *ExamineDatabaseDrivers()*

Untersucht die auf dem System verfügbaren installierten Datenbank-Treiber. Wenn ODBC nicht installiert ist oder keine Treiber vorhanden sind, wird 0 zurückgegeben. Sonst kann *NextDatabaseDriver()* zum Auflisten der Treiber benutzt werden.

FirstDatabaseRow()

Ergebnis = *FirstDatabaseRow()*

Gibt eine Information über die ersten Datenbank-Zeile zurück. Ist das ‚Ergebnis‘ gleich 0, dann ist keine (Datenbank-Zeile (kein Eintrag) verfügbar. Um auf die Informationen innerhalb des Eintrags (bzw. der Zeile) zuzugreifen, benutzen Sie *GetDatabaseLong()*, *GetDatabaseFloat()* und *GetDatabaseString()*.

GetDatabaseFloat(Spalte)

Ergebnis.f = *GetDatabaseFloat(Spalte)*

Gibt den Inhalt des angegebenen Feldes 'Spalte' als Fließkomma-Zahl (Float) zurück. Um den Typ eines Feldes zu ermitteln, kann *DatabaseColumnType()* benutzt werden.

GetDatabaseLong(Spalte)

Ergebnis.l = *GetDatabaseLong(Spalte)*

Gibt den Inhalt des angegebenen Feldes 'Spalte' als Standard-Zahl (Long) zurück. Um den Typ eines Feldes zu ermitteln, kann *DatabaseColumnType()* benutzt werden.

GetDatabaseString(Spalte)

Ergebnis\$ = *GetDatabaseString(Spalte)*

Gibt den Inhalt des angegebenen Feldes 'Spalte' als String zurück. Um den Typ eines Feldes zu ermitteln, kann *DatabaseColumnType()* benutzt werden.

NextDatabaseDriver()

Ergebnis = *NextDatabaseDriver()*

Ermittelt die Informationen über den nächsten verfügbaren Datenbank-Treiber. Ist das ‚Ergebnis‘ gleich 0, dann sind keine weiteren Treiber verfügbar. Dieser Befehl muss nach *ExamineDatabaseDriver()* aufgerufen werden. Um die Treiber-Informationen auszuwerten, können *DatabaseDriverName()* und *DatabaseDriverDescription()* benutzt werden.

NextDatabaseRow()

Ergebnis = *NextDatabaseRow()*

Ermittelt die Informationen über den nächsten Datenbank-Eintrag (Zeile). Ist das ‚Ergebnis‘ gleich 0, dann sind keine weiteren Einträge mehr verfügbar (Ende der Tabelle). Um auf die Informationen innerhalb des Eintrags (bzw. der Zeile) zuzugreifen, benutzen Sie *GetDatabaseLong()*, *GetDatabaseFloat()* und *GetDatabaseString()*.

OpenDatabase(#Database, ODBCDatabaseName\$, User\$, Password\$)

Ergebnis = *OpenDatabase(#Database, ODBCDatabaseName\$, User\$, Password\$)*

Öffnet die angegebene ODBC Datenbank 'ODBCDatabaseName\$' und legt diese als aktuelle Datenbank fest. Alle weiteren Operationen beziehen sich jetzt auf diese neu geöffnete Datenbank. Zum Ändern der aktuellen Datenbank benutzen Sie einfach den *UseDatabase()* Befehl. Ist das ‚Ergebnis‘ gleich 0, wurde die Datenbank nicht gefunden oder Benutzer/Passwort (User\$/Passwort\$) sind falsch. Der Passwort\$ kann ein Null-String sein, wenn kein Passwort benötigt wird. Zum Deklarieren einer ODBC-Datenbank lesen Sie das ODBC-Hilfedokument von Windows. Wenn zuvor eine andere Datenbank mit derselben #Database Nummer geöffnet war, wird diese automatisch geschlossen.

OpenDatabaseRequester(#Database)

Ergebnis = *OpenDatabaseRequester(#Database)*

Öffnet den Standard Windows ODBC-Requester, um eine Datenbank zum Öffnen auszuwählen. Ist das ‚Ergebnis‘ gleich 0, konnte die Datenbank nicht geöffnet werden, andernfalls wird sie zur aktuell geöffneten Datenbank. Alle weiteren Operationen beziehen sich jetzt auf diese neu

geöffnete Datenbank. Zum Ändern der aktuellen Datenbank benutzen Sie einfach den *UseDatabase()* Befehl.

PreviousDatabaseRow()

Ergebnis = *PreviousDatabaseRow()*

Ermittelt die Informationen über den vorherigen Datenbank-Eintrag (Zeile). Ist das ‚Ergebnis‘ gleich 0, dann sind keine weiteren Einträge mehr verfügbar (Anfang der Tabelle). Um auf die Informationen innerhalb des Eintrags (bzw. der Zeile) zuzugreifen, benutzen Sie *GetDatabaseLong()*, *GetDatabaseFloat()* und *GetDatabaseString()*.

UseDatabase(#Database)

Ergebnis = *UseDatabase(#Database)*

Ändert die aktuelle Datenbank auf die neu angegebene. Ist das ‚Ergebnis‘ gleich 0, wurde die angegebene Datenbank nicht initialisiert.

16.35 Packer

Die Packer Library bietet eine Reihe effiziente Befehle zum Packen und Entpacken von Daten ("Komprimieren/Dekomprimieren"). Die Pack-/Entpack-Routinen basieren auf dem excellenten JCalG1-Algorithmus und sind in PureBasic integriert (keine externen DLLs werden benötigt).

AddPackFile(DateiName\$)

Ergebnis = *AddPackFile(DateiName\$)*

Komprimiert die angegebene Datei und fügt diese zur aktuellen Pack-Datei, die zuvor mit *CreatePack()* geöffnet wurde, hinzu. Das Hinzufügen einer großen Datei kann eine längere Zeit dauern, aber die Entpackgeschwindigkeit ist dafür sehr sehr schnell (nahezu sofort) und die Packrate ist sehr gut.

AddPackMemory(Speicheradresse, Länge)

Ergebnis = *AddPackMemory(Speicheradresse, Länge)*

Komprimiert den angegebenen Speicherbereich und fügt diesen zur

aktuellen Pack-Datei, die zuvor mit *CreatePack()* geöffnet wurde, hinzu. Das Hinzufügen eines großen Speicherbereichs kann eine längere Zeit dauern, aber die Entpackgeschwindigkeit ist dafür sehr sehr schnell (nahezu sofort) und die Packrate ist sehr gut.

ClosePack()

Schließt eine zuvor mittels *OpenPack()* oder *CreatePack()* geöffnete Pack-Datei. Hinweis: PureBasic schließt am Programmende automatisch alle noch offenen Pack-Dateien.

CreatePack(DateiName\$)

Ergebnis = *CreatePack(DateiName\$)*

Erstellt eine neue leere Pack-Datei. Ist das ‚Ergebnis‘ gleich 0, konnte die Pack-Datei nicht erstellt werden.

Die folgenden Befehle können benutzt werden, um Daten zu einer Pack-Datei hinzuzufügen:

AddPackFile() : Hinzufügen und Komprimieren einer ganzen Datei.

AddPackMemory() : Hinzufügen und Komprimieren eines Speicherbereichs.

Sobald alle Schreiboperationen abgeschlossen wurden, sollte *ClosePack()* aufgerufen werden.

Hinweis

PureBasic schliesst am Programmende automatisch alle noch offenen Pack-Dateien.

NextPackFile()

Speicheradresse = *NextPackFile()*

Liest und entpackt die nächste gefundene Datei in der aktuellen Pack-Datei, die zuvor mittels *OpenPack()* geöffnet wurde. Die zurückgegebene Speicheradresse wird geteilt ('shared') und sollte nie vom Programmierer freigegeben werden. Um die Größe des Speichers (d.h. die Größe der entpackten Datei) zu ermitteln, benutzen Sie *PackFileSize()*.

OpenPack(DateiName\$)

Ergebnis = *OpenPack(DateiName\$)*

Öffnet eine zuvor mittels *CreatePack()* erstellte Pack-Datei. Ist das ‚Ergebnis‘ gleich 0, konnte die Pack-Datei nicht geöffnet werden. *NextPackFile()* wird benutzt, um Daten aus der Pack-Datei zu erhalten und zu entpacken.

Sobald alle Leseoperationen abgeschlossen wurden, sollte *ClosePack()* aufgerufen werden. Hinweis: PureBasic schliesst am Programmende automatisch alle noch offenen Pack-Dateien.

PackerCallback(@Procedure())

Fügt einen "Callback" beim Komprimieren von Daten hinzu. Da das Komprimieren sehr langsam abläuft, ist damit das regelmäßige Aufrufen einer Prozedur zum Verfolgen des Pack-Prozesses möglich. Die Zeit zwischen zwei Aufrufen ist nicht durch den Benutzer definierbar, sie wird automatisch kalkuliert. Diese Funktion ist nur für fortgeschrittene Programmierer.

Der Callback muss folgende Form haben:

```
Procedure CompressCallback(SourcePosition, DestinationPosition)
    ...
EndProcedure
```

SourcePosition: Aktuelle Position, in Byte, im Ausgangsbuffer (unkomprimiert)

DestinationPosition: Aktuelle Position, in Byte, im Zielbuffer (komprimiert)

Um das Komprimieren fortzusetzen, muss die Prozedur 1 zurückgeben. Um sie abubrechen, geben Sie einfach 0 zurück.

PackFileSize()

Größe = *PackFileSize()*

Ermittelt die Größe der aktuell (nach einem *NextPackFile()*) zu entpackenden Datei.

PackMemory(SourceMemoryID, DestinationMemoryID, SourceLength)

Ergebnis = PackMemory(SourceMemoryID, DestinationMemoryID, SourceLength)

Packt (komprimiert) den Inhalt des Ausgangsspeicherbereichs 'SourceMemoryID' mit der Länge 'SourceLength' in den Zielspeicherbereich 'DestinationMemoryID'. Der Zielspeicherbereich muss mindestens die Länge des Ausgangsspeicherbereichs + 8 haben. Ergibt 'Ergebnis' gleich 0, ist das Packen fehlgeschlagen, andernfalls enthält es die komprimierte Größe des Zielbereichs. Die komprimierten Daten können mittels *UnpackMemory()* wieder entpackt werden. Für fortgeschrittene Benutzer, kann ein Callback zum Verfolgen des Pack-Prozesses mittels *PackerCallback()* hinzugefügt werden. Speicherbereiche können einfach mittels dem *AllocateMemory()*Befehl reserviert werden.

Hinweis

Der Kompressions-Algorithmus ist derzeit sehr langsam, ergibt aber sehr gute Resultate (besser als das .ZIP Format). Und das Beste daran: das Entpacken geht wahnsinnig schnell (viel schneller als .ZIP).

UnpackMemory(SourceMemoryID, DestinationMemoryID)

Ergebnis = UnpackMemory(SourceMemoryID, DestinationMemoryID)

Entpackt einen zuvor (mit *PackMemory()*) komprimierten Speicherbereich 'SourceMemoryID' in den Zielspeicherbereich 'DestinationMemoryID'.

Hinweis

Das Dekomprimieren ist sehr sehr schnell, anders als das Komprimieren.

16.36 Cipher

Die 'Cipher' (Verschlüsselung) Library ist ein Set von Funktionen, die nützlich für das Chiffrieren und Verschlüsseln von Daten sind. Zum Beispiel ist die MD5 (Message Digest 5) eine sehr populäre "Fingerabdruck" Routine, welche wegen ihrem hohen Schutz gegen Cracker auf vielen Gebieten zum Einsatz kommt: Linux, BSD und Unix benutzen sie als Alternative zum Abspeichern von Passwörtern auf Disk.

Base64Encoder(EingabeBuffer, EingabeLänge, AusgabeBuffer, AusgabeLänge)

Verschlüsselt den angegebenen Speicherbuffer unter Verwendung des

Base64 Algorithmus. Dieser wird hauptsächlich in Email-Programmen benutzt, kann aber auch nützlich für alle anderen Programme sein, die das Verschlüsseln von rohen Binär-Dateien zu Dateien im ASCII-Format (7 Bit, Zeichen nur von 32 bis 127 der ASCII-Tabelle) benötigen. Der 'AusgabeBuffer' sollte 33% größer sein als der 'EingabeBuffer', mit einer minimalen Größe von 64 Bytes.

CRC32Fingerprint(Buffer, Länge)

Ergebnis = CRC32Fingerprint(Buffer, Länge)

Führt auf dem angegebenen Speicherbuffer eine CRC32 Checksumme aus. CRC32 ist ein 32-Bit "Fingerabdruck", der - da leicht zu knacken - nicht für Passwort-Sicherung gedacht ist, sondern für die schnelle Prüfung der Datenintegrität. Zum Beispiel haben ZIP-Dateien am Ende jeder Datei eine CRC32 Checksumme, um sicher zu stellen, dass das Archiv nicht korrupt ist. Der Hauptvorteil von CRC32 gegenüber MD5 oder anderen Hash-Algorithmen ist seine sehr hohe Geschwindigkeit.

DESFingerprint(Passwort\$, Schlüssel\$)

Ergebnis\$ = DESFingerprint(Passwort\$, Schlüssel\$)

Dieser Algorithmus basiert auf der DES (Data Encryption Standard) Verschlüsselungsmethode, welche einen String mit 13 Zeichen zurückliefert. Dieser String ist nahezu einmalig und nicht umkehrbar. Wegen seiner starken Chiffrierung ist er kaum zu knacken, wenn das Passwort lang genug (> 10 Zeichen) ist. Der Passwort\$ sollte aber aus Performancegründen auch nicht ein sehr langer String sein. Um einen größeren Speicherbuffer (> 16 KB) zu verschlüsseln, verwenden Sie besser den *MD5Fingerprint()* Befehl. Der 'Schlüssel\$' ist auch als "Salt" bekannt, insbesondere Linux/Unix/BSD Anwendern. Wenn ein 2 Zeichen langer 'Schlüssel\$' benutzt wird, gibt die Funktion einen 'Salt2' String zurück, welcher mit jedem Standard Linux "Hash" Passwort (etc/passwd) kompatibel ist. Dieser Befehl basiert auf dem OpenSource crypt() Befehl.

MD5Fingerprint(Buffer, Länge)

Ergebnis\$ = MD5Fingerprint(Buffer, Länge)

Gibt einen 15 Zeichen langen MD5 (Message Digest 5) Hash-Code

zurück. Hier eine kurze Erläuterung von RFC 1321 auf MD5:

'Der Algorithmus verwendet als Eingabe eine Mitteilung beliebiger Länge und produziert als Ausgabe einen 128-Bit "Fingerabdruck" oder "Message Übersicht" der Eingabe. Es wird allgemein angenommen, dass es unmöglich ist: zwei Mitteilungen zu produzieren, die in derselben "Message Übersicht" resultieren oder eine Mitteilung zu produzieren, wo vorher die daraus resultierende "Message Übersicht" bekannt ist. Der MD5 Algorithmus ist geeignet für Applikationen mit digitaler Unterschrift.'

MD5 Hash-Codes werden häufig für Passwort-Verschlüsselung benutzt, da sie eine sehr gute standardmäßige Sicherheit bieten. Weitere Infos finden Sie im RFC 1321: <http://www.ietf.org/rfc/rfc1321.txt>

16.37 SysTray

"SysTray" ist der rechte Abschnitt der MS Windows Taskleiste, welcher einige Icons und Datum/Uhrzeit enthält. PureBasic bietet vollen Zugriff auf diesen Bereich und erlaubt das Hinzufügen einer beliebigen Anzahl von Bildern.

AddSysTrayIcon(#SysTrayIcon, WindowID, ImageID)

Fügt ein Icon in den "SysTray" Bereich hinzu. 'WindowID' muss eine gültige *WindowID()* sein. 'ImageID' muss ein zuvor (mittels *LoadImage()*) geladenes Bild sein; bei diesem Befehl werden jedoch nur ICON (.ico) Bilder unterstützt. *UseImage()* kann benutzt werden, um einfach diese ID zu bekommen.

Wenn ein Ereignis auf irgendeinem der SysTray Icons auftritt, wird ein #PB_EventSysTray Ereignis zurückgegeben. *EventGadgetID()* kann benutzt werden, um das benutzte SysTrayIcon zu ermitteln. Der *EventType()* Befehl wird ebenfalls durch diesen Befehl aktualisiert.

Hinweis: Alle SysTray Icons werden automatisch am Programmende entfernt.

ChangeSysTrayIcon(#SysTrayIcon, ImageID)

Ändert das angegebene Icon im SysTray Bereich. 'ImageID' muss ein zuvor (mittels *LoadImage()*) geladenes Bild sein, es werden aber nur Bilder vom ICON Typ (.ico) bei diesem Befehl unterstützt. *UseImage()* kann benutzt werden, um einfach diese ID zu ermitteln.

RemoveSysTrayIcon(#SysTrayIcon)

Entfernt das angegebene #SysTrayIcon.

SysTrayIconTooltip(#SysTrayIcon, Text\$)

Assoziiert den angegebenen 'Text\$' mit dem SysTray Icon. Ein ToolTip-Text ist ein Text, welcher (in einer kleinen gelben Box) angezeigt wird, wenn der Mauspfel eine Weile über dem Icon verharrt.

16.38 Thread

Ein Thread ist ein Programmteil, welcher asynchron, d.h. im Hintergrund des Hauptprogramms, läuft. Dies bedeutet: es ist möglich, langwierige Operationen (Packen, Bildverarbeitung...) durchzuführen, ohne das gesamte Programm anzuhalten und damit dem Benutzer das Weiterarbeiten zu ermöglichen. Wenn das Hauptprogramm beendet wird, werden alle Threads zerstört. Unter PureBasic sind Threads einfach eine Prozedur, welche asynchron aufgerufen wird. Der Thread läuft, bis die Prozedur endet.

CreateThread(@ProcedureName(), Wert)

ThreadID = *CreateThread(@ProcedureName(), Wert)*

Erstellt einen neuen Thread, welcher im Hintergrund der Applikation läuft. Wenn der Thread korrekt erstellt wurde, wird die 'ThreadID' zurückgegeben, welche zusammen mit den anderen Befehlen, wie *KillThread()*, *PauseThread()*, etc..., benutzt wird. Die Prozedur muss im nachfolgenden Format erstellt werden:

```
Procedure DeineProzedur(Wert)
...
EndProcedure
```

Das Argument 'Wert' bei CreateThread() wird an den 'Wert' Parameter der Prozedur übergeben.

KillThread(ThreadID)

Zerstört den angegebenen Thread, der zuvor mit *CreateThread()* erstellt wurde. Dies ist eine sehr gefährliche Operation, benutzen Sie sie sofern möglich nicht.

PauseThread(ThreadID)

Hält den angegebenen Thread an, der zuvor mit *CreateThread()* erstellt wurde. Der Thread kann mittels *ResumeThread()* fortgesetzt werden.

ResumeThread(ThreadID)

Setzt den angegebenen Thread fort, welcher zuvor mittels *PauseThread()* angehalten wurde.

ThreadPriority(ThreadID, Priorität)

Ändert die 'Priorität' des angegebenen Threads 'ThreadID' und gibt die alte Priorität zurück. Der Wert von 'Priorität' kann zwischen 1 und 32 liegen. 1 ist die niedrigste Priorität, 16 ist die normale Priorität und 32 ist die zeitkritische Priorität (die höchste Priorität; bitte benutzen Sie diese nicht, außer Sie wissen was Sie tun). Wird als neue Priorität 0 übergeben, dann wird die Priorität nicht verändert (nützlich um nur die Priorität des Threads zu ermitteln, aber nicht zu verändern).

16.39 Library

Libraries (Befehls-Bibliotheken) sind "shared" (zur Benutzung mit verschiedenen Applikationen freigegebene) OS-Komponenten, die dem Programmierer spezielle Funktionen anbieten. Zum Beispiel kann eine Library Befehle zum einfachen Handhaben und Manipulieren von Bildern enthalten. Jedes OS bietet eine Anzahl an "shared" Libraries, um das Programmierer-Leben zu vereinfachen. Mit PureBasic ist es möglich, diese Libraries einfach und dynamisch zu benutzen!

Unter Windows sind diese Libraries gut bekannt unter dem Namen 'DLL'.

CallFunction(#Library, FunctionName\$ [,Parameter1 [, Parameter2...]])

Ergebnis = *CallFunction(#Library, FunctionName\$ [,Parameter1 [, Parameter2...]])*

Ruft eine Funktion in der angegebenen #Library, die zuvor mittels *OpenLibrary()* geöffnet wurde, anhand ihres Namens 'FunctionName\$' auf. Eine beliebige Anzahl an Parametern kann angegeben werden, muss aber mit der von der Funktion benötigten Anzahl übereinstimmen. Zum Beispiel, wenn eine Funktion 2 Parameter benötigt, müssen 2 Parameter angegeben werden, auch wenn diese 2 Werte gleich null sind. 'Ergebnis' enthält das Ergebnis der aufgerufenen Funktion.

CallFunctionFast(*FunctionPointer [,Parameter1 [, Parameter2...]])

Ergebnis = *CallFunctionFast(*FunctionPointer [,Parameter1 [, Parameter2...]])*

Ruft eine Library-Funktion mittels ihres Zeigers '*FunctionPointer', der zuvor mit *IsFunction()* ermittelt wurde, auf. Dies ist die schnellste Möglichkeit hierfür. Eine beliebige Anzahl an Parametern kann angegeben werden, muss aber mit der von der Funktion benötigten Anzahl übereinstimmen. Zum Beispiel, wenn eine Funktion 2 Parameter benötigt, müssen 2 Parameter angegeben werden, auch wenn diese 2 Werte gleich null sind. 'Ergebnis' enthält das Ergebnis der aufgerufenen Funktion.

CloseLibrary(#Library)

Schließt die angegebene #Library und gibt den zuvor reservierten Speicher frei.

IsFunction(#Library, FunctionName\$)

Ergebnis = *IsFunction(#Library, FunctionName\$)*

Überprüft, ob die angegebene '#Library', welche zuvor mit *OpenLibrary()* geöffnet wurde, die angegebene Funktion 'FunctionName\$' enthält. Achtung, diese Funktion arbeitet immer "case-sensitive" (achten Sie auf Groß- und Kleinschreibung). Ist das 'Ergebnis' gleich 0, dann wurde die Funktion nicht gefunden. Andernfalls wird ein Zeiger (Pointer) auf die Funktion zurückgegeben, nützlich in Verbindung mit *CallFunctionFast()*.

Ergebnis = *OpenLibrary(#Library, Dateiname\$)*

Öffnet die angegebene Library. Ist das 'Ergebnis' gleich 0, dann konnte die Library nicht geöffnet werden (die Library wurde nicht gefunden).

oder ist korrupt).

17. Fehlermeldungen

Kein Anwender kommt beim Programmieren ohne Fehler aus, sei es durch Tippfehler oder durch einen falsch angewendeten Befehls-Syntax. Der PureBasic-Compiler überprüft daher bereits vor dem Kompilieren Ihren Programmcode auf offensichtliche Fehler und quittiert diese mit einer entsprechenden Fehlermeldung.

Damit Sie deren Bedeutung verstehen können, nachfolgend eine alphabetisch sortierte Liste mit allen möglichen (englischen) Fehlermeldungen und deren Bedeutung:

<i>Fehlermeldung</i>	<i>Bedeutung und ggf. mögliche Ursachen</i>	<i>Zugehöriges Kapitel</i>
Syntax Error !	Syntax Fehler! Kommt immer dann vor, wenn Sie eine Funktion falsch anwenden. Schauen Sie dann in die Erklärung bzw. Befehlsreferenz.	
Too many '(' parenthesis	Zu viele öffnende '(' Klammern. Auf einer Programmzeile befinden sich mehr öffnende als schließende Klammern.	
Too many ')' parenthesis	Zu viele schließende ')' Klammern. Auf einer Programmzeile befinden sich mehr schließende als öffnende Klammern.	
Garbage to the end of line	„Schrott“ am Zeilenende. Kann immer dann vorkommen, wenn Sie Parameter	

	den, einen Variablen-Typ angeben wo keiner erforderlich ist usw.	
Overflow error: a 'byte' value (.b) must be between -128 and +255	Überlauf-Fehler: ein 'Byte' Wert (.b) muss zwischen -128 und +255 liegen	7.2
)' expected	')' Klammer wird erwartet.	
Trying to write a numeric value into string variable	Versuch einen numerischen Wert in eine String-Variable zu speichern.	7.2
Can't mix strings with numerical values	Strings können nicht mit numerischen Werten vermischt werden.	7.2
Trying to write a string into numeric variable	Versuch einen String in eine numerische Variable zu schreiben	7.2
The specified variable type is different than before	Der angegebene Variablentyp ist anders als bisher. Kommt vor, wenn Sie z.B. Variable ,a' bereits mit ,a.l' als Long-Variable definiert haben und später im Code versuchen ,a.w' zu verwenden.	7.3
The specified number is incorrect for '<<' or '>>' operator (must be 1-31)		

array, or a linked list	gegebene Schlüsselwort ist PureBasic nicht bekannt.	
Syntax error.	Syntax-Fehler. Die wohl häufigste Fehlermeldung, wenn z.B. die von Ihnen angegebenen Parameter einer Funktion nicht korrekt sind.	
Bad parameter type: a string is expected	Der angegebene Parameter ist vom falschen Typ: es wird ein String erwartet.	7.2
Bad parameter type: an array is expected	Der angegebene Parameter ist vom falschen Typ: es wird ein Array erwartet.	7.6
Bad parameter type: a linked list is expected	Der angegebene Parameter ist vom falschen Typ: es wird eine verknüpfte Liste („Linked List“) erwartet.	7.7
Strange error: object type not detected		
Another 'End Condition' operand is expected here	Ein weiteres Schlüsselwort zum Abschließen des Bedingungsblocks (z.B. EndIf) wird hier noch erwartet.	9.2
An array must have a name	Sie wollen ein Array definieren, haben jedoch keinen Namen dafür angegeben.	7.6
A constant numeric	Ein fester numerischer Wert	9.3

'Step'	Schlüsselwort erwartet.	
After 'For' you must have 'To'	Bei einer For : Next Schleife haben Sie das Schlüsselwort "To" vergessen.	9.3
Illegal expression to For/Next. Must be: 'For Var=Expression To'	Ungültiger Ausdruck bei diesem For/Next. Korrekt muss es heißen: „For Variable=Ausdruck To“	9.3
Bad variable for this 'Next'	Falsche Variable bei diesem 'Next'.	9.3
This array doesn't have a Structure.	Dieses Array hat keine Struktur.	7.6
Only a litteral string can be put after 'IncludeBinary'	Nach einem 'IncludeBinary' kann nur ein Buchstaben-String angegeben werden.	15.1
Only a litteral string can be put after 'IncludePath'	Nach einem 'IncludePath' kann nur ein Buchstaben-String angegeben werden.	15.1
Only a litteral string can be put after 'IncludeFile'	Nach einem 'IncludeFile' kann nur ein Buchstaben-String angegeben werden.	15.1
Only a litteral string can be put after 'XIncludeFile'	Nach einem 'XIncludeFile' kann nur ein Buchstaben-String angegeben werden.	15.1
XIncluded file not found (Die per 'XInclude...' einzufügen	15.1

gefunden.

A type or structure must be specified after 'DefType'	Nach 'DefType' muss ein Typ oder eine Struktur angegeben werden.	7.3
Included file not found:	Einzufügende Datei wurde nicht gefunden.	
Unfinished condition	Unvollständig eingegebene Bedingung.	9.2
Assembly error ! Please mail us this file.	Assembler-Fehler! Bitte mailen Sie uns diese Datei.	
Illegal operator for 'DefType'	Ungültiger Operator bei 'DefType'.	7.3
You can't define a procedure inside another procedure	Sie können innerhalb einer Prozedur keine weitere Prozedur definieren.	10.2
Bad parameter type, number expected instead of string	Falscher Parameter-Typ, anstelle eines Strings wird eine Zahl erwartet.	7.2
'EndProcedure' without 'Procedure'	'EndProcedure' ohne vorheriges 'Procedure'.	10.2
'Global' only accept 'Variables'	'Global' akzeptiert nur 'Variablen'.	10.3
'Shared' only accept 'Variables'	'Shared' akzeptiert nur 'Variablen'.	10.3

'Shared' can be used only inside a procedure	'Shared' kann nur innerhalb einer Prozedur benutzt werden.	10.3
'Else' without 'If'	'Else' ohne vorheriges 'If'.	9.2
'Wend' without 'While'	'Wend' ohne vorheriges 'While'.	9.3
'Until' without 'Repeat'	'Until' ohne vorheriges 'Repeat'.	9.3
'EndIf' without 'If'	'EndIf' ohne vorheriges 'If'.	9.2
'EndSelect' without 'Select'	'EndSelect' ohne vorheriges 'Select'.	9.2
'Forever' without 'Repeat'	'Forever' ohne vorheriges 'Repeat'.	9.3
'Next' without 'For'	'Next' ohne vorheriges 'For'.	9.3
'(' is missing after the function call	Die Klammer '(' nach einem Funktionsaufruf fehlt.	
() : Incorrect number of parameters.	Falsche Anzahl an angegebenen Parametern.	
Too few parameters for	Zu wenige Parameter für ...	
' offset not found in this structure	Offset in dieser Struktur nicht gefunden.	7.8
—	Zu viele Parameter für ...	

for

Too few argument for the procedure	Zu wenige Argumente beim Prozeduraufruf angegeben.	10.2
Numbers between " are limited to 4 characters	Zahlen zwischen ‘’ sind auf vier Ziffern beschränkt.	
Too many arguments for the procedure	Zu viele Argumente beim Prozeduraufruf angegeben.	10.2
Not a valid Decimal number	Keine gültige Dezimalzahl.	7.2
Not a valid Hexadecimal number	Keine gültige Hexadezimalzahl.	7.2
Not a valid Binary number	Keine gültige Binärzahl.	7.2
Numeric overflow: too big number	Zahlen-Überlauf: die Zahl ist zu groß.	7.2
Constant not found: #	Die angegebene Konstante wurde nicht gefunden.	7.4
Array type is different than before !	Der angegebene Array-Typ ist anders als vorher schon einmal angegeben!	7.6
A constant can't be composed by a variable or a function	Eine Konstante kann sich nicht aus einer Variable oder einer Funktion ergeben.	7.4

Constant has already been declared: #	Die angegebene Konstante wurde bereits deklariert.	7.4
A constant must have a name !	Für die Konstante muss eine Name angegeben werden!	7.4
'EndStructure' without 'Structure'	'EndStructure' ohne vorheriges 'Structure'.	7.8
A variable in a 'Structure' need a declared Type	Bei einer Variable innerhalb einer Struktur muss der Typ mit deklariert werden.	7.8
A 'Structure' must have a Name !	Eine 'Structure' muss einen Namen haben!	7.8
This structure is already declared.	Diese Struktur wurde vorher bereits deklariert.	7.8
Can't have 'Structure' in Structure	'Structure' kann nicht innerhalb einer Struktur verwendet werden.	7.8
Can't do it, else endless recursivity	Diese Funktion kann nicht ausgeführt werden, da dies sonst zu endloser Rekursivität führt.	10.4
Structure not found:	Die angegebene Struktur wurde nicht gefunden.	7.8
'EndStructure' is missing	'EndStructure' fehlt.	7.8
'EndProcedure' is missing	'EndProcedure' fehlt.	10.2

ing

This variable doesn't have a 'Structure'	Die angegebene Variable hat keine 'Structure'.	7.8
--	--	-----

This list doesn't have a 'Structure'	Die angegebene Liste hat keine 'Structure'.	7.8
--------------------------------------	---	-----

'NewList' syntax is: NewList listname()	Der Syntax von 'NewList' lautet: NewList Liste()	7.7
---	--	-----

'Dim' syntax is: Dim arrayname(nb)	Der Syntax von 'Dim' lautet: Dim Array(Anzahl)	7.6
------------------------------------	--	-----

must be called in your program	... muss in Ihrem Programm aufgerufen werden.	
--------------------------------	---	--

' is not a valid operator	... ist kein gültiger Operator.	
---------------------------	---------------------------------	--

Source too big for demo version	Der Sourcecode ist zu groß für die Demoversion.	
---------------------------------	---	--

A Procedure must have a name	Eine Prozedur muss einen Namen haben.	10.2
------------------------------	---------------------------------------	------

A Procedure must begin with a '('	Eine Prozedur muss mit einer '(' beginnen.	10.2
-----------------------------------	--	------

Procedure is already declared	Diese Prozedur wurde bereits deklariert.	10.2
-------------------------------	--	------

A ')' is expected to close the Procedure	Eine ')' wird erwartet, um die Prozedur zu schließen.	10.2
--	---	------

Syntax error in the Procedure arguments	Syntax-Fehler bei den Prozedur-Argumenten.	10.2
A structure can't be used for procedure return	Eine Struktur kann nicht als Rückgabewert einer Prozedur benutzt werden.	10.2
Not a valid Float number	Keine gültige Fließkommazahl.	7.2
Can't load any PureLibraries ! Please re-install PureBasic	Konnte keine PureLibraries laden! Bitte installieren Sie PureBasic erneut.	
Can't load any OSLibraries ! Please re-install PureBasic	Konnte keine OSLibraries laden! Bitte installieren Sie PureBasic erneut.	
'()' expected after a Linked List	'()' werden nach einer verknüpften Liste erwartet.	7.7
File not found (Die angegebene Datei wurde nicht gefunden.	
A Structure name is expected inside SizeOf()	Der Name einer Struktur wird innerhalb von 'SizeOf()' erwartet.	7.8
Can't create the 'PureBasic.exe' file (already running ?)	Konnte kein ausführbares Programm (PureBasic.exe) erzeugen. (Läuft dies bereits?)	5.2
'ProcedureReturn' expected	'ProcedureReturn' erwartet	10.2

meric value.

schen Wert.

Duplicated
command (

library

Der angegebene Library-Befehl ist doppelt vorhanden. Kann passieren, wenn Sie eine neue Version einer Befehlsbibliothek in ‚Pure-Libraries‘ kopiert haben ohne die alte Version zu löschen.

Can't load the following
library:

Die angegebene Befehlsbibliothek konnte nicht geladen werden.

Invalid name: same as
an external command.

Ungültiger Name: entspricht dem Namen eines externen Befehls.

Invalid name: same as
an array.

Ungültiger Name: entspricht dem Namen eines Arrays.

Invalid name: same as a
linked list.

Ungültiger Name: entspricht dem Namen einer verknüpften Liste.

Invalid name: same as a
procedure.

Ungültiger Name: entspricht dem Namen einer Prozedur.

Label not found (

Die angegebene Sprungmar- 9.4
ke wurde nicht gefunden.

Overflow error: a 'word'
value (.w) must be
between -32768 and
+65535

Überlauf-Fehler: eine 7.2
‘Word’-Variable (.w) muss
zwischen -32768 und +65535
liegen.

Structure offset is missing	Der Struktur-Offset fehlt.	7.8
Icon file not found or of incorrect format	Icon-Datei nicht gefunden bzw. im falschen Format.	
Gosub not allowed inside a Procedure	‘Gosub’ ist innerhalb von Prozeduren nicht erlaubt.	9.4
Divide by 0 forbidden	Division durch Null ist verboten.	
Duplicate label.	Doppelte Sprungmarke.	9.4
Litteral string not terminated (\ " missing).	Buchstaben-String wurde nicht abgeschlossen (\ " fehlt).	
'Case' without 'Select'	‘Case’ ohne ‘Select’.	9.2
The following PureLibrary is missing:	Die angegebene PureLibrary fehlt. (Schauen Sie im PureLibraries Verzeichnis nach.)	
Too complex expression (out of CPU registers). Please split it.	Zu komplexer Ausdruck (CPU-Register ausgelastet). Bitte splitten Sie den Ausdruck.	
Can't use the following operands with floats: <<, >>, &,	Die Operanden <<, >>, &, können nicht mit Fließkommazahlen benutzt werden.	8.5
A Procedure can't be	Eine Prozedur kann nicht	10.2

Repeat, While or For.	Bedingung bzw. innerhalb von ‚Repeat‘, ‚While‘ oder ‚For‘-Schleifen deklariert werden.	
Can't load any resident !	Es konnten keine Resident-Dateien geladen werden!	
A type or structure must be specified after 'Data'	Ein Typ oder eine Struktur muss nach ‚Data‘ angegeben werden.	7.5
'Data' can be declared only in the 'DataSection'	‚Data‘ kann nur innerhalb der ‚DataSection‘ deklariert werden.	7.5

18. Index

o

- 59, 60

!

! 67, 69, 140

#

#PB_Button_Default.....	236
#PB_Button_Left.....	236
#PB_Button_MultiLine	236
#PB_Button_Right.....	236
#PB_Button_Toggle.....	236
#PB_CheckBox_Center :	237
#PB_CheckBox_Right	237
#PB_ClipboardImage	265
#PB_ComboBox_Editable	239
#PB_ComboBox_LowerCase	239
#PB_ComboBox_UpperCase.....	239
#PB_EventCloseWindow :	232
#PB_EventGadget.....	232
#PB_EventMenu	232
#PB_EventMoveWindow	232
#PB_EventRepaint	232
#PB_EventType_Change.....	228
#PB_EventType_Focus.....	228
#PB_EventType_LeftClick	227
#PB_EventType_LeftDoubleClick	228
#PB_EventType_LostFocus.....	228
#PB_EventType_ReturnKey.....	228

#PB_EventType_RightClick.....	228
#PB_EventType_RightDoubleClick.....	228
#PB_FileSystem_Archive	177
#PB_FileSystem_Compressed	177
#PB_FileSystem_Hidden	177
#PB_FileSystem_Normal	177
#PB_FileSystem_ReadOnly	177
#PB_FileSystem_System.....	177
#PB_Font_Bold	261
#PB_Font_Italic.....	261
#PB_MessageRequester_Ok.....	112, 259
#PB_MessageRequester_YesNo	113, 258
#PB_MessageRequester_YesNoCancel	113
#PB_Sprite_Alpha.....	204
#PB_Sprite_Memory.....	204
#PB_Sprite_Texture	204
#PB_String_BorderLess	251
#PB_String_LowerCase	251
#PB_String_Numeric	250
#PB_String_Password.....	250
#PB_String_ReadOnly	250
#PB_String_UpperCase	251
#PB_Text_Border	251
#PB_Text_Center	251
#PB_Text_Right.....	251
#PB_ToolBarIcon_Copy	109
#PB_ToolBarIcon_Cut.....	108
#PB_ToolBarIcon_Delete.....	109
#PB_ToolBarIcon_Find	108
#PB_ToolBarIcon_Help	109
#PB_ToolBarIcon_New	108
#PB_ToolBarIcon_Open	108
#PB_ToolBarIcon_Paste	109
#PB_ToolBarIcon_Print.....	108

#PB_ToolBarIcon_Properties.....	109
#PB_ToolBarIcon_Redo	109
#PB_ToolBarIcon_Replace.....	108
#PB_ToolBarIcon_Save.....	108
#PB_ToolBarIcon_Undo.....	109
#PB_TrackBar_Ticks	252
#PB_TrackBar_Vertical.....	252
#PB_Tree_AlwaysShowSelection	253
#PB_Tree_CheckBoxes.....	253
#PB_Tree_NoButtons.....	253
#PB_Tree_NoLines.....	253
#PB_Web_Back.....	254
#PB_Web_Forward.....	254
#PB_Web_Refresh.....	254
#PB_Web_Stop	254
#PB_Window_BorderLess.....	230
#PB_Window_Invisible	230
#PB_Window_MaximizeGadget	229
#PB_Window_MinimizeGadget	229
#PB_Window_SizeGadget	230
#PB_Window_SystemMenu.....	229
#PB_Window_TitleBar	230

&

& 67

(

() is not a function, an array, or a linked list..... 292

,

)' expected 291

*

* 59, 61

/

/ 59, 61

/? 24

/COMMENTED 24

/CONSOLE 25

/DEBUGGER 24

/DLL 25

/EXE "Dateiname" 24

/ICON "IconName" 24

/INLINEASM 25

/NT4 25

/QUIET 25

/REASM 25

/RESIDENT "Dateiname" 25

|

| 67, 68

~

~ 67, 70

+

+ 59, 60

<

< 62, 63

<< 67, 70

<= 62, 63

<> 62, 64

=

= 58, 62

>

> 62, 63

>=62, 64

>>67, 71

2

2D Drawing..... 154

2DDrawing 188

A

A constant numeric value is expected after 'Step' 293

Abfrage von Windowsereignissen..... 117

Ablaufsteuerung..... 74

Abs(Zahl.f) 159

ACos(Winkel.f)..... 159

ActivateGadget(#Gadget) 235

ActivateWindow() 225

AddElement(LinkedList()) 166

AddGadgetColumn(#Gadget, Position, Titel\$, Breite) 235

AddGadgetItem(#Gadget, Position, Text\$ [, ImageID])). 235

AddGadgetItem() 238

Addition.....59, 60

AddKeyboardShortcut(#Window, Shortcut, EventID) 225

AddPackFile() 281

AddPackFile(Dateiname\$) 280

AddPackMemory() 281

AddPackMemory(Speicheradresse, Länge) 280

AddStatusBarField(Breite)..... 263

AddSysTrayIcon(#SysTrayIcon, WindowID, ImageID) 285

Adressen von Prozeduren 56

Adressen von Sprungmarken	57
Adressen von Variablen.....	55
After 'For' you must have 'To'	293
Allgemeiner Programmaufbau	94
AllocateMemory(#Memory, Größe, Flags)	169
Alphanumerische Datentypen.....	42
An array must have a name	293
And.....	65
AND	67
Another 'End Condition' operand is expected here.....	293
API.....	38, 122
Arithmetische Operatoren.....	59
Array	38
Arrays	46
Asc(String\$).....	161
ASCII.....	38
ASin(Winkel.f).....	159
Asm	29
Assembler	139
ATan(Winkel.f)	160
ATOM	130
AttachProcess.....	150
AttachThread.....	150
Aufruf.....	86
Ausdruck.....	38
Ausrufezeichen	140
Ausschneiden	15, 21, 108

B

BackColour(Rot, Grün, Blau).....	188
Bad parameter type: a linked list is expected.....	292
Bad parameter type: a string is expected.....	292
Bad parameter type: an array is expected.....	292

Base64Encoder(EingabeBuffer, EingabeLänge, AusgabeBuffer, AusgabeLänge)	284
BASIC	34
Bedingungen	74
Befehl	152
Befehlsreferenz	152
Benutzen einer Struktur	51
Benutzung von Zeigern	54
Binär-Daten	143
Bit	38
Bitweise Operatoren	66
Bitweises AND	67
Bitweises Linksschieben (Shift)	67
Bitweises NOT	67, 70
Bitweises OR	67, 68
Bitweises Rechtsschieben (Shift)	67
Bitweises Shift	70, 71
Bitweises XOR	67, 69
Blue(Farbe)	180
BOOL	126
BOOLEAN	126
Boolesche Variablen	42
Box(x, y, Breite, Höhe [, Farbe])	188
Breite = ImageWidth()	187
Bug	39
ButtonGadget	98
ButtonGadget(#Gadget, x, y, Breite, Höhe, Text\$ [, Flags])	236
ButtonImageGadget	98
ButtonImageGadget(#Gadget, x, y, Breite, Höhe, ImageID)	236
Byte	41
BYTE	126

C

CALLBACK.....	130
CallDebugger	30
CallFunction(#Library, FunctionName\$ [,Parameter1 [, Parameter2...]])	288
CallFunctionFast(*FunctionPointer [,Parameter1 [, Parameter2...]])	288
Can't mix strings with numerical values	291
CatchImage(#Image, Speicheradresse).....	185
CatchJPEGSprite(#Sprite, SpeicherAdresse, JPEGLength, Modus)	204
CatchSound(#Sound, Speicheradresse)	193
CatchSprite(#Sprite, SpeicherAdresse, Modus).....	203
CDAudio.....	154, 196
CDAudioLength()	197
CDAudioName().....	197
CDAudioStatus().....	197
CDAudioTrackLength(TrackNummer)	198
CDAudioTracks()	198
CDAudioTrackSeconds()	198
ChangeAlphaIntensity(R, G, B)	204
ChangeCurrentElement(LinkedList(), *NewElement)	167
ChangeGamma(R, G, B, Flags)	205
ChangeListIconGadgetDisplay((#Gadget, Modus).....	237
ChangeSysTrayIcon(#SysTrayIcon, ImageID)	286
CHAR	127
CheckBoxGadget.....	98
CheckBoxGadget(#Gadget, x, y, Breite, Höhe, Text\$ [, Flags])	237
CheckBoxGadget()	242
Chr(ASCII-Wert)	161
Cipher.....	156, 283
Circle(x, y, Radius [, Farbe]).....	189
ClearClipboard()	264

ClearConsole()	157
ClearGadgetItemList	103
ClearGadgetItemList(#Gadget)	238
ClearGadgetItemList()	238
ClearList(LinkedList())	167
ClearScreen(R, G, B)	205
Clipboard	155, 264
ClipSprite(#Sprite, x, y, Breite, Höhe)	205
CloseConsole()	33, 56, 57, 157
CloseDatabase(#Database)	276
CloseFile(#Datei)	172
CloseFont(#Font)	234
CloseHelp()	275
CloseLibrary(#Library)	288
CloseNetworkConnection(ConnectionID)	271
CloseNetworkServer()	271
ClosePack()	281
ClosePanelGadget()	238
ClosePreferences()	266
CloseScreen()	205
CloseSubMenu	105
CloseSubMenu()	255
CloseTreeGadgetNode()	238
CloseWindow(#Window)	227
Code	40
COLORREF	129
ColorRequester	115
ColorRequester()	258
ComboBoxGadget	98
ComboBoxGadget(#Gadget, x, y, Breite, Höhe [, Flags])	238
ComboBoxGadget()	235, 238, 243
ComboBoxGadget()	242
ComboGadget()	239, 242
Common Control Library	122

CompareMemory(MemoryID1, MemoryID2, Länge).....	170
CompareMemoryString(*String1, *String2 [, Modus [, Länge]]).....	170
Compiler	24, 39
Compiler Optionen	16, 19
Compiler-Direktiven	144
Console	20, 153, 156, 157
ConsoleColor(Zeichenfarbe, Hintergrundfarbe).....	157
ConsoleCursor(Höhe)	158
ConsoleLocate(x, y)	158
ConsoleTitle(Titel\$).....	158
CONST	127
Cont.....	29
ConText.....	145, 147
CopyFile(Ausgangsdatei\$, Zielfile\$)	176
CopyImage(#Image1, #Image2)	185
CopyMemory(SourceMemoryID, DestinationMemoryID, Länge).....	170
CopyMemoryString(*String [, @*DestinationMemoryID])	170
Copyright.....	6
CopySprite(#Sprite1, #Sprite2, Modus)	205
Cos(Winkel.f).....	160
CountGadgetItems(#Gadget).....	239
CountList(LinkedList()).....	167
CRC32Fingerprint(Buffer, Länge).....	284
CreateDirectory(VerzeichnisName\$).....	176
CreateFile(#Datei, DateiName\$).....	172
CreateGadgetList.....	103
CreateGadgetList(WindowID)	239
CreateImage(#Image, Breite, Höhe).....	185
CreateMenu	104
CreateMenu(#Menu, WindowID))	255
CreateNetworkServer(Port).....	271

CreatePack(DateiName\$).....	281
CreatePalette(#Palette).....	184
CreatePopupMenu.....	106
CreatePopupMenu(#Menu).....	255
CreatePreferences.....	266
CreatePreferences(DateiName\$).....	266
CreateSprite3D(#Sprite3D, #Sprite).....	215
CreateSprite3D().....	203
CreateStatusBar(#StatusBar, WindowID).....	263
CreateThread(@ProcedureName(), Wert).....	286
CreateToolBar.....	108
CreateToolBar(#ToolBar, WindowID).....	261
CRITICAL_SECTION.....	130

D

Danksagungen	6
DATA	44
Database.....	156, 275
DatabaseColumnName(Spalte).....	276
DatabaseColumns().....	276
DatabaseColumnType(Spalte).....	276
DatabaseDriverDescription().....	277
DatabaseDriverName().....	277
DatabaseQuery(Request\$).....	277
Datentypen.....	40
Debug	30
Debugger	16, 24, 27
Debuggers.....	27
DebugLevel.....	30
Declare'.....	91
Default.....	42
Definition	86
Definition einer Liste.....	49
Definition einer Struktur.....	50

Definition von Arrays.....	47
Deklaration.....	42, 91
Delay(Zeit)	179
DeleteElement(LinkedList()).....	167
DeleteFile(DateiName\$)	176
DESFingerprint(Passwort\$, Schlüssel\$)	284
DetachMenu()	227
DetachProcess	150
DetachThread	150
DirectoryEntryAttributes().....	176, 177
DirectoryEntryName()	177, 178
DirectoryEntrySize()	178
DirectX	195
Direkt NASM.....	141
Direktiven	144
DisableDebugger	30
DisableGadget(#Gadget, Status).....	240
DisableMenuItem	105
DisableMenuItem(MenuItem, Status)	256
DisableToolBarButton(#ButtonID, Status)	261
DisplayAlphaSprite(#Sprite, x, y).....	206
DisplayAlphaSprite()	203
DisplayPalette(#Palette)	184
DisplayPopupMenu	106
DisplayPopupMenu(#Menu, WindowID() [, x, y])	256
DisplayRGBFilter(x, y, Breite, Höhe, R, G, B)	206
DisplayShadowSprite(#Sprite, x, y).....	207
DisplayShadowSprite()	203
DisplaySolidSprite(#Sprite, x, y, R, G, B).....	207
DisplaySprite(#Sprite, x, y)	207
DisplaySprite3D(#Sprite3D, x, y, Transparenz).....	215
DisplayTranslucideSprite(#Sprite, x, y, Intensität)	208
DisplayTransparentSprite(#Sprite, x, y).....	208
Division.....	59, 61

DLL.....	39, 142, 149
DLLSample.pb	150
DrawImage(ImageID, x, y)	189
DrawingFont(FontID()).....	189
DrawingMode(Modus)	189
DrawText(Text\$)	190
Drucken	108, 111
DWORD	128
DWORD_PTR	128
DWORD32.....	128
DWORD64.....	131
Dynamic-Link Libraries	39

E

Editor	12
EditPadPro	146
Eigenschaften.....	109
Einfügen	15, 21, 109
Einfügen von Programmcode.....	142
Einleitung.....	6
EjectCDAudio(Status).....	198
Ellipse(x, y, RadiusX, RadiusY [, Farbe]).....	190
Elself	75
EnableDebugger.....	31
End.....	37, 83
EndIf	75
EndProcedure	86
EndSelect.....	76
Entstehungsgeschichte	10
Eof(#Datei)	173
Ersetzen.....	108
Erstelle.....	17
Erstellung einer DLL.....	149
EventGadgetID().....	227

EventMenuID()	227
EventType	117
EventType()	227
EventWindowID().....	228
ExamineDatabaseDrivers()	277
ExamineDirectory(#Verzeichnis, VerzeichnisName\$, Pattern\$)	178
ExamineJoystick()	224
ExamineKeyboard()	221
ExamineMouse()	219
Executable.....	17, 39
Executable Format.....	20
Exklusives Oder.....	69

F

FakeEndSelect	78
FakeReturn	84
Features.....	9
Fehlermeldungen	290
Fenster.....	95
File	153, 172
FileSeek(NeuePosition).....	173
FileSize(DateiName\$)	178
FileSystem	153, 176
FindString(String\$, SuchString\$, StartPosition)	162
FirstDatabaseRow()	277
FirstElement(LinkedList()).....	168
FlipBuffers()	208
Float	41
FLOAT.....	127
Font	155, 234
FontID().....	234
FontRequester	115
FontRequester(FontName\$, FontGröße, Flags)	258

For : Next Schleife	79
Forever	82
Frame3DGadget	99
Frame3DGadget(#Gadget, x, y, Breite, Höhe, Text\$, Flags)	240
FreeGadget(#Gadget)	240
FreeImage(#Image)	186
FreeMemory(#Memory).....	171
FreeMenu(#Menu)	256
FreeModule(#Module)	195
FreeMovie(#Movie)	199
FreePalette(#Palette)	184
FreeSound(#Sound).....	193
FreeSprite(#Sprite)	208
FreeSprite3D(#Sprite)	216
FreeStatusBar(#StatusBar)	263
FreeToolBar(#ToolBar)	262
FrontColour()	189, 190, 191
FrontColour(Rot, Grün, Blau)	190
Funktionen.....	86
Funktionsschalter in der Toolbar-Leiste.....	20

G

Gadget	155, 234
GadgetDEMO.pb.....	119
GadgetHeight(#Gadget)	241
GadgetID = GadgetID(#Gadget)	240
Gadgets	97
GadgetToolTip(#Gadget, Text\$)	240
GadgetWidth(#Gadget).....	241
GadgetX(#Gadget)	241
GadgetY(#Gadget).....	241
Garbage to the end of line	290
GDI.....	122

Gehe zu	15
GetClipboardData(Datentyp)	265
GetClipboardText()	265
GetDatabaseFloat(Spalte)	278
GetDatabaseLong(Spalte).....	278
GetDatabaseString(Spalte)	278
GetExtensionPart(DateiPfad\$).....	180
GetFilePart(DateiPfad\$)	180
GetGadgetState(#Gadget)	242
GetGadgetState()	237, 239, 243, 252
GetGadgetText(#Gadget).....	243
GetGadgetText().....	244
GetGadgetText():.....	254
GetMenuItemState.....	105
GetMenuItemState(#Menu, MenuItem).....	256
GetModulePosition().....	195
GetModuleRow()	196
GetPaletteColor(Index).....	184
GetPathPart(DateiPfad\$).....	180
Gleich.....	62
Global	89
Globale und lokale Variablen.....	88
Glossar	38
Gosub	21
Gosub : Return	83
Goto.....	78, 84, 85
GrabImage(#Image1, #Image2, x, y, Breite, Höhe)	186
GrabSprite(#Sprite, x, y, Breite, Höhe)	208
Graphics Device Interface.....	122
Green(Farbe)	179
Größer als.....	62, 63
Größer als oder gleich.....	62, 64
Grundlagen	33
Grundlegende Bedienung.....	12

GUI.....	39
----------	----

H

HACCEL	132
Haftungsausschluss.....	6
Handle	39
HANDLE.....	132
HBITMAP	132
HBRUSH	132
HCONV	132
HCONVLIST	132
HCURSOR	132
HDC.....	132
HDDEDATA.....	132
HDESK	132
HDROP.....	132
HDWP.....	132
Header.....	92
Help.....	156, 275
HENHMETAFILE.....	133
Hex(Wert).....	162
HFILE	133
HFONT	133
HGDIOBJ.....	133
HGLOBAL.....	133
HHOOK	133
HICON.....	133
HideGadget(#Gadget, Status)	243
HideWindow(#Window, Status).....	229
Hilfe.....	17, 109
HIMAGELIST	133
HIMC	133
HINSTANCE.....	133
HKEY	133

HKL.....	133
HLOCAL.....	133
HMENU.....	134
HMETAFILE.....	134
HMODULE.....	134
HMONITOR.....	134
HPALETTE.....	134
HPEN.....	134
HRGN.....	134
HRSRC.....	134
HSZ.....	134
HWINSTA.....	134
HWND.....	134
hWnd'.....	97

I

Icon.....	19
ID's.....	111
If : Else : EndIf.....	74
Illegal expression to For/Next. Must be: 'For Var=Expression To'	293
Image.....	154
ImageDepth()	186
ImageGadget.....	99
ImageGadget(#Gadget, x, y, Breite, Höhe, ImageID [, Flags])	243
ImageHeight().....	186
ImageID()	186, 235, 265
ImageOutput()	186, 188
Include'	142
IncludeFile	142
Info	17
Init.....	152
InitCDAudio().....	197

InitDatabase()	275
InitJoystick()	223
InitKeyboard()	221
InitModule(#Module)	195
InitMouse()	218
InitMovie()	199
InitNetwork()	270
InitPalette(#NumPaletteMax)	183
InitSound()	192
InitSprite()	202
InitSprite3D()	214
Inkey()	158
Inline ASM	140
InlineASM	20, 140
Inline-Assembler	139
Input()	33, 56, 57, 84, 158
InsertElement(LinkedList())	168
Installation	11
INT	128
INT_PTR	128
INT32	128
INT64	131
IPAddressField(IPAdresse, Feld)	271
IPAddressGadget	99
IPAddressGadget(#Gadget, x, y, Breite, Höhe)	243
IPAddressGadget()	271, 272
IsFunction(#Library, FunctionName\$)	289

J

Jens File Editor	146
Joystick	155, 223
JoystickAxisX()	224
JoystickAxisY()	224
JoystickButton(ButtonNumber)	224

K

Keyboard	154, 220
KeyboardPushed(KeyID).....	221
KeyboardReleased(KeyID).....	223
Keywords.....	40
KillThread(ThreadID).....	287
Klammern	72
Kleiner als	62, 63
Kleiner als oder gleich	62, 63
Kompilieren & Starten	21
Kompilieren/Starten	16
Konstanten	44
Kopieren	15, 21, 109

L

LANGID.....	129
LastElement(LinkedList()).....	168
LCase(String\$)	162
LCID	129
LCTYPE.....	129
Left(String\$, Länge).....	162
Len(String\$).....	163
Library	39, 156, 288
Lieferumfang.....	11
Line(x, y, Breite, Höhe [, Farbe])	190
LineXY(x1, y1, x2, y2 [, Farbe]).....	191
Linked List.....	153
LinkedList.....	166
Links	70, 150
Listen.....	48
Listeninhalt.....	49
ListIconGadget.....	100, 238

ListIconGadget(#Gadget, x, y, Breite, Höhe, Titel\$, TitelBreite [, Flags])	244
ListIconGadget()	235, 239, 242
ListIconGadget():	235
ListIndex(LinkedList())	168
ListViewGadget	100
ListViewGadget(#Gadget, x, y, Breite, Höhe)	245
ListViewGadget()	235, 238, 239, 242, 243
ListViewGadget() :	242
ListViewGadget():	241
LoadFont(#Font, Name\$, YSize)	234
LoadImage(#Image, Dateiname\$)	187
LoadJPEGSprite(#Sprite, Dateiname\$, Modus)	210
LoadModule(#Module, "Dateiname")	196
LoadMovie(#Movie, Dateiname\$)	200
LoadPalette(#Palette, Dateiname\$)	184
LoadSound(#Sound, Dateiname\$)	193
Loc()	173
Local'	90
Locate(x, y)	191
Lof()	173
Log(Zahl.f)	160
Log10(Zahl.f)	160
Logische Operatoren	65
Logisches Oder	65
Logisches Und	65
lokale Variablen	88
Long	41
LONG	128
LONG_PTR	128
LONG32	129
LONG64	131
LONGLONG	131
Löschen	109

LPARAM.....	129
LPBOOL.....	135
LPBYTE.....	135
LPCOLORREF	135
LPCRITICAL_SECTION	135
LPCSTR.....	135
LPCTSTR	135
LPCVOID.....	135
LPCWSTR	135
LPDWORD.....	135
LPHANDLE	135
LPINT	135
LPLONG	136
LPSTR	136
LPTSTR.....	136
LPVOID	136
LPWORD	136
LPWSTR.....	136
LRESULT	129
LTrim(String\$)	163
LUID	131

M

MakeIPAddress()	272
MakeIPAddress(Feld3, Feld2, Feld1, Feld0).....	272
Math	153, 159
MD5Fingerprint(Buffer, Länge)	285
Memory	153, 169
MemoryID()	170, 171
MemoryStringLength(*String)	171
Menu	155, 255
MenuBar	104
MenuBar()	256
Menü-Funktionen	13

MenuHeight.....	105
MenuHeight()	256
MenuItem.....	104
MenuItem(MenuID, Text\$).....	257
Menüs	104
MenuTitle.....	104
MenuTitle(Title\$)	257
MessageRequester(Titel\$, Text\$, Flags)	258
Microsoft Windows®	93
Mid(String\$, StartPosition, Länge).....	163
MIDAS	195
midas I I.dll	195
Minus.....	60
Misc	153, 179
Module.....	154, 194
Mouse	154, 218
MouseButton(ButtonNumber)	219
MouseDeltaX()	219
MouseDeltaY()	219
MouseWheel().....	220
MouseX()	220
MouseY().....	220
MoveWindow(x, y).....	229
Movie	154, 199
MovieAudio(Volume, Balance)	200
MovieHeight()	200
MovieInfo(Flags).....	200
MovieLength()	200
MovieSeek(Frame)	201
MovieStatus()	201
MovieWidth()	201
Multiplikation	59, 61

N

NASM	139, 140
Negation.....	70
Network	156, 270
Network Services	122
NetworkClientEvent(ConnectionID).....	272
NetworkClientID().....	272
Neu	13, 20, 108, 111
NewPrinterPage()	269
Next.....	79
NextDatabaseDriver()	278
NextDatabaseRow().....	279
NextDirectoryEntry().....	176, 177, 178
NextElement(LinkedList()).....	168
NextPackFile()	281
Nicht	70
nicht gleich zu	64
NOT	70
NT 4.0	20
Numerische Datentypen	41

O

Oder	68
------------	----

Ö

Öffnen.....	14, 20, 108, 111
-------------	------------------

O

Online-Hilfe.....	22
OpenConsole().....	33, 56, 57, 84, 157
OpenDatabase(#Database, ODBCDatabaseName\$, User\$, Password\$)	279
OpenDatabaseRequester(#Database).....	279

OpenFile(#Datei, DateiName\$)	173
OpenFileRequester	115
OpenFileRequester(Titel\$, StandardDatei\$, Pattern\$, PatternPosition)	259
OpenHelp(Dateiname\$, Thema\$)	275
OpenNetworkConnection(ServerName\$, Port)	273
OpenPack(DateiName\$)	282
OpenPreferences(DateiName\$)	266
OpenScreen()	203
OpenScreen(Breite, Höhe, Tiefe, Titel\$)	210
OpenSubMenu	105
OpenSubMenu(Text\$)	257
OpenTreeGadgetNode()	246
OpenWindow	97
OpenWindow(#Window, x, y, InnereBreite, InnereHöhe, Flags, Titel\$)	229
OpenWindowedScreen()	203
OpenWindowedScreen(WindowID, x, y, Breite, Höhe, AutoStretch, RightOffset, BottomOffset)	211
Operating System	39
Operatoren	58
OptionGadget	100
OptionGadget(#Gadget, x, y, Breite, Höhe, Text\$)	246
OptionGadget()	242
Or	65, 66
OR	68
OS	39
Overflow error: a 'byte' value (.b) must be between -128 and +255	291

P

Packer	156, 280
PackerCallback(@Procedure())	282
PackFileSize()	282

PackMemory(SourceMemoryID, DestinationMemoryID, SourceLength)	283
Palette	153, 183
PanelGadget.....	100, 235, 238, 242
PanelGadget(#Gadget, x, y, Breite, Höhe).....	246
PanelGadget().....	239, 242
Parameter	39
PathRequester.....	116
PathRequester(Titel\$, UrsprungsPfad\$)	259
PauseCDAudio()	198
PauseMovie().....	201
PauseThread(ThreadID)	287
PB_MessageRequester_YesNoCancel	259
PBCompiler.exe.....	24
PBOOL.....	136
PBOOLEAN	136
PBYTE.....	136
PCHAR.....	136
PCritical_SECTION	136
PCSTR	136
PCTSTR.....	136
PCWCH.....	136
PCWSTR.....	137
PDWORD	137
PeekBWL(*MemoryBuffer).....	182
PeekF(*MemoryBuffer)	182
PeekS(*MemoryBuffer [, Länge])	182
PFLOAT	137
PHANDLE	137
PHKEY	137
PINT.....	137
PlayCDAudio(StartTrack, EndTrack)	198
PlayModule(#Module).....	196
PlayMovie(#Movie, WindowID)	201

PlaySound(#Sound [, Modus]).....	193
PLCID	137
PLONG	137
Plot(x, y [, Farbe]).....	191
PLUID	137
Plus	60
Point(x, y)	191
Pointer	40, 54
POINTER_32	137
POINTER_64	137
PokeBWL(*MemoryBuffer, Number.L)	182
PokeF(*MemoryBuffer, Number.f).....	183
PokeS(*MemoryBuffer, Text\$ [, Länge])	183
Popup-Menüs	106
Pow(Basis.f, Exponent.f)	160
Preference	155, 265
PreferenceComment(Text\$).....	267
PreferenceGroup(Name\$).....	267
PreviousDatabaseRow()	280
PreviousElement(LinkedList())	168
Print(Text\$)	158
Printer.....	155, 269
PrinterOutput()	188, 269
PrinterPageHeight()	269
PrinterPageWidth()	270
PrintN(Text\$).....	159
PrintRequester().....	269
Procedure	86
Programmierung.....	140
ProgramParameter().....	181
ProgressBarGadget.....	100
ProgressBarGadget(#Gadget, x, y, Breite, Höhe, Minimum, Maximum [, Flags])	247
Protected	90

Prozedur	86
Prozeduren	86
PSHORT	137
PSTR	137
PTBYTE	137
PTCHAR	138
PTSTR	138
PUCHAR	138
PUINT	138
PULONG	138
PureBasic\Compilers	24
PUSHORT	138
PVOID	138
PWCHAR	138
PWORD	138
PWSTR	138

Q

Quit	14, 29
------------	--------

R

Random(Maximum)	181
RandomSeed(Wert)	182
ReadByte()	174
ReadData(*MemoryBuffer, LengthToRead)	174
ReadFile(#Datei, DateiName\$)	174
ReadLong()	174
ReadPreferenceFloat(Keyword\$, StandardWert.f)	267
ReadPreferenceLong(Keyword\$, StandardWert)	267
ReadPreferenceString(Keyword\$, StandardWert\$)	268
ReadWord()	174
ReAllocateMemory(#Memory, Größe)	172

ReceiveNetworkData(ConnectionID, *DataBuffer, Länge)	273
ReceiveNetworkFile(ConnectionID, Dateiname\$).....	274
Rechts	71
Red(Farbe)	179
Redo	15
REGSAM	130
Rekursivität.....	91
ReleaseMouse()	220
RemoveGadgetItem()	238
RemoveKeyboardShortcut(#Window, Shortcut).....	230
RemoveSysTrayIcon(#SysTrayIcon)	286
RenameFile(AlterDateiname\$, NeuerDateiname\$)	179
Repeat : Until	82
ReplaceString(String\$, SuchString\$, ErsatzString\$ [, Modus])	163
Requester.....	112, 155, 257
ResetList(LinkedList())	169
ResizeGadget(#Gadget, x, y, Breite, Höhe).....	248
ResizImage(#Image, Breite, Höhe).....	187
ResizeMovie(x, y, Breite, Höhe).....	202
ReSizeWindow(Breite, Höhe).....	230
ResumeCDAudio()	198, 199
ResumeMovie().....	202
ResumeThread(ThreadID)	287
Return.....	83
RGB(Rot, Grün, Blau)	180
Right(String\$, Länge)	164
RotateSprite3D(#Sprite3D, Winkel, Modus)	216
Round(Zahl.f, Modus)	160
RTrim(String\$).....	164
Rückgabewert	86
Rückgängig	109
RunProgram(Dateiname\$, Parameter\$, Flags).....	181

S

SaveFileRequester	116
SaveFileRequester(Titel\$, StandardDatei\$, Pattern\$, PatternPosition)	260
SaveImage(#Image, DateiName\$)	187
SaveSprite(#Sprite, DateiName\$)	211
SC_HANDLE	134
SC_LOCK	134
Schalter	97
Schleifen	79
Schlüsselwörter	40
Screen	154
ScreenID()	211
ScreenOutput()	188, 212
Select : EndSelect	76
SelectedFontColor()	260
SelectedFontName()	260
SelectedFontSize()	260
SelectedFontStyle()	261
SelectElement(LinkedList(), Position)	169
SendNetworkData(ConnectionID, *MemoryBuffer, Länge)	274
SendNetworkFile(ConnectionID, DateiName\$)	274
SendNetworkString(ConnectionID, String\$)	274
SERVICE_STATUS_HANDLE	135
SetClipboardData(Datentyp, DataID)	265
SetClipboardText(Text\$)	265
SetFrameRate(FrameRate)	212
SetGadgetFont(FontID)	248
SetGadgetItemState(#Gadget, Eintrag, Status)	248
SetGadgetItemText(#Gadget, Eintrag, Text\$, Spalte)	249
SetGadgetState(#Gadget, Status)	249
SetGadgetState()	237, 238, 244, 252
SetGadgetState():	254

SetGadgetText(#Gadget, Text\$)	249
SetGadgetText()	244, 254
SetMenuItemState	105
SetMenuItemState(#Menu, MenuItem, Status)	257
SetModulePosition(Position)	196
SetPaletteColor(Index, Farbe)	184
SetRefreshRate(RefreshRate)	212
SetWindowCallback(@ProcedureName())	230
Shared	89
SHORT	127
Show Debug Window	29
Show Variables	29
Sin(Winkel.f)	161
SIZE_T	131
Sort	153, 165
SortArray(ArrayName(), Option [, Start, Ende])	166
Sound	154, 192
SoundFrequency(#Sound, Frequenz)	193
SoundPan(#Sound, Pan)	194
SoundVolume(#Sound, Volume)	194
Sourcecode	40
Space(Länge)	164
Speichern	14, 21, 108, 111
Speichern als	14
Spieleprogrammierung	121
SpinGadget	101
SpinGadget(#Gadget, x, y, Breite, Höhe, Minimum, Maximum)	250
Sprite	154, 202
Sprite3D	154, 214
Sprite3DBlendingMode(Ausgangsmodus, Zielmodus)	216
Sprite3DQuality(Qualität)	217
SpriteCollision(#Sprite1, x1, y1, #Sprite2, x2, y2)	212
SpriteDepth(#Sprite)	212

SpriteHeight(#Sprite)	212
SpriteOutput(#Sprite).....	213
SpriteOutput()	188
SpritePixelCollision(#Sprite1, x1, y1, #Sprite2, x2, y2) ...	213
SpriteWidth(#Sprite)	213
Sprungbefehle.....	82
Sprungmarken	21
Sqr(Zahl.f)	161
SSIZE_T	131
Standard-Menüs.....	104
Start.....	152
Start3D()	217
StartDrawing()	190
StartDrawing(OutputID)	188
Starten	16
StartPrinting(JobName\$)	270
StartSpecialFX()	206, 213
StatusBar	155, 263
StatusBarIcon(#StatusBar, Feld, ImageID)	263
StatusBarText(#StatusBar, Feld, Text\$, Aussehen)	264
Step	29
Stop	29
Stop3D()	217
StopCDAudio()	199
StopDrawing().....	191, 192
StopModule(#Module).....	196
StopMovie().....	202
StopPrinting()	270
StopSound(#Sound).....	194
StopSpecialFX()	214
Str(Wert).....	88, 164
Strange error: object type not detected	293
StrF(Wert.f [, NbDecimal])	164
String.....	42, 153, 161

StringGadget.....	101
StringGadget(#Gadget, x, y, Breite, Höhe, Inhalt\$ [, Flags])	250
StringGadget().....	243
StrU(Wert, Typ)	164
Strukturen.....	50
strukturierte.....	85
Subtraktion	59, 60
Suchen	16, 108, 111
Syntax	36, 40, 139
Syntax Error	290
Syntax error.....	292
Systemvoraussetzungen	11
SysTray	156, 285
SysTrayIconTooltip(#SysTrayIcon, Text\$).....	286

T

Tan(Winkel.f)	161
Tastatur-Abfrage	116
Tastatur-Shortcut.....	230
Tastenkürzel.....	230
TBYTE	127
TCHAR	127
Text\$ = GetGadgetItemText(#Gadget, Eintrag, Spalte)..	242
Text\$ = ReadString()	174
TextGadget.....	101
TextGadget(#Gadget, x, y, Breite, Höhe, Text\$ [, Flags])	251
TextGadget().....	243
TextLength(Text\$)	192
The specified number is incorrect for '<<' or '>>' operator (must be 1-31).....	292
The specified variable type is different than before	291
Then	75

Thread	40, 156, 286
ThreadPriority(ThreadID, Priorität)	287
Too many '(' parenthesis	290
ToolBar	155
ToolBar	261
ToolBarImageButton(#ButtonID, ImageID)	262
ToolBars	107
ToolBarSeparator()	262
ToolBarStandardButton	108
ToolBarStandardButton(#ButtonID, #ButtonIcon)	262
ToolBarToolTip(#ButtonID, Text\$)	263
TrackBarGadget	101
TrackBarGadget(#Gadget, x, y, Breite, Höhe, Minimum, Maximum [, Flags])	252
TransformSprite3D(#Sprite3D, x1, y1, x2, y2, x3, y3, x4, y4)	218
TransparentSpriteColor(#Sprite, Rot, Grün, Blau)	214
TreeGadget	102
TreeGadget(#Gadget, x, y, Breite, Höhe [, Flags])	252
TreeGadget()	235, 238, 239, 242
TreeGadget() :	242
Trim(String\$)	165
Trying to write a numeric value into string variable	291
Trying to write a string into numeric variable	291

U

UCase(String\$)	165
UCHAR	127
UINT	130
UINT_PTR	130
UINT32	130
UINT64	131
ULONG	130
ULONG_PTR	130

ULONG32.....	130
ULONG64.....	131
ULONGLONG	131
UltraEdit.....	146
Umwandlung von API-Datentypen.....	126
Undo	15
Ungleich.....	62, 64
UnpackMemory(SourceMemoryID, DestinationMemoryID)	283
UNSIGNED	127
Until	82
UpdateStatusBar(#StatusBar).....	264
UseCDAudio(CDAudioDrive).....	199
UseDatabase(#Database).....	280
UseDirectory(#Verzeichnis)	179
UseFile(#Datei)	175
UseFont(#Font).....	234
UseGadgetList.....	103
UseGadgetList(WindowID)	253
UseImage(#Image)	187
UseMemory(#Memory).....	172
UseMovie(#Movie).....	202
UsePalette(#Palette)	185
User Interface	122
User Interface).....	122
UseWindow(#Window).....	231
USHORT	127

V

Val(String\$).....	165
ValF(String\$).....	165
Variablen	41
Variablentypen	41
Vergleichsoperatoren.....	62

Verschieben.....	70, 71
Verwendung alternativer Editoren.....	145
von WinAPI-Funktionen.....	124
Voreinstellungen.....	14, 17
Vorwort	7
Vorzeichen.....	58

W

WaitWindowEvent.....	117
WaitWindowEvent().....	231
WCHAR	127
WebGadget.....	102
WebGadget(#Gadget, x, y, Breite, Höhe, URL\$)	254
Weitersuchen	16
Wend.....	81
Werkzeugleiste.....	108
Werkzeugleisten	107
Werkzeug-Leisten.....	104
While : Wend	81
Wiederholen	109
WINAPI.....	130
WinAPI-Dokumentation	123
Window	155, 225
WindowEvent	117
WindowEvent()	231
WindowHeight()	232
WindowID([#Window])	232
WindowMouseX()	233
WindowMouseY()	233
WindowOutput().....	188, 233
Windows.....	93
Windows API.....	40
Windows Shell.....	122, 123
Windows XP Skin Support.....	20

Windows-API.....	122
Windows-Konsole.....	24
Windows-Programmierung	93
WindowWidth().....	233
WindowX()	233
WindowY().....	233
Word	41
WORD.....	127
WPARAM	129
WriteByte(Number.b)	175
WriteData(*MemoryBuffer, LengthToWrite).....	175
WriteLong(Number.w).....	175
WritePreferenceFloat(Keyword\$, Wert.f)	268
WritePreferenceLong(Keyword\$, Wert)	268
WritePreferenceString(Keyword\$, Wert\$)	268
WriteString(Text\$).....	175
WriteStringN(Text\$)	175
WriteWord(Number.w)	175

X

XIncludeFile.....	142
XOR.....	69

Z

Zeige Debugger-Fenster.....	29
Zeige Variablen.....	29
Zeiger	54
ZoomSprite3D(#Sprite3D, Breite, Höhe)	218
Zusätzliche Informationen	142
Zuweisungsoperator	58