Aktualisiert für PureBasic V4.5x

Deutsche Erstausgabe

PureBasic

Eine Einführung in die Computer Programmierung

Beta Version

Aktuelle Informationen zum Projekt unter:
http://www.purebasic.fr/german/viewtopic.php?f=1&t=23627
Stand: 01.06.2011, 22:19 Uhr

Gary Willoughby

PureBasic

Eine Einführung in die Computer Programmierung

Gary Willoughby

PureBasic - Eine Einführung in die Computer Programmierung
von Gary Willoughby

Fantaisie Software
10, rue de Lausanne
67640 Fegersheim France
www.purebasic.com

Herausgeber 2006, Aardvark Global Publishing.

Autor
Gary Willoughby

Druck
1. Auflage Juli 2006

Übersetzung ins Deutsche und Anpassung an aktuelle PureBasic Version
Andreas Schweitzer (Juli 2010 - Dezember 2010)
1. Auflage X 2011

Stand: 01.06.2011

Widmung

Dieses Buch widme ich meiner Freundin Sara Jane Gostick und ihrem Hund 'Stella' für ihre vielen einsamen Nächte und die liebevolle Versorgung mit Mahlzeiten während ich dieses Buch schrieb.

Widmung des Übersetzers

Ich widme dieses Buch meiner Frau Constanze und unseren drei Kindern Tabea, Johanna und Manuel für die großartige Unterstützung bei der Schaffung von Freiräumen, um an diesem Werk zu arbeiten.

Danksagung

Danke an Fred und das PureBasic Team für die PureBasic Programmiersprache, macht weiter so mit der genialen Arbeit!
Danke an Paul Dixon für die Klarstellung einiger Details zur Binären Kodierung von Fließkommazahlen.
Danke an Timo Harter für die Hilfe bei der Auswahl der passenden PureBasic Datentypen als Ersatz für die Win32 API Datentypen,
und für die Demonstration wie man Mithilfe der verschiedenen Win32 API String Pointer Zeichenketten im Speicher aufspürt.

"Aus großer Kraft folgt große Verantwortung!"
--Ben Parker (Spiderman's Onkel)

Die Beispiel Codes in diesem Buch können kostenlos von folgender Seite heruntergeladen werden
www.pb-beginners.co.uk (englisch)
asw.gmxhome.de (deutsch)

Dieses Buch wurde mit "TextMaker 2008" erstellt.
www.softmaker.de

ISBN 1-4276-0428-2 19/7/2006 (englische Originalausgabe)

Inhalt

Vorwort

Über dieses Buch

Dieses Buch ist eine schnelle Einführung in die Programmiersprache PureBasic, dessen Popularität in den letzten Jahren sehr stark gewachsen ist. Es wird für sehr vielfältige Zwecke eingesetzt wie z.B. schnelle Software Entwicklung, Erstellung von kommerziellen Anwendungen und Spielen sowie Internet CGI Anwendungen. Viele benutzen es einfach zum Erstellen kleiner Werkzeuge. Dieses Buch wurde mit einem Blick auf den kompletten Anfänger geschrieben. Jeder muss irgendwo anfangen und ich denke das PureBasic ein fantastischer erster Schritt in die Programmierwelt ist. Mit der weiteren Verbreitung von PureBasic kommt es immer häufiger vor, dass Menschen einen "Schubs in die richtige Richtung" oder eine Erklärung für eine bestimmte Funktion benötigen. Dieses Buch ist ein Leitfaden für den Einsteiger, der ihn durch die ersten Schritte führt. Fortgeschritteneren Programmierern gibt das Buch einen schnellen Einblick in die Sprache.

Der Themenbereich dieses Buches

Obwohl dieses Buch die wesentlichen Elemente von PureBasic abdeckt, habe ich den Themenbereich sehr eng gehalten, um neue Benutzer nicht mit zu viel Informationen zu überfrachten. Manchmal verweist der Text auf Konzepte und Syntax einfachster Art, was als ein Zwischenschritt zu fortgeschritteneren Themen oder ein Querbezug zur PureBasic Hilfe verstanden werden kann.

Ich werde zum Beispiel nicht viel darüber erzählen wie einfach es ist, DirectX oder OpenGL direkt in PureBasic zu verwenden, sonst wäre dieses Buch dreimal so dick, und auch wenn Themen wie Zeiger und das Win32 API später in diesem Buch besprochen werden, erwarten Sie nicht zu viele spezielle Beispiele, da ich nur einen Überblick zu den fortgeschrittenen Themen liefere.

PureBasic hebt die Messlatte nicht nur für Basic Dialekte, sondern für Programmiersprachen allgemein: saubere, nicht überladene Syntax, kleine ausführbare Dateien und eine fantastische aktive Programmiergemeinde. Ich hoffe dieses Buch gibt Ihnen ein klares Verständnis der Kernfunktionen von PureBasic und den dahinterstehenden Design Zielen und der Philosophie, und ich hoffe Sie wollen mehr lernen.

Ungeachtet des begrenzten Themenbereiches denke ich, Sie haben hier ein gutes erstes Buch über PureBasic, das eine gute Grundlage zum Programmieren bildet, selbst wenn Sie später zu anderen Programmiersprachen wechseln. Sie werden alles lernen was Sie zum schreiben von nützlichen, eigenständigen Programmen benötigen. Wenn Sie dieses Buch durchgearbeitet haben, werden Sie nicht nur die Grundlagen der Sprache verstanden haben, sondern Sie werden in der Lage sein das erworbene Wissen in Ihren Programmieralltag zu transportieren. Sie werden gut ausgerüstet sein, sich an fortgeschrittenere Themen zu wagen, die sich Ihnen in den Weg stellen.

Die Struktur dieses Buches

Vieles in diesem Buch ist so gestaltet, Ihnen so schnell wie möglich einen Einstieg in PureBasic zu geben. Die wesentlichen Sprachelemente sind in einzelne Abschnitte unterteilt. Jedes Kapitel ist weitestgehend in sich abgeschlossen, aber spätere Kapitel, besonders die fortgeschrittenen, berufen sich auf Informationen aus vorhergehenden Kapiteln. Wenn wir zum Beispiel die grafischen Benutzeroberflächen und die Grafik Programmierung behandeln, setzt ich voraus, das Sie die Prozeduren und Strukturen verstanden haben.

Teil I: Die Grundlagen der Sprache
Dieser Teil des Buches ist geradlinige, von Grund auf geführte Erklärung der wesentlichen Elemente der Sprache, wie Typen, Prozeduren, Strukturen und so weiter. Die meisten der angeführten Beispiele sind als eigenständige Programme nicht sehr nützlich, aber sie demonstrieren und erklären den aktuell behandelten Punkt.

Kapitel 1, Wir Beginnen
Hier beginnen wir mit einer schnellen Einführung in PureBasic und die Geschichte hinter der Sprache. Sie bekommen einen Überblick, wie PureBasic Programme aus der IDE gestartet werden und für was der Debugger gut ist.

Kapitel 2, Datentypen
In diesem Kapitel liste ich alle eingebauten Datentypen, wie Strings, numerische Typen und Konstanten, auf und erkläre sie. Ich werde Ihnen Richtlinien zur Verwendung, sowie Informationen über die numerischen Limits und den Speicherverbrauch geben.

Kapitel 3, Operatoren
Hier beschreibe ich, wie Sie Variablen Werte zuweisen und welche Operatoren zur Berechnung von Daten nötig sind. Eine vollständige Erklärung aller Operatoren mit Tabellen und Beispielen ist vorhanden. Der 'Debug' Befehl wird ebenfalls in diesem Kapitel vorgestellt, da er einer der nützlichsten Befehle in PureBasic ist, und dessen Verwendung man sehr früh erlernen sollte.

Kapitel 4, Bedingungen und Schleifen
In diesem Kapitel erkläre ich, wie PureBasic mit Booleschen Werten umgeht. Die 'If' und 'Select' Anweisungen sowie Schleifen werden vorgestellt und ausführlich an Beispielen erklärt.

Kapitel 5, Andere Datenstrukturen
Dieses Kapitel offenbart andere Methoden zum speichern und organisieren von Daten, wie z.B. benutzerdefinierte Strukturen, Arrays und Listen. All dies wird vollständig erklärt und mit Beispielen versehen.

Kapitel 6, Prozeduren und Unterroutinen
Prozeduren und Unterroutinen sind ein wesentliches Element jeder Programmiersprache, da sie es ermöglichen bestimmte Code Abschnitte (im Falle von Prozeduren sogar mit der Übergabe von Variablen als Parameter) an jeder Stelle im Programm aufzurufen. Das erleichtert die Programmierung erheblich, da das eigentliche Programm in leichter handhabbare Teilprobleme aufgeteilt werden kann. Diese modulare Bauweise erleichtert zudem die Wiederverwendbarkeit von bereits erstelltem Code für andere Projekte.

Kapitel 7, Verwendung der eingebauten Befehle
Dieses Kapitel demonstriert einige der am häufigsten verwendeten Befehle. Es ist keine komplette Referenz oder ein Leitfaden zu jedem einzelnen Befehl oder jeder Befehlsbibliothek, aber es bildet ein gutes Fundament, wie und wann Sie die eingebauten Befehle benutzen können. Es gibt auch eine Erklärung zu Handles und ID's, welche beide recht leicht zu verstehen sind, manchmal aber für etwas Verwirrung sorgen.

Kapitel 8, Guter Programmierstil
Dieses Kapitel ist ein Leitfaden für gute Programmierpraktiken, die Sie während der Benutzung dieses Buches begleiten und es verschafft einen Überblick zu einfachen Fehlerbehandlungstechniken. Wenn Sie in irgendeiner Sprache programmieren, sind Fehler immer ein Problem, sei es ein einfacher Tippfehler oder ein Fehler in der Sprache selbst. Dieses Kapitel zeigt Wege auf, wie sie bewusst Fehler vermeiden können und wie und warum Sie in Ihrem Programm auf Fehler überprüfen, und wie Sie mit gefundenen Fehlern umgehen sollen.

Teil II: Grafische Benutzeroberflächen
Nahezu jedes heutige Programm hat eine beschreibende Benutzeroberfläche, und hier werde ich Ihnen zeigen, wie Sie solche erstellen. Aufbauend auf Ideen und Beispielen für Konsolen Anwendungen werden Sie lernen, wie man Windows Anwendungen mit Standard Controls (Gadgets) aufbaut, inklusive Menüs, Schaltflächen und Grafik.

Kapitel 9, Erstellen von Benutzeroberflächen
Hier zeige ich Ihnen, wie Sie ihre eigenen Benutzeroberflächen erstellen können. Wir beginnen mit der Erklärung und Demonstration von Konsolenprogrammen und gehen dann zu Fensterbasierten Oberflächen über. Ereignisse werden ebenfalls beschrieben, und es gibt viele Beispiele, wie Sie reagieren sollten, wenn ein Ereignis in Ihrer Benutzeroberfläche auftritt. Der mitgelieferte Visual Designer wird ebenfalls kurz besprochen.

Teil III: Grafik und Sound
Grafik und Sound haben einen enormen Stellenwert in nahezu jedem heutigen Computersystem. Dieser Abschnitt zeigt, wie Sie Klänge abspielen, Grafiken auf dem Bildschirm anzeigen und dort manipulieren können, das ganze in 2D oder 3D.

Kapitel 10, 2D Grafik
Diese Kapitel führt Sie in die Welt der zweidimensionalen Grafiken, wie Linien und Formen, ein und wie sie auf den Bildschirm gezeichnet werden. Weiterhin werden 'Sprites' (Bilder die angezeigt und manipuliert werden können) 'Screens' und 'Double Buffering' besprochen.

Kapitel 11, 3D Grafik
Dreidimensionale Grafiken werden in PureBasic von der 'OGRE Engine' erzeugt. In diesem Kapitel gibt es einen Überblick und einige Beispiele, die demonstrieren, was mit dieser Grafikengine möglich ist. Die 'OGRE Engine' steht unter ständiger Weiterentwicklung und ist vollständig in PureBasic integriert. Damit sind schon einige nette Dinge möglich.

Kapitel 12, Sound
Dieses Kapitel zeigt, wie Sie Klänge mit PureBasic verwenden und es zeigt, wie Sie bekannte Sound Dateiformate laden und abspielen.

Teil IV: Fortgeschrittene Themen
Der letzte Teil des Buches behandelt Themen, die ein Einsteiger als sehr fortgeschritten betrachtet. Die Themen in diesem Teil müssen nicht verstanden werden, um vollwertige Programme zu schreiben, aber mit ihnen sind einige Sachen möglich, die mit den normalen Befehlen nicht zu realisieren sind. Dieser Teil dient dazu Ihnen den Mund wässrig zu machen, Ihr Wissen und Verständnis von PureBasic, und dem Programmieren allgemein, ständig zu erweitern.

Kapitel 13, Jenseits der Grundlagen
In diesem Kapitel geht es um fortgeschrittenes Speichermanagement mittels Zeigern. Compiler Direktiven werden erklärt und Sie bekommen einen Leitfaden für die Erstellung von DLL's. Ein weiterer Teil behandelt das Windows Application Programming Interface.

Teil V: Anhang
Dies ist der letzte Abschnitt dieses Buches, und es endet mit nützlichen Links zu Internetseiten, hilfreichen Diagrammen und einem Glossar, in dem wichtige Begriffe erklärt werden.

Schlusswort
Ich hoffe, Sie wissen, wie man einen Computer benutzt. In diesem Buch wird nicht darüber gesprochen, wie man eine Maus bedient oder was ein 'Icon' ist, aber was das Programmieren betrifft, gehe ich von einem absoluten Neuling aus (nicht nur mit PureBasic, sondern Programmieren generell).

Alles was Sie zum Starten benötigen, ist ein wenig Zeit und eine Kopie von PureBasic, die Sie unter www.purebasic.com erhalten.

I. Die Grundlagen der Sprache

In diesem Abschnitt werden wir die Programmiersprache PureBasic untersuchen. Ich habe diesen Teil 'Die Grundlagen der Sprache' genannt, weil wir uns auf die wesentlichen Elemente der Sprache konzentrieren: die eingebauten Typen, Anweisungen und Ausdrücke. Wenn Sie diesen Teil gelesen und die Beispiele nachvollzogen haben, werden Sie in der Lage sein, eigene Programme zu erstellen.

Der Begriff 'Grundlagen' in der Überschrift soll verdeutlichen, dass dieser Abschnitt keine erschöpfende Abhandlung zu jedem Detail von PureBasic ist. Ich werde mit Sicherheit einige Punkte überspringen, aber das hier gelernte wird eine solide Basis zum Lösen unbekannter Problemstellungen bilden.

Die Geschichte von PureBasic sowie dessen Entwicklungsphilosophie werde ich dem Interessierten Leser natürlich ebenfalls nicht vorenthalten.

1. Wir Beginnen

Dieses erste Kapitel beginnt mit einem kurzen Abriss der Geschichte von PureBasic, dann werden wir kurz sehen wie PureBasic Programme funktionieren. Das Primäre Ziel ist es Sie in die Lage zu versetzen PureBasic zu installieren, Programme zu kompilieren und die Programme auf Ihrem Computer zu starten. Danach sind Sie in der Lage sich anhand der Anleitungen und der Beispiele durch das Buch zu arbeiten. Im weiteren Verlauf werden wir verschiedene Möglichkeiten der Kompilierung von PureBasic Programmen kennenlernen - kurz gesagt, Sie bekommen alle nötigen Informationen zum beginnen.
Wir werden weiterhin einen Blick auf die mitgelieferte Entwicklungsumgebung (IDE) werfen. Sie wirkt auf neue Benutzer etwas abschreckend, aber nach einer kurzen Einführung ist dieser anfängliche Schrecken schnell vorüber.

Die Geschichte von PureBasic

Die Geschichte von PureBasic begann 1995 als eine Befehlserweiterung für BlitzBasic, nachdem PureBasic's Autor Frederic Laboureur auf viele Einschränkungen in BlitzBasic gestoßen war, die ihn bei der Programmierung einer Anwendung mit Namen 'TheBoss' (ein mächtiger Programmstarter für den Commodore Amiga) behinderten. Die Erweiterung mit dem Namen 'NCS' (NewCommandSet) wurde komplett in 68000 Assembler programmiert, da zu dieser Zeit alle neuen Befehle für BlitzBasic in Assembler programmiert werden mussten. Fred's Fortschritt war sehr langsam, da es schwer war, gute Assembler Dokumentationen zu finden und Online Foren für BlitzBasic Plugin Programmierung existierten noch nicht.

Nachdem die Entwicklung von 'NCS' bereits ein Jahr lief, bekam er sehr positive Rückmeldungen zu seiner Arbeit, die ihn sehr vertraut mit der Assembler Programmierung und dem Debugging gemacht hatte. Er war auch sehr erstaunt darüber, welche phantastischen Dinge mit einem alten 68000 Prozessor verwirklicht werden konnten, wenn alles korrekt programmiert wurde.

Zu dieser Zeit erschienen für den Amiga IBM PowerPC basierende Prozessorkarten, die eine sehr mächtige Alternative zu den Motorola 68000 Prozessoren darstellten. Sie waren sehr schnell und wurden, im Vergleich zu den High-End 68060 Prozessoren, zu einem relativ moderaten Preis verkauft. Mit dem eintreffen dieser neuen Chips wollten die Programmierer eine Native Version von BlitzBasic für diese Prozessoren, da es zu dieser Zeit eine sehr populäre Sprache war, aber alle wussten, dass die Entwicklung für die Amiga Plattform, zugunsten der Intel x86 basierenden PCs, eingefroren wurde. Eine Gelegenheit für die Erstellung einer neuen Programmiersprache bot sich an. Diese sollte ein logischer Nachbau sowie eine Erweiterung von BlitzBasic sein, die volle Unterstützung für die 680x0- und die PowerPC Prozessoren bot.

Bahn frei für PureBasic!

Das frühe Design und die erste Version Von PureBasic begannen 1998. Der Hauptunterschied zwischen PureBasic und 'normalen' Compilern war der Einbau eines 'Virtuellen Prozessors' (der eigentlich die 680x0 Assembler Mnemonics benutzte) direkt zu Beginn, was verschiedene Arten von Assembler Ausgabe (oder jede andere Sprache) ermöglichte, ohne den Compiler Kern zu ändern. Nachdem das anfängliche Design fertig war und die Programmierung begann, kam das Projekt schnell in Fahrt. Fred widmete all seine Zeit der Compiler Programmierung und lernte eine Menge darüber, die Sprache C in sein Projekt einfließen zu lassen, um die Möglichkeit zu haben, einen vollständig portablen Compiler zu erstellen.

Die erste PureBasic Version wurde nur für den Amiga veröffentlicht, und hatte einen (wenn auch sehr fehlerhaften) vollständig plattformübergreifenden Editor, einen integrierten Debugger und sehr viele integrierte Befehle, die (Sie haben es bereits erraten) direkt dem vorhergehenden 'NCS' Blitz Paket

entnommen wurden. Während der Verbesserung und der Fehlerbehebung studierte Fred weitere Programmiersprachen um mehr Sicherheit in anderen Gebieten zu gewinnen und um ein gutes Fundament für die interne Designplanung zu bekommen und wie PureBasic in Zukunft wachsen und sich erweitern soll.

Während des vierten Jahres von Fred's Studium zum Diplom Informatiker wurde der Amiga immer mehr zu einer toten Plattform und viele seiner Mitstudenten fragten ihn, warum er nicht an einer Windows basierten Version arbeiten wolle. Fred verteidigte sich mit der Aussage, das es ein leichtes sei, PureBasic auf ein neues System zu portieren, aber er musste es nachweisen.

Eine Kurze Information zur Assemblersprache

Assemblersprache oder kurz Assembler, ist eine von Menschen lesbare Form der Maschinensprache, die eine bestimmte Computerarchitektur benutzt. Maschinensprache, die aus Bitmuster codierten Maschinenbefehlen besteht, wird in eine lesbare Form gebracht, indem die Rohbefehle durch Symbole ersetzt werden. Diese Symbole nennt man Mnemonics.

Die Programmierung in Maschinencode, indem man den Computer mit Nummern der Befehle versorgt, ist eine große Bürde, da zu jedem Befehl die entsprechende Nummer nachgeschlagen oder gemerkt werden muss. Daher ersann man sich einen Satz mit Mnemonics. Jede Nummer bekam einen alphabetischen Code zugewiesen. Anstatt zum Beispiel die entsprechende Nummer für die Addition einzugeben, um zwei Zahlen zu addieren, war von nun an nur noch ein 'ADD' nötig. Assembler Quelltext wird mit einem Assembler kompiliert.

Eine größere Arena

Fred begann Microsoft DirectX und Win32 API Programmierung (siehe Kapitel 13) zu lernen - komplett in Assembler - eine enorme Leistung! Den Intel x86 zu Programmieren war für ihn, mit seinem Motorola 680x0 Hintergrund, ein Albtraum, da die beiden Chips ein völlig unterschiedliches Design haben. Sogar die interne Speichermethode für Nummern war umgekehrt. Nach drei Monaten Entwicklung und der Gründung seiner neuen Firma 'Fantaisie Software', veröffentlichte er eine neue Webseite - PureBasic für Windows war fertiggestellt. Die Zahl der PureBasic Benutzer und Tester wuchs und Fred erhielt sehr viele hilfreiche und begeisterte Mails, die ihn weiter anspornten die bestmögliche Sprache zu entwickeln.

Nach mehreren Jahren sorgfältiger Entwicklung bildete sich ein Team um Fred, das ihn bei der Entwicklung und beim Testen neuer Versionen unterstützte. Dieses Team setzt sich aus erfahrenen Programmierern, Web Designern und Dokumentationsschreibern zusammen, die alle Fred's Vision der Sprache teilten.

Nach dem sehr großen Erfolg der Windows Version war der nächste logische Schritt, weitere Betriebssysteme zu unterstützen. So wurden später, einem wachsenden und begeisterten Publikum, eigene Linux und Mac OS Versionen von PureBasic präsentiert. Alle Versionen unterstützen die jeweilige Schnittstelle zur Anwendungsentwicklung (Application Programming Interface [API]) dieses bestimmten Betriebssystems sowie die jeweilige grafische Benutzer Schnittstelle um den Programmen ein betriebssystemtypisches Aussehen und Verhalten zu verleihen.

Die Entwicklung der Commodore Amiga Version wurde 2002 gestoppt, nachdem sich herauskristallisierte, das immer mehr Anwender vom Amiga auf den PC umstiegen, da sie akzeptierten (von den Hardcore Benutzern einmal abgesehen) das es sich um eine tote Plattform handelt. Die Windows, Linux und Mac OS Versionen werden bis heute unermüdlich weiterentwickelt und unterstützt!

Version 4 ist der letzte Stand im Entwicklungszweig von PureBasic. Für diese Version wurde fast alles von Grund auf neu geschrieben. Dies war nötig, um spätere Erweiterungen leichter integrieren zu können, sowie die 'Cross Platform' Entwicklung zu vereinfachen. PureBasic 4 brachte außerdem eine riesige Menge an Sprachverbesserungen, die fast alle hier im Buch behandelt werden.

Die Entwicklungsphilosophie von PureBasic

Die Entwicklungsphilosophie von PureBasic unterscheidet sich in mancher Beziehung ein wenig von anderen Programmiersprachen. Hier ist eine Liste der Entwicklungsziele und der Vertriebspolitik von PureBasic.

Nach dem Erwerb einer PureBasic Lizenz sind alle folgenden Updates (auf Lebenszeit) kostenlos.
Alle mit PureBasic entwickelten Programme können, ohne zusätzliche Lizenzgebühren, frei oder kommerziell vertrieben werden.

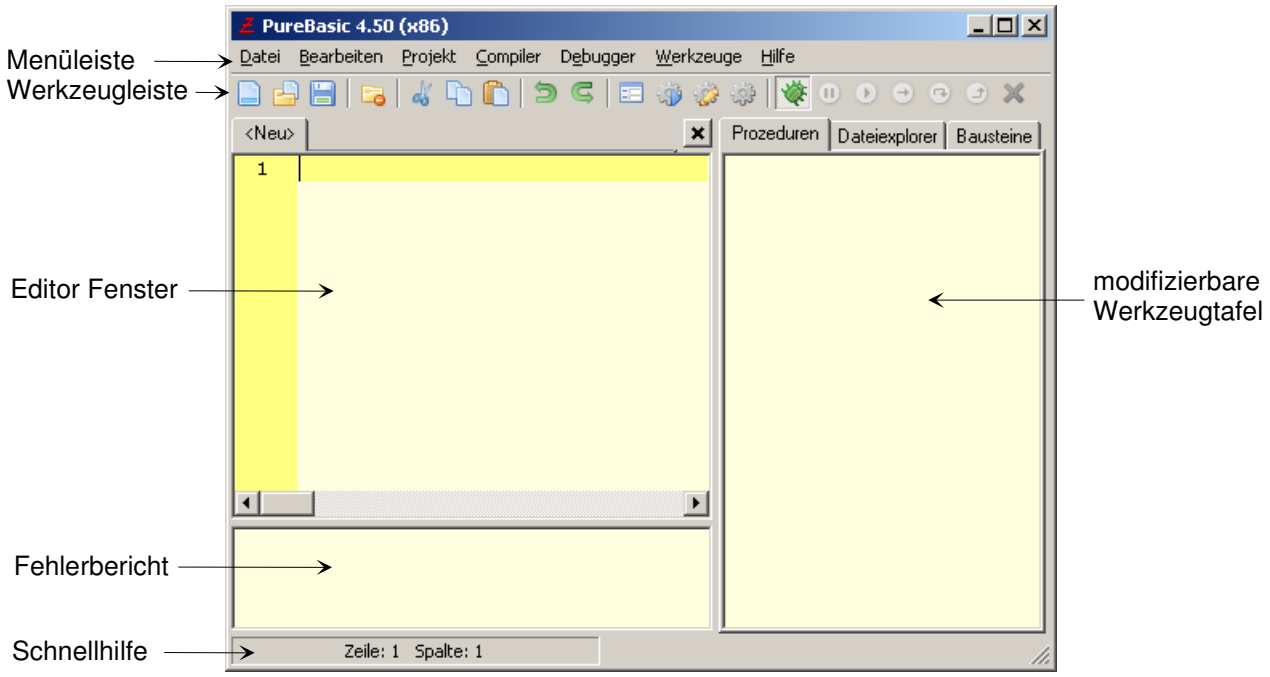
Alle Programme werden auf eine minimale Dateigröße kompiliert und enthalten keinen aufgeblasenen Code. Die erstellten Programme sind von keiner Runtime Bibliothek abhängig, sie sind eigenständig lauffähig.

Die obigen Punkte sind gute Verkaufsargumente und stehen im Gegensatz zu mancher Entwicklungsphilosophie von konkurrierenden Unternehmen. Können Sie Microsoft überzeugen, Ihnen lebenslang kostenlose Updates für VB.NET zur Verfügung zu stellen? Ich nicht.

PureBasic's Entwicklungsphilosophie ist eine Programmierumgebung bereitzustellen, die Spaß macht und funktionell ist. Sie gibt dem Benutzer ein mächtiges Instrument an die Hand, mit dem er die Programme die er braucht, auf die einfachst mögliche Weise erstellen kann. Mit jedem vergangenen Update wurden jede Menge Fehler behoben, neue Funktionen hinzugefügt, sowie eine neue IDE und ein Visual Designer integriert. Die beiden letztgenannten werden in späteren Kapiteln im Buch besprochen. Updates bestehen also nicht nur aus Fehlerbereinigungen, sondern enthalten auch Befehlserweiterungen der Sprache und nützliche Werkzeuge für die Entwicklungsumgebung.

Ein erster Blick auf die Entwicklungsumgebung

Die integrierte Entwicklungsumgebung von PureBasic setzt sich aus dem Quelltext Editor, dem Visual Designer und dem Compiler zusammen. Der Visual Designer wird in Kapitel 9 besprochen, hier werden wir uns zunächst nur mit dem Editor und dem Compiler befassen. In der PureBasic Gemeinde wird der Quelltext Editor im allgemeinen mit 'IDE' und der Visual Designer mit 'VD' abgekürzt, damit im Forum bei Fragen nicht ständig lange Namen geschrieben werden müssen. Da sich diese Bezeichnungen etabliert haben, schließe ich mich hier im Buch dieser verbreiteten Schreibweise an.



Die Windows IDE Abb. 1

Die IDE (Abb. 1) ist selbst komplett in PureBasic programmiert worden. Sie ist das Hauptwerkzeug zum erstellen von PureBasic Programmen. Im folgenden werden wir nun einen Blick auf die Benutzeroberfläche werfen. Ganz oben befindet sich die Menüleiste, die Zugriff auf die einzelnen Funktionen der IDE gewährt. Darunter sehen wir die Werkzeugleiste, die individuell mit Funktionen aus der Menüleiste bestückt werden kann. Links unterhalb der Werkzeugleiste befindet sich das Editorfenster, in das alle Quelltexte eingegeben werden. Rechts vom Editor befindet sich eine modifizierbare Werkzeugtafel in der verschiedene Werkzeuge, wie z.B. Prozedurliste, Variablenliste, Dateiexplorer, usw. verankert werden können. Standardmäßig wird unterhalb des Editorfensters der Fehlerbericht angezeigt. Dieser kann via Menübefehl an- oder ausgeschaltet werden (Menü: Debugger → Fehlerbericht → Fehlerbericht zeigen). Am unteren Ende der IDE befindet sich die Statusleiste. Hier werden die aktuellen Koordinaten des Cursors im Editorfenster sowie, bei Verfügbarkeit, eine Schnellhilfe zum aktuell markierten Befehl angezeigt.

Die IDE ist die Benutzerschnittstelle zum eigentlichen PureBasic Compiler. Nachdem Sie einen Quelltext in den Editor eingegeben habe und in der Werkzeugleiste den Knopf 'Kompilieren/Starten' gedrückt haben (Tastenkürzel: F5), wird der Code zum Compiler weitergeleitet. Dieser erstellt aus dem Quelltext ein

ausführbares Programm. Prinzipiell kann zum bearbeiten des Quellcodes jeder beliebige Editor verwendet werden. Ich empfehle allerdings den offiziellen Editor zu verwenden, da dieser von Grund auf nur für die Unterstützung des PureBasic Compilers konzipiert wurde. Andere Editoren müssen häufig umkonfiguriert werden, damit sie den Quelltext in korrekter Form an den Compiler übermitteln, dies überfordert die meisten Anfänger.

Die IDE Schnellhilfe

Wenn Sie im Editor einen Befehl aus den mitgelieferten Bibliotheken eingeben, wird dieser vervollständigt mit Beispielen für gegebenenfalls erforderliche Parameter in der Statusleiste angezeigt. Durch Anzeige dieser Informationen lässt sich mancher Blick in die Hilfe sparen, da auf einen Blick ersichtlich wird, welche Parameter ein Befehl benötigt. Mitgelieferte Befehle werden in Kapitel 7 besprochen.

Wie werden PureBasic Programme gestartet

Nun werden wir lernen wie Programme gestartet werden. Bevor PureBasic Programme kompiliert sind handelt es sich um reine Text Dateien (standardmäßig mit der Dateiendung *.pb), die den Quellcode enthalten. Um daraus ein ausführbares Programm zu erstellen muss die Datei einfach zur Verarbeitung an den Compiler weitergereicht werden. Hierfür gibt es mehrere Möglichkeiten, z.B.:

in der IDE:

Drücken Sie das Tastenkürzel F5 für 'Kompilieren/Starten'.
Drücken Sie den 'Kompilieren/Starten' Knopf in der Werkzeugleiste.
Wählen Sie den Menübefehl: Menü: 'Compiler → Kompilieren/Starten'.
Wählen Sie den Menübefehl: Menü: 'Compiler → Executable erstellen...'.

Benutzen der Kommandozeile:

Geben Sie auf der Kommandozeile 'PBCompiler Dateiname' ein, wobei 'Dateiname' für den Namen der Textdatei steht.

Wie es scheint, gibt es viele verschiedene Möglichkeiten, zum selben Ergebnis zu kommen. Jedoch sind einige Ergebnisse geringfügig anders und sollten hier genauer erklärt werden.

Die ersten drei IDE Methoden führen alle zum gleichen Ergebnis und können während des Schreibens zum testen des Programms verwendet werden (es macht keinen Unterschied, welche Methode Sie anwenden). Der Zweck der drei Methoden ist das Programm zu kompilieren und zu starten.

Nach Anwahl der Methode wird die Textdatei unverzüglich kompiliert, und es wird im Ordner '/PureBasic/Compilers' eine temporäre ausführbare Datei mit dem Namen 'PureBasic0.exe' erstellt, die im Anschluss gestartet wird. Diese Vorgehensweise ist praktisch, wenn Sie sofort sehen möchten, wie Ihr Programm funktioniert, ohne erst einen Namen für Ihr Programm vergeben zu müssen. Wenn bereits ein Programm mit diesem Namen läuft, wird ein neues Executable mit der nächstgrößeren Nummer erstellt (in diesem Fall 'PureBasic1.exe'). Wenn zwischenzeitlich bereits laufende Programme beendet werden, sind die entsprechenden Nummern wieder verfügbar und werden bei erneuten Programmstarts wieder mit berücksichtigt. Auf diese Weise werden Programme mit sehr hohen Nummern vermieden.

Die letzte IDE Methode 'Compiler → Executable erstellen...' wird in der Regel benutzt wenn der Entwicklungsprozess abgeschlossen ist und das Programm eingesetzt werden soll. Wenn Sie diese Methode zum ersten mal benutzen, erscheint ein Dialogfenster, in das Sie den Namen für das Programm eingeben müssen.

Die Kommandozeilen Methode eignet sich mehr für fortgeschrittene Benutzer. Sie ermöglicht es Ihnen den Quelltext direkt als Parameter an den Compiler zu übergeben. Durch Übergabe weiterer Parameter haben Sie die Möglichkeit den Kompilierungsprozess individuell zu steuern. Diese weiteren Parameter werden in Kapitel 13 besprochen.

Nun wissen Sie alles, was Sie zum starten Ihres ersten Programms benötigen: Nachdem Sie ihre Anweisungen in den Editor eingegeben haben, drücken Sie einfach F5 und Sie sind durch.

Der Debugger

Der PureBasic Debugger ist ein sehr wertvolles Werkzeug. Er überwacht die Ausführung Ihres laufenden Programms und behält dabei die Kontrolle über alle Variablen, Prozedurparameter und vieles weiteres. Tritt während der Programmausführung ein Fehler auf, markiert er die entsprechende Zeile in Ihrem Programmcode und gibt zusätzlich einen Kommentar zum aufgetretenen Fehler aus. Eine weitere Stärke ist die Vermeidung von potentiellen Programmabstürzen wie Division durch Null, ungültige Zugriffe über Arraygrenzen hinweg oder Datenüberlauf Fehler. Zusätzlich haben Sie die Möglichkeit Ihr Programm jederzeit anzuhalten und die momentanen Werte von Variablen zu beobachten. Danach können Sie Ihr Programm auch im Einzelschrittmodus fortsetzen um Fehler oder die Quelle ungewöhnlicher Verhaltensweisen zu lokalisieren. Beim aufspüren von Endlosschleifen hat diese Funktion einen unschätzbaren Wert.

Der Debugger kann jederzeit ein- und ausgeschaltet werden, indem Sie auf den 'Debugger verwenden' Knopf in der Werkzeugleiste klicken oder den Menübefehl 'Menü: Debugger → Debugger verwenden' anwählen. Größte Vorsicht ist geboten, wenn Sie in der Entwicklungsphase ein Programm mit ausgeschaltetem Debugger starten; ein nicht erkannter gravierender Fehler kann zum Absturz des kompletten Systems führen.

Hinweise zur Programmstruktur

Die Struktur eines PureBasic Programms ist ganz einfach zu verstehen. Der Compiler arbeitet den Quelltext von oben nach unten durch. Die oberen Befehle werden vor den darunterliegenden verarbeitet. Im Prinzip verarbeitet der Compiler den Text so, wie Sie ihn lesen. Erkennt der Debugger ein Problem, stoppt er den Kompilierungsprozess und gibt eine Fehlermeldung aus. Lassen Sie uns einen Blick auf folgenden (nicht mit PureBasic lauffähigen) Pseudo-Code werfen:

```
1 PRINT "ERSTE  AUSGEFÜHRTE ZEILE"
2 PRINT "ZWEITE AUSGEFÜHRTE ZEILE"
3 PRINT "DRITTE AUSGEFÜHRTE ZEILE"
```

Die Ausgabe diese Programms wären drei Textzeilen und zwar genau in der Reihenfolge, in der sie im Quelltext stehen. Es ist wichtig, das Sie diesen Umstand verstehen und beachten, da Sie sich sonst der Gefahr von Fehlern aussetzen (wenn Sie z.B. versuchen auf eine Datei zuzugreifen die noch nicht geöffnet ist). Das hört sich alles nach 'immer geradeaus' an, aber Sie werden noch auf ein paar Besonderheiten stoßen, wenn Sie mit Prozeduren programmieren (diese werden in Kapitel 6 besprochen). Eine Programmstruktur ist mehr als dieser kleine Codeschnipsel vermitteln kann. Das wir Ihnen nach und nach bewusst werden, wenn ich im Fortgang dieses Buches meine Beispiele mit Anweisungen und Prozeduren erweitere.

Einführung in die PureBasic Hilfe

Mit jeder PureBasic Installation wird ein komplettes Hilfesystem installiert. Diese Hilfe ist eine fantastische Referenz zur PureBasic Programmiersprache. Auf neue Benutzer wirkt sie zumeist etwas abschreckend, da einige Dinge nicht vollständig erklärt werden, andere hingegen sind so ausführlich, dass sie nicht mehr druckfreundlich sind. Trotzdem ist sie ein unschätzbares Hilfsmittel um nach Befehlen zu suchen oder die Syntax zu überprüfen, zudem ist sie nahtlos in die IDE integriert. Tatsache ist, dass ich jedes Mal, wenn ich mit der PureBasic IDE ein Programm entwickle, die Hilfe im Hintergrund geöffnet habe, um schnell zwischen beiden umschalten zu können. Diese Angewohnheit kann Stunden wertvoller Zeit sparen.

Integration in die IDE

Jederzeit wenn Sie die IDE benutzen, um an Ihrem Programm zu arbeiten, könne Sie die 'F1' Taste drücken um die Hilfe aufzurufen. Wenn sich der Cursor in der IDE auf einem PureBasic Schlüsselwort befindet, wird die Hilfe in den Vordergrund gebracht und zeigt zu diesem eine Beschreibung (häufig mit einem Beispiel) an. Diese Integration von IDE und Hilfe ist unbezahlbar, wenn Sie ihre Programmiergeschwindigkeit steigern.

Lassen Sie uns ein kleines Beispiel ausprobieren um die Hilfe in Aktion zu erleben. Tippen Sie folgende Zeilen exakt so in Ihre IDE:

```
OpenConsole()
Print("Druecken Sie Enter zum beenden")
Input()
End
```

Nach dem starten dieses kleinen Programms öffnet sich ein Konsolen Fenster, in dem eine Textzeile den Benutzer darüber informiert, das er zum beenden des Programms die 'Enter' Taste drücken muss. Bis dies geschehen ist wartet das Programm auf eine Eingabe. Nach drücken der 'Enter' Taste wird das Programm geschlossen.

Wenn Sie nun den blinkenden Cursor in der IDE auf ein Schlüsselwort bewegen und die 'F1' Taste drücken, öffnet sich im Vordergrund die Hilfe und springt auf die Seite, die diesen Befehl beschreibt. Platzieren Sie den Cursor beispielsweise irgendwo innerhalb des 'OpenConsole()' Schlüsselwortes und drücken 'F1'. Wie durch Zauberei erscheint die Hilfeseite des 'OpenConsole()' Befehls.

Wenn Sie mehr über die eingebaute Hilfe erfahren möchten, springen Sie zu Kapitel 7.

2. Datentypen

Nachdem die Einführungen jetzt durch sind, werden wir uns jetzt einem Kapitel mit mehr Gewicht zuwenden, nämlich den Datentypen. Sie dürften wissen, das Sie in Computer Programmen Prozessdaten manipulieren. Datentypen sind die Beschreibungen für die Behälter dieser Daten. In diesem Kapitel werde ich Ihnen alle verfügbaren Datentypen vorstellen und Ihnen genau erklären wie und wann sie verwendet werden.

Um Ihnen die Möglichkeit zum schnellen Einstieg zu geben, habe ich jede Menge Beispiele hinzugefügt und ich werde alles in einfacher Sprache erklären.

Eingebaute Datentypen

Datentypen (oder einfach auch 'Typen' genannt) können als eine Möglichkeit zur Beschreibung von gespeicherten Daten betrachtet werden. Die Initiale Idee Daten bestimmten Typen zuzuordnen war der Ansammlung von Binären Ziffern eine Bedeutung zu geben. Ob Text oder Zahlen, die Beschreibung dieser Daten mit Typen macht es einfacher diese zu verstehen, zu manipulieren oder abzurufen. Daten werden im RAM des Computers gehalten, bis sie vom Programm benötigt werden. Die Menge an Speicher die für jeden Datentyp benötigt wird ist abhängig von der Art des Datentyps.

Zahlen

Die ersten Datentypen die ich anführe sind die numerischen Typen. Zahlen können für alles Mögliche benutzt werden. Vom Datum über Längen bis zum Ergebnis einer langen Berechnung ist alles möglich. Überall wo Sie in der reellen Welt Zahlen verwenden, können Sie in PureBasic die numerischen Typen verwenden um diese Daten zu speichern.

In PureBasic gibt es zwei Varianten von Zahlen: Ganzzahlen (Integer) und Fließkommazahlen (Floating Point). Integer haben keinen Dezimalpunkt und können positive oder negative Werte haben. Hier einige Beispiele für Integer:

```
16543     -1951434     100     -1066     0
```

Floating Point Zahlen (oder auch 'Floats') auf der anderen Seite sind Zahlen mit einem Dezimalpunkt und können ebenfalls positiv oder negativ sein. Hier einige Beispiele für Floats:

```
52.887     -11.0005     1668468.1     -0.000004     0.0
```

PureBasic stellt zehn numerischen Datentypen bereit, die Sie in Ihren Programmen verwenden können. Jeder benötigt unterschiedlich viel Platz im Speicher und hat unterschiedliche numerische Grenzen. Die numerischen Typen werden in Abb. 2 beschrieben.

PureBasic's Numerische Typen

Name	Suffix	Speicherverbrauch	Bereich
Byte	.b	1 Byte (8 Bit)	-128 bis +127
Ascii	.a	1 Byte (8 Bit)	0 bis +255
Character	.c	1 Byte (8 Bit) (Ascii)	0 bis +255
Word	.w	2 Byte (16 Bit)	-32768 bis +32767
Unicode	.u	2 Byte (16 Bit)	0 bis +65535
Character	.c	2 Byte (16 Bit) (Unicode)	0 bis +65535
Long	.l	4 Byte (32 Bit)	-2147483648 bis +2147483647
Integer	.i	4 Byte (32 Bit)*	-2147483648 bis +2147483647
Integer	.i	8 Byte (64 Bit)*	-9223372036854775808 bis +9223372036854775807
Quad	.q	8 Byte (64 Bit)	-9223372036854775808 bis +9223372036854775807
Float	.f	4 Byte (32 Bit)	unlimitiert**
Double	.d	8 Byte (64 Bit)	unlimitiert**

* Größe abhängig vom Betriebssystem (32/64 Bit), ** Erklärung in Kapitel 13 (numerische Datentypen) Abb. 2

In Abb. 2 können Sie sehen, das viele Typen ein numerisches Limit haben. Dieses Limit ist direkt an den Speicherverbrauch des jeweiligen Typs gekoppelt. Der Speicherverbrauch und die Namen der numerischen Typen sind weitestgehend mit den Typen der Programmiersprache C identisch. In C werden Sie noch eine Menge mehr Typen als die hier aufgelisteten finden aber PureBasic versucht die Dinge einfach zu halten und Sie nicht mit Hunderten fortgeschrittenen Typen zu überfrachten. Für Anfänger ist es eigentlich nur wichtig zu wissen, welche numerischen Limits die Typen haben und darauf zu achten, das diese nicht überschritten werden. Wie die entsprechenden Zahlen im RAM gespeichert werden, erkläre ich in Kapitel 13.
Wenn ein numerisches Limit überschritten wird springt der Wert auf den niedrigeren Level dieses Typs. Wenn Sie zum Beispiel den Wert '129' an eine Variable vom Typ Byte übergeben, dann überschreitet dieser Wert das numerische Limit dieses Typs und springt auf '-127'.

Zeichenketten

Der letzte Standard PureBasic Datentyp ist die Zeichenkette (String). Strings sind so wichtige und nützliche Datentypen, das sie in nahezu jeder Programmiersprache verfügbar sind.
Wie ihr Name schon sagt, sind Zeichenketten eine Aneinanderreihung von Zeichen. Anders als bei Zahlen, ist eine ganz bestimmte Schreibweise notwendig, damit Strings als solche erkannt werden. In diesem Fall bedeutet das, der String muss in Anführungszeichen geschrieben werden. Hier einige Beispiele für diese Syntax:

```
"abcdefghijklmnopqrstuvwxyz"   "Alle meine Entchen"  "123456789"
```

Beachten Sie, das der letzte der drei Strings aus Zahlen besteht. Da er in Anführungszeichen gekapselt ist, wird er als solcher und nicht als Zahl erkannt. Strings wie dieser werden auch als literale Strings bezeichnet.

Strings sind wahrscheinlich die einfachsten Datentypen, weil sie so einfach zu benutzen sind. Immer wenn Sie die Anführungszeichen um Zeichen bemerken, wissen Sie, das es sich um einen String handelt.

PureBasic's String Typen

Name	Suffix	Speicherverbrauch	Bereich
String	.s	Länge des Strings + 1 Byte	unlimitiert
String	$	Länge des Strings + 1 Byte	unlimitiert
Fester String	.s{Länge}	Länge des Strings	benutzerdefiniert*
Fester String	${Länge}	Länge des Strings	benutzerdefiniert*

* der Länge Parameter definiert die maximale Länge des Strings Abb. 3

Strings können aus jedem Zeichen des ASCII Zeichensatzes bestehen, inklusive der Steuerzeichen (siehe Anhang B für eine volle Übersicht der ASCII Zeichen) aber nicht dem 'NULL' Zeichen, welches zur Kennzeichnung des Stringendes verwendet wird. Strings die ihr Ende mit dem 'NULL' Zeichen markieren nennt man 'Null terminierte Strings'.

Variablen und Konstanten

Zum speichern und manipulieren von Daten brauchen Sie die richtigen Datentypen aber Sie brauchen auch eine Möglichkeit, einfach auf diese Daten im Speicher zuzugreifen. Variablen und Konstanten sind die Antwort auf dieses Problem. Sie verknüpfen ein bestimmtes Stück Daten im Speicher mit einem klar lesbaren Namen. Dies vereinfacht den späteren Zugriff auf diese Daten. Beachten Sie, Variablen verweisen auf Daten die verändert werden können, während Konstanten auf Daten verweisen, die nicht verändert werden können.

Variablen

Typischerweise ist der Name einer Variable mit einem kleinen Stück Speicher im RAM verbunden (Größe abhängig vom Datentyp) und alle Operationen an der Variable manipulieren diesen kompletten Speicherbereich. Variablen können Sie benennen wie Sie wollen, die meisten Menschen bevorzugen allerdings eine Benennung, die einer Beschreibung der enthaltenen Daten entspricht. Variablen sind die Bausteine eines jeden Computer Programms, weil sie Daten enthalten die man manipulieren, abrufen und anzeigen kann. Variablen sind essentiell für die Organisation und Speicherung Ihrer Daten.

Na gut, lassen Sie uns ein wenig mit PureBasic spielen. Öffnen Sie die PureBasic IDE und lassen Sie uns eine eigene Variable erstellen. Die Syntax zum erstellen einer Variable ist sehr einfach. Sie geben einen

Variablennamen mit entsprechender Erweiterung zur Typdefinition, gefolgt von einer Operation die Sie ausführen möchten, ein. Diese Operation ist eine anfängliche Wertzuweisung.

In der folgenden Anweisung werden wir dem Namen 'AnzahlCodeZeilen' den Wert '1' mittels des Gleich-Operators (=) zuweisen. Als Datentyp verwenden wir Byte.

```
AnzahlCodeZeilen.b = 1
```

Schauen Sie etwas genauer auf diese Anweisung. Sie werden feststellen, das der Variablenname keine Leerzeichen enthält; das ist sehr wichtig. Variablen dürfen keine Leerzeichen enthalten! Wenn Sie zur besseren Lesbarkeit einige Wörter separieren wollen, können Sie Unterstriche verwenden, wie hier:

```
Anzahl_Code_Zeilen.b = 1
```

Sie können jeden Namen für eine Variable benutzen, dabei sind aber einige Regeln zu beachten. Eine Variable darf nicht mit einer Zahl beginnen und darf keine Operatoren enthalten (siehe Abb. 15 für eine komplette Liste der Operatoren). Des Weiteren sind keine Sonderzeichen erlaubt wie z.B. akzentuierte Zeichen (ß, ä, ö, ü). Das '.b' am Ende des Variablennamens ist ein Suffix, das dem Compiler mitteilt, dass es sich um eine Byte Variable handelt. Als solche wird für sie der entsprechende Speicherplatz reserviert und sie ist an die entsprechenden numerischen Limits gebunden. Abb. 2 zeigt alle Suffixe, die Sie für numerische Variablen verwenden können, während Abb. 3 die notwendigen Suffixe für Strings auflistet. Wenn wie hier kein Suffix verwendet wird:

```
AnzahlCodeZeilen = 1
```

dann wird die Variable automatisch als Integer deklariert - das ist der PureBasic Standard Typ. Dies ist wichtig zu verstehen, denn wenn Sie das Suffix bei einer Variable vergessen, erstellen Sie eine Variable vom Typ Integer, was eventuell Fehler verursachen kann. PureBasic bietet die Möglichkeit den Standardtyp für neue Variablen mittels des 'Define' Schlüsselwortes zu ändern:

```
Define.b
AnzahlCodeZeilen = 1
HeutigesDatum    = 18
```

Das 'Define' Schlüsselwort bekommt ein eigenes Suffix und alle Variablen die danach deklariert werden sind von diesem neuen Typ. Die beiden Variablen in obigem Beispiel sind beide vom Typ Byte, aufgrund des '.b' Suffixes hinter dem 'Define' Schlüsselwort. Wird dieses Schlüsselwort in einem PureBasic Programm nicht verwendet, ist der Standardtyp Integer.

Wenn Sie einige Variablen zur späteren Verwendung erstellen, diesen aber noch keinen Wert zuweisen wollen, können Sie folgende Syntax verwenden.

```
Define.w Tag, Monat, Jahr
```

Dieser Code schaltet den Standardtyp auf 'Word' und deklariert drei Variablen vom Typ 'Word' (Tag, Monat, Jahr). Weil den Variablen kein Wert zugewiesen wurde, erhalten diese automatisch den Wert Null (0). Hier ein Beispiel für die Erstellung von allen Variablentypen mit PureBasic Code:

```
ByteVariable.b          = 123
AsciiVariable.a         = 255
CharVariable.c          = 222
WordVariable.w          = 4567
UnicodeVariable.u       = 1973
LongVariable.l          = 891011
IntegerVariable.i       = 78936
QuadVariable.q          = 9223372036854775807
FloatVariable.f         = 3.1415927
DoubleVariable.d        = 12.53456776674545
StringVariableEins.s    = "Test String Eins"
StringVariableZwei$     = "Test String Zwei"
StringVariableDrei.s{6} = "abcdef"
StringVariableVier${3}  = "abc"
```

Wie Sie sehen sind die letzten vier Variablen alle Strings, sie unterscheiden sich aber geringfügig in ihren Suffixen. Die ersten Zwei haben eine unlimitierte Länge, während die letzten Zwei eine fest definierte Länge haben. Jede dieser zwei Stringarten kann zwei unterschiedliche Suffixe haben. Diese zwei Suffixe sind '.s' und '$'. Beide sind identisch, in jeder Weise. Das '$' Suffix ist eine alte Schreibweise, die in vielen Basic

Dialekten auftaucht. Sie wurde beibehalten, um die Basic Puristen zu besänftigen. Beide Schreibweisen dürfen im selben Programm verwendet werden, sind aber nicht austauschbar. Diese beiden Variablen z.B. sind unterschiedlich:

```
StringVariable.s = "Test String Eins"
StringVariable$  = "Test String Zwei"
```

Auch wenn es scheint, dass beide den gleichen Namen haben, macht das unterschiedliche Suffix sie zu zwei unterschiedlichen Variablen. Sie glauben mir nicht? Nun, das kann mit dem 'Debug' Schlüsselwort überprüft werden.

```
StringVariable.s = "Test String Eins"
StringVariable$  = "Test String Zwei"
Debug StringVariable.s
Debug StringVariable$
```

In diesem Beispiel wird das 'Debug' Schlüsselwort verwendet um den Wert der beiden Variablen im Debug Ausgabefenster anzuzeigen. Tippen Sie dieses Beispiel in die IDE und Drücken den 'Kompilieren/Starten' Knopf (F5). Es erscheinen zwei Zeilen im Debug Ausgabefenster. Sie zeigen die Werte der beiden Variablen mit dem 'Debug' Befehl an. Dieses Schlüsselwort ist wahrscheinlich der meistgenutzte Befehl in der Programmiersprache PureBasic. Es kann zum prüfen von Werten und zur Anzeige anderer nützlicher Texte während der Entwicklungsphase, verwendet werden. Wenn das fertige Programm erstellt wird, werden alle 'Debug' Befehle aus dem fertigen Programm entfernt, so das ein kleines ausführbares Programm entsteht.

Der 'Debug' Befehl

Der Debug Befehl ist sehr nützlich, wenn es darum geht nützliche Texte schnell ins Debug Ausgabefenster zu bringen. Da jeder Datentyp mit diesem Befehl benutzt werden kann, ist er unverzichtbar wenn es darum geht schnell ein paar hilfreiche Zahlen, Speicheradressen, zurückgegebene Strings und/oder Ergebnisse von Berechnungen anzuzeigen.
Wenn Sie den Debugger abschalten oder ihr fertiges Programm mit 'Menü: Compiler → Executable erstellen...' erstellen, werden alle 'Debug' Befehle ignoriert und nicht mit kompiliert.

Sie sollten außerdem noch wissen, das Variablen nicht "case sensitive" sind. Das bedeutet, das ein Wechsel von Groß- und Kleinschreibung ignoriert wird. Hierbei handelt es sich um ein Standardverhalten von Basic Dialekten. Schauen Sie auf folgendes Beispiel:

```
TestVariable.s = "Test String Eins"
testvariable   = "Test String Zwei"
TeStVaRiAbLe   = "Test String Drei"
Debug tEsTvArIaBlE
```

Im Beispiel sieht es so aus als würde ich drei Werte an drei unterschiedliche Variablen übergeben. In der Realität ordne ich der selben Variable immer neue Werte zu; ich verwende nur eine andere Kombination von Groß- und Kleinschreibung. Wie Sie sehen können, ist PureBasic die Schreibweise egal, da dieses Beispiel den Text 'Test String Drei' ausgibt. Dieses Beispiel demonstriert eine weitere Eigenart von PureBasic. Wenn Sie einer Variable einen Typ zugewiesen haben, behält sie diesen bis zum Programmende bei. Sie deklarieren z.B. eine Variable vom Typ String, dann ist es von nun an nicht mehr möglich in dieser Variable einen Integer- oder Float Wert zu speichern. Lassen Sie mich das an einem Beispiel verdeutlichen:

```
StringVariable.s = "Test String Eins"
StringVariable   = 100
```

Dieses Beispiel kann nicht kompiliert werden, und wenn Sie es trotzdem versuchen, werden Sie eine höfliche Meldung von der IDE erhalten, die Ihnen mitteilt, dass Sie keinen numerischen Wert in eine Stringvariable schreiben dürfen. Das folgende Beispiel funktioniert:

```
StringVariable.s = "Test String Eins"
StringVariable   = "100"
```

Weil die Variable 'StringVariable' ursprünglich als String deklariert war, kann sie nur Strings aufnehmen. Wenn wir den Wert in Anführungszeichen einschließen funktioniert es wunderbar, da wir dann den Zahlenwert als String übergeben. Lassen Sie uns noch einmal die Hauptregeln für Variablen aufzählen:

1). Variablen dürfen keine Leerzeichen enthalten
2). Variablennamen dürfen nicht mit einer Zahl beginnen, dürfen diese aber beinhalten
3). Variablennamen dürfen keine Operatoren enthalten (siehe Abb. 15)
4). Variablennamen dürfen keine Sonderzeichen enthalten (ß, ä, ö, ü)
5). Wenn kein Suffix angegeben wird, ist die Variable vom Typ Integer
6). Wenn eine Variable einmal deklariert wurde, kann ihr Typ nicht mehr geändert werden
7). Wenn eine Variable deklariert wurde, muss im folgenden das Suffix nicht mehr angegeben werden

Konstanten

Konstanten haben eine gewisse Ähnlichkeit mit den Variablen, indem sie ein einfache Möglichkeit bieten auf Daten zuzugreifen und sie dürfen beliebige Namen haben. Hier endet dann aber auch schon die Ähnlichkeit. Konstanten werden benutzt, wenn Sie einer Information einen Namen geben wollen und Sie gleichzeitig wissen, das sich der Inhalt der Information nicht mehr ändert. Schauen Sie auf dieses Beispiel:

```
#TAGE_IM_JAHR = "365"
```

Wir wissen, das sich die Anzahl der Tage in einem Standardjahr nicht ändert, deshalb können wir für diesen Wert eine Konstante verwenden. Wenn wir versuchen ihren Wert zu ändern, bekommen wir eine Fehlermeldung zu sehen. Die IDE wird Ihnen erklären, das Sie bereits eine Konstante mit diesem Namen deklariert haben und bricht die Verarbeitung ab.

Das gute an Konstanten ist, dass sie keinen Speicherplatz verbrauchen, da sie niemals als solche kompiliert werden. Vor der Kompilierung werden sie durch ihre enthaltenen Werte ersetzt. Hier ein Beispiel:

```
#TAGE_IM_JAHR = "365"
Debug "Das Jahr hat " + #TAGE_IM_JAHR + " Tage."
```

Bevor aus diesem Beispiel ihr Programm erstellt wird, sieht es für den Compiler so aus:

```
Debug "Das Jahr hat 365 Tage."
```

Die Konstanten werden vor dem kompilieren mit ihrem zugewiesenen Wert ersetzt, in diesem Fall mit '365'. Alle Konstanten folgen den gleichen Namensregeln wie Variablen, ausgenommen die Suffixe. Konstanten benötigen keine Suffixe da, unabhängig davon welchen Typ sie zugewiesen bekommen, kein Speicher belegt wird. Alle Konstanten werden mit einem Präfix und nicht mit einem Suffix deklariert. Das Präfix ist das Raute Zeichen (#).

Aufzählungen von Konstanten

Wenn Sie einen Block von Konstanten mit aufeinander folgenden Nummern benötigen, können Sie das 'Enumeration' Schlüsselwort verwenden.

```
Enumeration
  #NULL
  #EINS
  #ZWEI
  #DREI
EndEnumeration

Debug #NULL
Debug #EINS
Debug #ZWEI
Debug #DREI
```

Im Debug Ausgabefenster können Sie sehen, das jede Konstante einen größeren Wert als die vorhergehende hat, beginnend bei '0'. Wenn Sie ihre Aufzählung mit einer anderen Zahl als '0' beginnen wollen, können Sie an das 'Enumeration' Schlüsselwort einen optionalen numerischen Parameter anhängen, wie hier:

```
Enumeration 10
  #ZEHN
  #ELF
  #ZWOELF
EndEnumeration

Debug #ZEHN
Debug #ELF
Debug #ZWOELF
```

Die Konstante '#ZEHN' hat nun den Wert 10 und die folgenden werden von diesem Wert an hochgezählt. Sie können nach dem numerischen Parameter auch noch das 'Step' Schlüsselwort verwenden um die Schrittweite beim Hochzählen zu verändern. Hier ein Beispiel:

```
Enumeration 10 Step 5
  #ZEHN
  #FUENFZEHN
  #ZWANZIG
EndEnumeration

Debug #ZEHN
Debug #FUENFZEHN
Debug #ZWANZIG
```

Nun werden die Konstanten um '5' erhöht, beginnend bei '10'.

Wenn Sie an irgendeiner Stelle in der Aufzählung einer Konstanten einen Wert zuweisen, wird die Aufzählung von diesem Wert an fortgeführt.

```
Enumeration 5
  #FUENF
  #EINHUNDERT = 100
  #EINHUNDERT_UND_EINS
  #EINHUNDERT_UND_ZWEI
EndEnumeration

Debug #FUENF
Debug #EINHUNDERT
Debug #EINHUNDERT_UND_EINS
Debug #EINHUNDERT_UND_ZWEI
```

In diesem Beispiel können Sie sehen, das nach der Zeile '#EINHUNDERT = 100' alle Konstanten vom Wert '100' an weiter hochgezählt werden.

Aufzählungen werden meistens bei der Erstellung von Grafischen Benutzeroberflächen (siehe Kapitel 9) verwendet, wo jedes Fenster oder Steuerelement seine eigene Identifikationsnummer (ID) benötigt. Aufzählungen sind eine gute Möglichkeit diese IDs bereitzustellen. Aufzählungen ersparen Ihnen den Ärger, vielen Konstanten unterschiedliche Werte zuzuweisen.

3. Operatoren

Operatoren werden benutzt um Variablen Werte zuzuweisen und die Daten, die diese beinhalten, zu manipulieren. In diesem Kapitel erkläre ich ihnen alle Operatoren die PureBasic unterstützt, und für Jeden gebe ich Ihnen ein beschreibendes Beispiel wie er funktioniert und wie man ihn einsetzt. Hier finden Sie auch viele Tabellen die Ihnen zeigen wie die fortgeschritteneren Operatoren die Daten auf Binärer Ebene manipulieren. Die Vorrangigkeit (oder Priorität) von Operatoren und wie Operatoren in PureBasic Ausdrücken verwendet werden, findet ebenfalls Beachtung.

Einführung in die Operatoren

Operatoren sind ein Set von Funktionen, die Arithmetische Operationen an numerischen Daten, Boolesche Operationen auf wahre Werte und String Operationen zum manipulieren von Zeichenketten ausführen können. Manche Operatoren sind als überladene Operatoren bekannt, das heißt sie können mit unterschiedlichen Datentypen verwendet werden und können unterschiedliche Funktionen haben. Der Gleich-Operator (=) zum Beispiel, kann dazu verwendet werden, einer Variable einen Wert zuzuweisen, als auch zwei Werte auf Gleichheit zu überprüfen.

= (Gleich)

Dieser Operator ist, wie ich denke, wahrscheinlich am einfachsten zu erklären. Er kann auf Zwei Arten benutzt werden. Erstens kann er zur Wertzuweisung einer Variable verwendet werden:

```
LongVariable.l = 1
```

Zweitens kann er benutzt werden um die Gleichheit von zwei Ausdrücken, Variablen oder Werten zu überprüfen:

```
LongVariable.l = 1

If LongVariable = 1
  Debug "Ja, LongVariable ist gleich 1"
EndIf
```

Das ist das erste Mal, das Sie dem 'If' Schlüsselwort begegnet sind, aber keine Angst. Diese Schlüsselwort ermöglicht es, in Ihrem Programm auf Bedingungen zu reagieren. In diesem Fall, wenn 'LongVariable' gleich '1' ist, gebe einen Text im Debug Ausgabefenster aus.

+ (Plus)

Der Plus Operator ist ein weiterer mehrfach nutzbarer Operator. Er wird zum Aneinanderhängen von Strings, als auch für die Addition von Zahlen benutzt. Als erstes ein Beispiel zur Addition:

```
NummerEins.i = 50
NummerZwei.i = 25
NummerDrei.i = NummerEins + NummerZwei
Debug NummerDrei
```

Die Zahl im Debug Ausgabefenster muss '75' sein, da wir die Werte der Variablen 'NummerEins' und 'NummerZwei' addiert haben (50 + 25) und das resultierende Ergebnis (75) in der Variable 'NummerDrei' abgelegt haben. Danach geben wir diesen Wert im Debug Ausgabefenster aus. Hier eine weitere Möglichkeit dies zu zeigen:

```
NummerEins.i = 50 + 25
Debug NummerEins
```

Wenn Sie den Plus Operator mit Zahlen verwenden, können Sie auch eine Kurzform benutzen, wenn Sie nur den Wert einer Variablen um einen Wert oder Ausdruck erhöhen wollen:

```
NummerEins.i = 50
NummerEins + 25
Debug NummerEins
```

Wenn eine Variable mit einem Startwert initialisiert ist, können wir mit dem Plus-Operator einen weiteren Wert addieren. Deshalb ist der Wert der nun im Debug Ausgabefenster angezeigt wird '75'.

Hier ein Beispiel zum Aneinanderhängen von Strings mit dem Plus Operator:

```
StringEins.s = "Alle meine"
StringZwei.s = " Entchen"
StringDrei.s = StringEins + StringZwei
Debug StringDrei
```

Das Wort Aneinanderhängen meint einfach anschließen oder Aneinanderketten und genau das ist es, was wir mit diesen beiden Strings machen. Wir verbinden 'StringEins' mit 'StringZwei' und speichern den Resultierenden String in 'StringDrei' danach geben wir diesen Wert im Debug Ausgabefenster aus. Hier eine andere Möglichkeit:

```
StringEins.s = "Alle meine" + " Entchen"
Debug StringEins
```

Wenn Sie weiteren Text an eine Stringvariable anhängen möchten, können Sie ebenfalls die Kurzform des Plus Operators verwenden:

```
StringEins.s = "Alle meine"
StringEins   + " Entchen"
Debug StringEins
```

Diese Kurzform arbeitet ähnlich wie die numerische, aber anstatt die Werte numerisch zu addieren, wird der zweite String an die existierende Stringvariable angehängt.

- (Minus)

Der Minus Operator ist das exakte Gegenteil vom Plus Operator, da er subtrahiert und nicht addiert. Anders als der Plus Operator kann er nicht mit Strings verwendet werden. Hier ein Beispiel mit dem Minus Operator:

```
NummerEins.i = 50
NummerZwei.i = 25
NummerDrei.l = NummerEins - NummerZwei
Debug NummerDrei
```

Der ausgegebene Text im Debug Ausgabefenster sollte '25' sein, was 'NummerZwei' subtrahiert von 'NummerEins' entspricht. Auch hier kann wieder eine Kurzform verwendet werden, wenn Sie eine Variable um einen bestimmten Wert erniedrigen wollen:

```
NummerEins.i = 50
NummerEins - 10
Debug NummerEins
```

Hier wird 'NummerEins' der Wert '50' zugewiesen, dann wird 'NummerEins' mittels des Minus Operators um '10' erniedrigt. Der neue Wert von 'NummerEins' (40) wird dann im Debug Ausgabefenster angezeigt.

*** (Multiplikation)**

Der Multiplikations Operator wird benutzt um zwei Werte miteinander zu multiplizieren und wie der Minus Operator kann er nicht mit Strings benutzt werden. Hier ein Beispiel, um zu zeigen wie der Operator benutzt wird:

```
NummerEins.i = 5
NummerZwei.i = 25
NummerDrei.l = NummerEins * NummerZwei
Debug NummerDrei
```

Die Debug Ausgabe sollte '125' sein, da wir in diesem Beispiel 'NummerEins' mit 'NummerZwei' multipliziert haben (5 * 25 = 125). Auch hier kann wieder eine Kurzform benutzt werden um eine Variable mit einem bestimmten Wert zu multiplizieren:

```
NummerEins.i = 50
NummerEins * 3
Debug NummerEins
```

Hier wird 'NummerEins' der Wert '50' zugewiesen, dann wird 'NummerEins' mit '3', unter Zuhilfenahme des Multiplikations Operators, multipliziert. Der neue Wert von 'NummerEins' (150) wird dann im Debug Ausgabefenster ausgegeben.

/ (Division)

Der Divisions Operator ist ein weiterer mathematischer Operator, der nur mit Zahlen und nicht mit Strings funktioniert. Vom lesen der anderen Beispiele werden Sie vermutlich erraten, wie er benutzt wird, aber hier trotzdem nochmal ein Beispiel für die Benutzung:

```
NummerEins.i = 100
NummerZwei.i = 2
NummerDrei.l = NummerEins / NummerZwei
Debug NummerDrei
```

Hier wird 'NummerEins' der Wert '100' zugewiesen und 'NummerZwei' bekommt den Wert '2' zugewiesen. Dann dividieren wir 'NummerEins' (100) durch 'NummerZwei' (2) und speichern das Ergebnis (50) in 'NummerDrei'. Dann geben wir den Wert von 'NummerDrei' im Debug Ausgabefenster aus. Wie bei den vorhergehenden Operatoren kann auch hier wieder die Kurzform verwendet werden um eine Variable durch einen bestimmten Wert zu teilen:

```
NummerEins.i = 50
NummerEins / 5
Debug NummerEins
```

'NummerEins' wird der Wert '50' zugewiesen, dann benutzen wir den Divisions Operator um diesen Wert durch '5' zu teilen. Dann geben wir das in 'NummerEins' gespeicherte Ergebnis (10) im Debug Ausgabefenster aus.

& (bitweises UND [AND])

Die bitweisen Operatoren sind eine Gruppe von Operatoren, die Zahlen auf der binären Ebene manipulieren. Wenn Sie mit binären Werten und dem Vorgang, wie PureBasic Zahlen binär speichert, nicht vertraut sind, können Sie in Kapitel 13 (Ein genauerer Blick auf die Numerischen Datentypen) mehr darüber erfahren. Binäre Operatoren können nicht mit Fließkommazahlen oder Zeichenketten verwendet werden.

Der bitweise '&' Operator prüft zwei Werte darauf, das beide auf binärer Ebene True sind. Wenn zwei Bits miteinander verglichen werden und beide sind True (1) dann gibt der Operator True (1) zurück, in allen anderen Fällen wird False (0) zurückgegeben. Dieser Vorgang wird auf alle Bits in den beiden Zahlen angewendet. Hier ein Diagramm, das den Vorgang besser erklärt:

Der '&' (bitweise UND) Operator

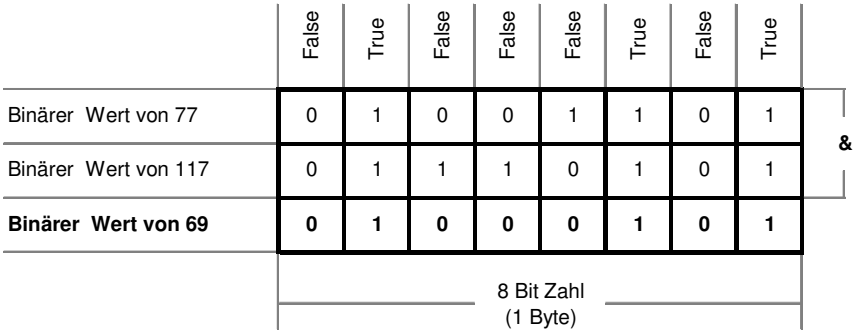


Abb. 4

In Abb. 4 sehen Sie zwei Zahlen, die mit dem '&' Operator verknüpft werden - '77' und '117'. Nach der kompletten Berechnung stellt sich ein Ergebnis von '69' ein. Um zu verstehen, wie dieser Wert zustande kommt, müssen Sie jede Spalte von oben nach unten betrachten. Wenn Sie zum Beispiel die ganz rechte Spalte betrachten (die der binären '1' entspricht), dann sehen Sie, das die Bits beider Zahlen in dieser Spalte '1' sind, weshalb das zurückgegebene Ergebnis des '&' Operators ebenfalls '1' ist (was in PureBasic True entspricht). Wenn wir eine Spalte weiter nach links gehen, können wir sehen, das die Bits beider Zahlen '0' sind, weshalb der '&' Operator '0' (False) zurückgibt. Merken Sie sich, wenn Sie den '&' Operator verwenden, müssen beide Bits '1' sein damit der Operator '1' zurückgibt. In allen anderen Fällen ist das Ergebnis '0'.

Dieser Operator wird auf alle Spalten in Abb. 4 angewandt, beginnend von rechts nach links, und wenn er damit fertig ist wird die resultierende Zahl zurückgegeben. In diesem Fall ist das Ergebnis der Berechnung '69'. Hier ein Beispiel, wie Abb. 4 als Code realisiert wird:

```
NummerEins.b = 77
NummerZwei.b = 117
NummerDrei.b = NummerEins & NummerZwei
Debug NummerDrei
```

In diesem kleinen Beispiel werden zwei Variablen Zahlen übergeben und diese dann mittels des '&' Operators verknüpft. Die Variable 'NummerDrei' enthält das Ergebnis der Berechnung. Der Wert von 'NummerDrei' wird dann im Debug Ausgabefenster angezeigt, was in diesem Fall '69' sein sollte. Wie bei den anderen Operatoren, gibt es auch für den bitweise '&' Operator eine Kurzform, wenn Sie nur eine Variable mit einer Zahl '&' verknüpfen wollen:

```
NummerEins.b = 77
NummerEins & 117
Debug NummerEins
```

Hier wird 'NummerEins' der Wert '77' zugewiesen. In der nächsten Zeile verknüpfen wir den Wert '117' mittels des '&' Operators mit 'NummerEins'. Dieser Wert wird dann im Debug Ausgabefenster angezeigt.

Abb. 5 zeigt den Vergleich zweier Bits mittels des '&' Operators und das entsprechende Ergebnis.

'&' (bitweises UND) Bit Vergleich

Linke Seite	Rechte Seite	Ergebnis
0	0	0
0	1	0
1	0	0
1	1	1

Abb. 5

| (bitweises ODER [OR])

Der bitweise '|' Operator prüft zwei Werte auf binärer Ebene darauf, ob ein oder mehr Bits True (1) sind. Wenn zwei Bits verglichen werden und eines der beiden True (1) ist, dann gibt der Operator True (1) zurück. Nur wenn beide Bits False (0) sind, gibt er False (0) zurück. Diese Operation wird mit allen Bits in den zwei Zahlen durchgeführt.

In Abb. 6 sehen Sie zwei Zahlen, die mittels des '|' Operators verknüpft werden, '54' und '102'. Nach der Berechnung stellt sich das Ergebnis '118' ein. Um zu verstehen, wie dieses Ergebnis zustande kommt, müssen Sie jede Spalte von oben nach unten betrachten. Wenn Sie die ganz rechte Spalte betrachten (die der binären '1' entspricht), sehen Sie, das beide Werte '0' sind, deshalb gibt der '|' Operator False (0) zurück. In der Spalte links daneben Sind beide Werte '1', deshalb gibt der '|' Operator True (1) zurück. Wenn wir in die fünfte Spalte von recht schauen, sehen Sie, das die erste Zahl ein auf '1' gesetztes Bit hat und die zweite Zahl ein auf '0' gesetztes Bit hat. In diesem Fall gibt der '|' Operator True (1) zurück, denn wenn mindestens eines der Bits den Wert True (1) hat, gibt der Operator den Wert True (1) zurück. Der '|' Operator gibt immer True (1) zurück, es sei denn, beide Bits sind False (0).

Der '|' (bitweise ODER) Operator

	False	True	True	True	False	True	True	False
Binärer Wert von 54	0	0	1	1	0	1	1	0
Binärer Wert von 102	0	1	1	0	0	1	1	0
Binärer Wert von 118	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**0**

8 Bit Zahl
(1 Byte)

Abb. 6

Dieser Operator wird auf alle Spalten in Abb. 6 angewandt, beginnend von rechts nach links, und wenn er damit fertig ist wird die resultierende Zahl zurückgegeben. In diesem Fall ist das Ergebnis der Berechnung '118'. Hier ein Beispiel, wie Abb. 6 als Code realisiert wird:

```
NummerEins.b = 54
NummerZwei.b = 102
NummerDrei.b = NummerEins | NummerZwei
Debug NummerDrei
```

In diesem kleinen Beispiel werden zwei Variablen Zahlen übergeben und dann mittels des '|' Operators verknüpft. Die Variable 'NummerDrei' enthält das Ergebnis der Berechnung. Der Wert von 'NummerDrei' wird dann im Debug Ausgabefenster angezeigt, was in diesem Fall '118' sein sollte. Wie bei den anderen Operatoren, gibt es auch für den bitweise '|' Operator eine Kurzform, wenn Sie nur eine Variable mit einer Zahl '|' verknüpfen wollen:

```
NummerEins.b = 54
NummerEins | 102
Debug NummerEins
```

Hier wird 'NummerEins' der Wert '54' zugewiesen. In der nächsten Zeile verknüpfen wir den Wert '102' mittels des '|' Operators mit 'NummerEins'. Dieser Wert wird dann im Debug Ausgabefenster angezeigt.

Abb. 7 zeigt den Vergleich zweier Bits mittels des '|' Operators und das entsprechende Ergebnis.

'|' (bitweises ODER) Bit Vergleich

Linke Seite	Rechte Seite	Ergebnis
0	0	0
0	1	1
1	0	1
1	1	1

Abb. 7

! (bitweises EXKLUSIV ODER [XOR])

Der bitweise '!' Operator prüft zwei Werte auf binärer Ebene darauf, ob eines der beiden Bits den Wert True (1) hat. Wenn eines der beiden Bits den Wert True (1) hat gibt der Operator True (1) zurück, in allen anderen Fällen gibt er False (0) zurück. Diese Operation wird mit allen Bits in den zwei Zahlen durchgeführt. Hier eine Tabelle, die den Vorgang etwas besser erklärt:

Der '!' (bitweise EXKLUSIV ODER) Operator

	False	True	True	False	True	True	False	False	
Binärer Wert von 38	0	0	1	0	0	1	1	0	!
Binärer Wert von 74	0	1	0	0	1	0	1	0	
Binärer Wert von 108	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	
				8 Bit Zahl (1 Byte)					

Abb. 8

In Abb. 8 können Sie zwei Zahlen sehen, die mittels des '!' Operators verknüpft werden, '38' und '74'. Nach der Berechnung stellt sich das Ergebnis '108' ein. Um zu verstehen, wie dieses Ergebnis zustande kommt, müssen Sie wieder jede Spalte von oben nach unten betrachten. Wenn Sie die ganz rechte Spalte betrachten (die der binären '1' entspricht), sehen Sie, das beide Werte '0' sind, weshalb der '!' Operator False (0) zurückgibt. Wenn wir eine Spalte weiter nach links gehen, können wir sehen, das beide Bits True (1) sind, weshalb der '!' Operator ebenfalls False (0) zurückgibt. Das ist so, weil der '!' Operator nur dann True

(1) zurückgibt, wenn nur eines der beiden Bits den Wert '1' hat. Wenn beide Bits '1' oder '0' sind gibt der '!' Operator False (0) zurück.

Dieser Operator wird auf alle Spalten in Abb. 8 angewandt, beginnend von rechts nach links, und wenn er damit fertig ist wird die resultierende Zahl zurückgegeben. In diesem Fall ist das Ergebnis der Berechnung '108'. Hier ein Beispiel, wie Abb. 8 als Code realisiert wird:

```
NummerEins.b = 38
NummerZwei.b = 74
NummerDrei.b = NummerEins ! NummerZwei
Debug NummerDrei
```

In diesem kleinen Beispiel werden zwei Variablen Zahlen übergeben und dann mittels des '!' Operators verknüpft. Die Variable 'NummerDrei' enthält das Ergebnis der Berechnung. Der Wert von 'NummerDrei' wird dann im Debug Ausgabefenster angezeigt, was in diesem Fall '108' sein sollte. Wie bei den anderen Operatoren, gibt es auch für den bitweise '!' Operator eine Kurzform, wenn Sie nur eine Variable mit einer Zahl '!' verknüpfen wollen:

```
NummerEins.b = 38
NummerEins ! 74
Debug NummerEins
```

Hier wird 'NummerEins' der Wert '38' zugewiesen. In der nächsten Zeile verknüpfen wir den Wert '74' mittels des '!' Operators mit 'NummerEins'. Dieser Wert wird dann im Debug Ausgabefenster angezeigt.

Abb. 9 zeigt den Vergleich zweier Bits mittels des '!' Operators und das entsprechende Ergebnis.

'!' (bitweises EXKLUSIV ODER) Bit Vergleich

Linke Seite	Rechte Seite	Ergebnis
0	0	0
0	1	1
1	0	1
1	1	0

Abb. 9

~ (bitweises NICHT [NOT])

Der bitweise '~' Operator ist recht einfach zu erklären, da er nichts anderes macht, als die Bits einer übergebenen Zahl oder eines Ausdrucks auf binärer Ebene zu Invertieren.

Der bitweise '~' Operator ist auch als unärer Operator bekannt, da er nur einen Wert benötigt, um ein Ergebnis zurückzuliefern. Das kann mit folgendem Code demonstriert werden:

```
NummerEins.b = 43
NummerZwei.b = ~NummerEins
Debug NummerZwei
```

Hier wird der Variable 'NummerEins' der Wert '43'. Dann erstellen wir eine Variable 'NummerZwei' und übergeben ihr den Wert von 'NummerEins', den wir mittels des '~' Operators auf binärer Ebene invertiert haben. Dieser Wert (welcher '-44' sein sollte) wird dann im Debug Ausgabefenster angezeigt.

Der '~' (bitweise NICHT) Operator

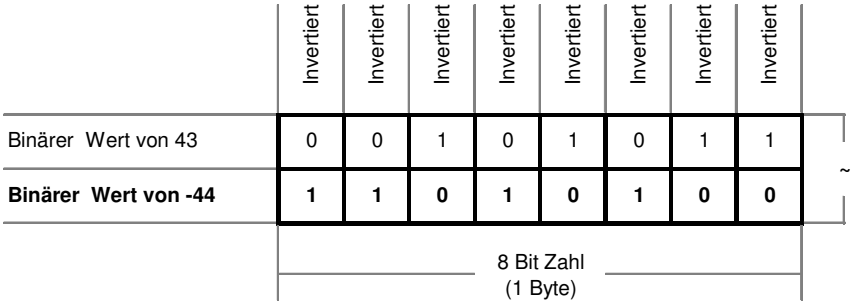


Abb. 10

In Abb. 10 können Sie sehen, das der '~' Operator einfach nur die Bits der Quellzahl invertiert und dann den neuen Wert zurückgibt. Um besser zu verstehen, wie PureBasic Zahlen (besonders die Vorzeichenbehafteten) in binärform verwaltet, sehen Sie in Kapitel 13 (Ein näherer Blick auf die Numerischen Datentypen) nach.

<< (Bit nach links schieben [shift left])

Die Bit schiebe Operatoren sind den anderen bitweise Operatoren in der Weise ähnlich, das sie Zahlen auf binärer Ebene manipulieren. Wie ihr Name schon sagt, schieben sie alle Bits nach links oder rechts, abhängig davon, welcher Operator benutzt wird. Hier ein Beispielcode, der den '<<' Operator verwendet:

```
NummerEins.b = 50
NummerZwei.b = NummerEins << 1
Debug NummerZwei
```

In diesem Beispiel weisen wir 'NummerEins' den Wert '50' zu. Dann erstellen wir eine Variable mit Namen 'NummerZwei' und weisen dieser den Wert von 'NummerEins' zu nachdem wir alle Bits dieser Variable um eine Stelle nach links verschoben haben. Den Ergebniswert (der '100' sein sollte) zeigen wir dann im Debug Ausgabefenster an. Sie werden die Funktion dieser Operation besser verstehen, wenn Sie einen Blick auf Abb. 11 werfen.

Der '<<' (Bit nach links schieben) Operator

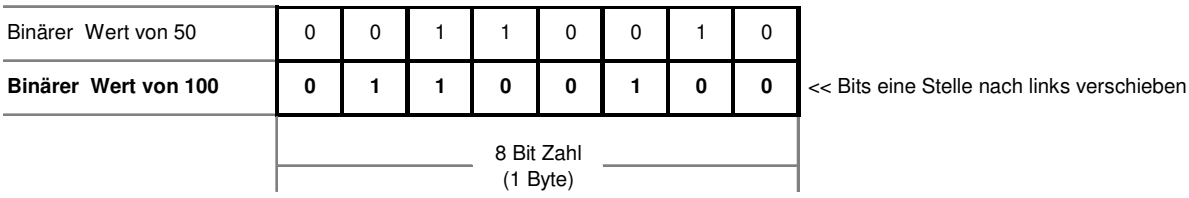


Abb. 11

Wie Sie sehen können, sind die Bits im Ergebniswert einfach nach links verschoben worden, in diesem Fall um eine Stelle. Wenn Bits nach links verschoben werden, werden die entstehenden Lücken auf der rechten Seite mit Nullen aufgefüllt, während die links "herausfallenden" Bits für immer verloren sind.

>> (Bit nach rechts schieben [shift right])

Der '>>' Operator ist exakt das gleiche wie der '<<' Operator, nur das er in die andere Richtung arbeitet. Hier ein Beispiel Code, der den '>>' Operator demonstriert:

```
NummerEins.b = 50
NummerZwei.b = NummerEins >> 1
Debug NummerZwei
```

In diesem Beispiel weisen wir 'NummerEins' den Wert '50' zu. Dann erstellen wir eine Variable mit Namen 'NummerZwei' und weisen dieser den Wert von 'NummerEins' zu nachdem wir alle Bits dieser Variable um eine Stelle nach rechts verschoben haben. Den Ergebniswert (der '25' sein sollte) zeigen wir dann im Debug Ausgabefenster an. Sie werden die Funktion dieser Operation besser verstehen, wenn Sie einen Blick auf diese Tabelle werfen:

Der '>>' (Bit nach rechts schieben) Operator

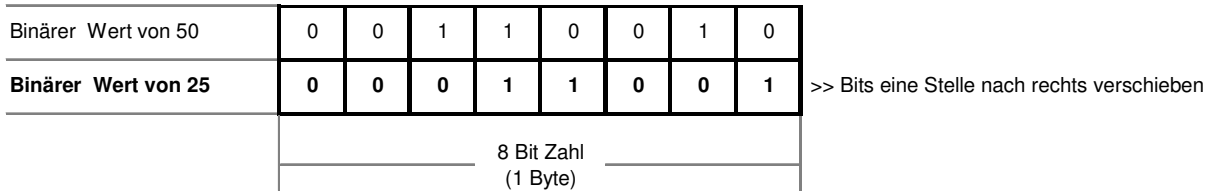


Abb. 12

Wie Sie sehen können, sind die Bits im Ergebniswert einfach nach rechts verschoben worden, in diesem Fall um eine Stelle. Wenn Bits nach rechts verschoben werden, ist es wichtig zu verstehen, welche Bits in der links entstehenden Lücke aufgefüllt werden. Wenn die Zahl eine positive Zahl ist, dann hat das ganz linke Bit (manchmal auch das 'most significant Bit' genannt) den Wert Null. In diesem Fall wird die Lücke mit Nullen aufgefüllt. Wenn die Quellzahl negativ (vorzeichenbehaftet) ist, dann hat das ganz linke Bit den Wert Eins. In diesem Fall wird die Lücke mit Einsen aufgefüllt. Die links "herausfallenden" Bits sind für immer verloren.

< (kleiner als)

Der '<' Operator wird zum vergleichen von zwei Variablen oder Ausdrücken verwendet. Wenn der Wert auf der linken Seite des Operators kleiner ist als der Wert auf der rechten Seite, gibt der Operator True (1) zurück, anderenfalls gibt er False (0) zurück. Hier ein Beispiel, das die Verwendung demonstriert:

```
NummerEins.i = 1
NummerZwei.i = 2

If NummerEins < NummerZwei
  Debug "1: NummerEins ist kleiner als NummerZwei"
Else
  Debug "2: NummerZwei ist kleiner oder gleich NummerEins"
EndIf
```

Hier in der 'If' Anweisung prüfen wir, ob 'NummerEins' kleiner als 'NummerZwei' ist, was hier auf jeden Fall zutrifft, und weshalb die erste Debug Anweisung ausgeführt wird. Was passiert, wenn wir den Wert von 'NummerEins' in '3' ändern, wie hier:

```
NummerEins.i = 3
NummerZwei.i = 2

If NummerEins < NummerZwei
  Debug "1: NummerEins ist kleiner als NummerZwei"
Else
  Debug "2: NummerZwei ist kleiner oder gleich NummerEins"
EndIf
```

Im Debug Ausgabefenster können wir sehen, das jetzt die zweite Anweisung ausgeführt wurde, da 'NummerEins' nicht mehr kleiner als 'NummerZwei' war.

> (größer als)

Der '>' Operator wird zum vergleichen von zwei Variablen oder Ausdrücken benutzt. Wenn der Wert links vom Operator größer ist, als der Wert rechts vom Operator, dann gibt der Operator True (1) zurück, anderenfalls gibt er False (0) zurück. Hier ein Beispiel, das die Verwendung demonstriert:

```
NummerEins.i = 2
NummerZwei.i = 1

If NummerEins > NummerZwei
  Debug "1: NummerEins ist größer als NummerZwei"
Else
  Debug "2: NummerZwei ist größer oder gleich NummerEins"
EndIf
```

Hier in der 'If' Anweisung prüfen wir, ob 'NummerEins' größer als 'NummerZwei' ist, was hier auf jeden Fall zutrifft, und weshalb die erste Debug Anweisung ausgeführt wird. Was passiert, wenn wir den Wert von 'NummerEins' in '0' ändern, wie hier:

```
NummerEins.i = 0
NummerZwei.i = 1

If NummerEins > NummerZwei
  Debug "1: NummerEins ist größer als NummerZwei"
Else
  Debug "2: NummerZwei ist größer oder gleich NummerEins"
EndIf
```

Im Debug Ausgabefenster können wir sehen, das jetzt die zweite Anweisung ausgeführt wurde, da 'NummerEins' nicht mehr größer als 'NummerZwei' war.

<= (kleiner oder gleich)

Der '<=' Operator wird zum vergleichen von zwei Variablen oder Ausdrücken verwendet. Wenn der Wert auf der linken Seite des Operators kleiner oder gleich dem Wert auf der rechten Seite ist, gibt der Operator True (1) zurück, anderenfalls gibt er False (0) zurück. Hier ein Beispiel, das die Verwendung demonstriert:

```
NummerEins.i = 0
NummerZwei.i = 1

If NummerEins <= NummerZwei
  Debug "1: NummerEins ist kleiner oder gleich NummerZwei"
Else
  Debug "2: NummerEins ist NICHT kleiner oder gleich NummerZwei"
EndIf
```

Hier in der 'If' Anweisung prüfen wir, ob 'NummerEins' kleiner oder gleich 'NummerZwei' ist, was hier auf jeden Fall zutrifft, und weshalb die erste Debug Anweisung ausgeführt wird. Wenn wir den Wert von 'NummerEins' in '1' ändern, wird ebenfalls die erste Debug Anweisung ausgeführt, da 'NummerEins' immer noch kleiner oder gleich 'NummerZwei' ist. Um die zweite Debug Anweisung auszuführen, müssen wir dafür sorgen, das der '<=' Operator False (0) zurückgibt. Das erreichen wir einfach indem wir uns versichern, dass 'NummerEins' NICHT kleiner oder gleich 'NummerZwei ist', wie hier:

```
NummerEins.i = 2
NummerZwei.i = 1

If NummerEins <= NummerZwei
  Debug "1: NummerEins ist kleiner oder gleich NummerZwei"
Else
  Debug "2: NummerEins ist NICHT kleiner oder gleich NummerZwei"
EndIf
```

>= (größer oder gleich)

Der '>=' Operator wird zum vergleichen von zwei Variablen oder Ausdrücken verwendet. Wenn der Wert auf der linken Seite des Operators größer oder gleich dem Wert auf der rechten Seite ist, gibt der Operator True (1) zurück, anderenfalls gibt er False (0) zurück. Hier ein Beispiel, das die Verwendung demonstriert:

```
NummerEins.i = 2
NummerZwei.i = 1

If NummerEins >= NummerZwei
  Debug "1: NummerEins ist größer oder gleich NummerZwei"
Else
  Debug "2: NummerEins ist NICHT größer oder gleich NummerZwei"
EndIf
```

Hier in der 'If' Anweisung prüfen wir, ob 'NummerEins' größer oder gleich 'NummerZwei' ist, was hier auf jeden Fall zutrifft, und weshalb die erste Debug Anweisung ausgeführt wird. Wenn wir den Wert von 'NummerEins' in '1' ändern, wird ebenfalls die erste Debug Anweisung ausgeführt, da 'NummerEins' immer noch größer oder gleich 'NummerZwei' ist. Um die zweite Debug Anweisung auszuführen, müssen wir dafür sorgen, das der '>=' Operator False (0) zurückgibt. Das erreichen wir einfach indem wir uns versichern, dass 'NummerEins' NICHT größer oder gleich 'NummerZwei ist', wie hier:

```
NummerEins.i = 0
NummerZwei.i = 1

If NummerEins >= NummerZwei
  Debug "1: NummerEins ist größer oder gleich NummerZwei"
Else
  Debug "2: NummerEins ist NICHT größer oder gleich NummerZwei"
EndIf
```

<> (ungleich)

Der '<>' Operator wird für den Vergleich von zwei Variablen oder Ausdrücken verwendet. Er funktioniert beim Vergleich (nicht bei der Zuweisung) genau umgekehrt wie der '=' Operator. Wenn der Wert links vom Operator ungleich dem Wert rechts vom Operator ist, dann gibt der Operator True (1) zurück, anderenfalls gibt er False (0) zurück. Hier ein Beispiel, das die Verwendung demonstriert:

```
NummerEins.i = 0
NummerZwei.i = 1

If NummerEins <> NummerZwei
  Debug "1: NummerEins ist ungleich NummerZwei"
Else
  Debug "2: NummerEins ist gleich NummerZwei"
EndIf
```

Hier in der 'If' Anweisung prüfen wir, ob 'NummerEins' ungleich 'NummerZwei' ist, was hier auf jeden Fall zutrifft, und weshalb die erste Debug Anweisung ausgeführt wird. Was passiert, wenn wir den Wert von 'NummerEins' auf '1' ändern:

```
NummerEins.i = 1
NummerZwei.i = 1

If NummerEins <> NummerZwei
  Debug "1: NummerEins ist ungleich NummerZwei"
Else
  Debug "2: NummerEins ist gleich NummerZwei"
EndIf
```

Wir sehen im Debug Ausgabefenster, das die zweite Debug Anweisung ausgeführt wurde, weil 'NummerEins' nun gleich 'NummerZwei' ist und der '<>' Operator False zurückgibt.

And (logisches UND)

Die logischen Operatoren werden verwendet, um die Ergebnisse der normalen Vergleichsoperatoren zu kombinieren und somit die Möglichkeit eines Vergleichs von mehreren Ausdrücken zu haben.

Der 'And' Operator wird verwendet um zwei Ausdrücke daraufhin zu überprüfen, dass beide Ergebnisse der Ausdrücke True sind. Schauen Sie auf dieses Beispiel:

```
StringEins.s = "Alle meine Entchen"
NummerEins.i = 105

If StringEins = "Alle meine Entchen" And NummerEins = 105
  Debug "1: Beide Ausdrücke geben True (1) zurück"
Else
  Debug "2: Einer oder beide Ausdrücke geben False (0) zurück"
EndIf
```

Wie wir sehen, prüft die 'If' Anweisung die String Variable 'StringEins' darauf, ob sie den Wert 'Alle meine Entchen' enthält und das die Integer Variable 'NummerEins' den Wert '105' enthält. Da dies in beiden Fällen zutrifft, gibt der 'And' Operator True zurück und die erste Debug Anweisung wird ausgeführt. Wenn einer der beiden Ausdrücke, links oder rechts vom 'And' Operator False zurückliefert, gibt auch der 'And' Operator False zurück. Der Operator ist dahingehend optimiert, dass er bereits beim Ersten Ausdruck mit dem Ergebnis False ebenfalls den Wert False zurückgibt. In diesem Fall prüft er also nur einen Ausdruck und nicht beide. Das ist wichtig zu wissen, wenn Sie zeitkritische Anwendungen Entwickeln.

Not (logisches NICHT)
Der 'Not' Operator wird benutzt, um das Ergebnis eines Ausdrucks oder Booleschen Wertes zu negieren. Mit anderen Worten bedeutet das, alles was rechts vom Operator ein True zurückgibt wird in ein False gewandelt und umgekehrt. Schauen Sie auf dieses Beispiel:

```
Eins.i = 1
Zwei.i = 2

If Not Eins = 5
  Debug "1: Eins = 5 wurde als True (1) ermittelt"
Else
  Debug "2: Eins = 5 wurde als False (0) ermittelt"
EndIf

If Not Zwei = 2
  Debug "1: Zwei = 2 wurde als True (1) ermittelt"
Else
  Debug "2: Zwei = 2 wurde als False (0) ermittelt"
EndIf
```

In der ersten 'If' Anweisung wird überprüft ob 'Eins' den Wert '5' enthält, was nicht der Fall ist. Deshalb liefert die Anweisung False zurück. Da wir aber den 'Not' Operator vorgeschaltet haben, wird dieses Ergebnis in ein True umgewandelt und als solches zurückgegeben. Das Gegenteil sehen wir bei der zweiten 'If' Anweisung. Hier ist das Ergebnis der Prüfung True aber durch den 'Not' Operator wird dieses in ein False gewandelt.

Or (logisches ODER)
Der 'Or' Operator wird zum überprüfen von zwei Ausdrücken verwendet. Es wird geprüft ob der Erste oder der Zweite ein True zurückgibt. Schauen Sie auf dieses Beispiel:

```
StringEins.s = "Alle meine Entchen"
NummerEins.i = 105

If StringEins = "Alle meine Entchen" Or NummerEins = 100
  Debug "1: Einer oder mehrere Ausdrücke geben True (1) zurück"
Else
  Debug "2: Beide Ausdrücke geben False (0) zurück"
EndIf
```

Die 'If' Anweisung prüft, das die String Variable 'StringEins' gleich 'Alle meine Entchen ist' oder dass die Integer Variable 'NummerEins' gleich '100' ist. Sie werden bemerkt haben, dass die zweite Anweisung False zurückgibt, da 'NummerEins' nicht den Wert '100' enthält.

Da eine der Anweisungen True zurückliefert, gibt auch der 'Or' Operator True zurück und die erste Debug Anweisung wird ausgeführt. Der 'Or' Operator gibt nur False zurück, wenn beide Ausdrücke (links und rechts vom Operator) False zurückliefern. Der Operator ist ebenfalls in der Weise optimiert, das er bereits nach dem ersten Ausdruck der True zurückliefert die weitere Prüfung abbricht und True zurückliefert. Das ist wichtig zu wissen, wenn Sie zeitkritische Anwendungen Entwickeln.

XOr (logisches EXKLUSIV ODER)
Der 'XOr' Operator wird zum überprüfen von zwei Ausdrücken verwendet. Es wird geprüft ob lediglich einer der beiden Ausdrücke True zurückgibt. Schauen Sie auf dieses Beispiel:

```
StringEins.s = "Alle meine Entchen"
NummerEins.i = 105

If StringEins = "Alle meine Entchen" XOr NummerEins = 105
  Debug "1: Nur ein Ausdruck gibt True (1) zurück"
Else
  Debug "2: Die Ausdrücke geben beide True(1) oder beide False (0) zurück"
EndIf
```

Die 'If' Anweisung prüft beide Ausdrücke mittels des 'XOr' Operators, um sicherzustellen, das nur ein Ausdruck True zurückgibt. Wenn beide True oder False zurückgeben, gibt die 'If' Anweisung selbst False zurück und die zweite Debug Anweisung wird ausgeführt, was im Beispiel der Fall ist. Wenn dieses Beispiel so verändert wird, das nur einer der beiden Ausdrücke True zurückgibt, dann wird die erste Debug Anweisung ausgeführt.

% (Modulo)
Der '%' Operator dividiert die Zahl auf der linken Seite durch die auf der rechten Seite und gibt dann den Teilungsrest zurück. Hier ein Beispiel:

```
NummerEins.i = 20 % 8
Debug NummerEins
```

Hier dividieren wir die Zahl '20' durch '8' mittels des '%' Operators. Da die '8' zweimal in die '20' passt, bleibt ein Teilungsrest von '4'. Dieser Teilungsrest wird der Variable 'NummerEins' zugewiesen. Deren Wert geben wir dann im Debug Ausgabefenster aus.

( ) (Klammern)
Klammern sind eigentlich keine Operatoren, da sie kein Ergebnis zurückliefern. Sie werden benutzt um die Ausführungsreihenfolge von verschachtelten Ausdrücken zu ermitteln. Generell gilt die Regel, das Ausdrücke innerhalb von Klammern zuerst verarbeitet werden. Im Fall von ineinander verschachtelten Klammern werden die inneren zuerst aufgelöst und von dort weiter nach außen bis zur Letzten. Hier ein Beispiel:

```
NummerEins.i = 2 * 5 + 3
Debug NummerEins
```

Hier ist der Wert von 'NummerEins' '13', da die Verarbeitungsreihenfolge '2 * 5' und dann '+ 3' gilt. Wenn wir Klammern hinzufügen ändert sich das:

```
NummerEins.i = 2 * (5 + 3)
Debug NummerEins
```

Hier ändert sich die Verarbeitungsreihenfolge in '5 + 3' und dann '* 2', was '16' ergibt.

Prioritäten von Operatoren

Die Priorität von Operatoren legt fest, in welcher Reihenfolge die Operatoren während des Kompilierens verarbeitet werden. In Abb. 13 sehen Sie die einzelnen Operatoren nach ihrer Priorität sortiert. In der linken Spalte sehen Sie die Prioritätsnummern. Die kleinste Zahl (1) hat die höchste Priorität, das heißt diese Operatoren werden zuerst bearbeitet und alle anderen folgen entsprechend der Nummerierung.

Prioritäten von Operatoren

Priorität*	Operatoren
1	( )
2	~
3	<< >> % !
4	| &
5	* /
6	+ -
7	> >= < <= = <>
8	And Or Not XOr

* die oberen Operatoren werden zuerst verarbeitet Abb. 13

In diesem Beispiel:

```
Debug 3 + 10 * 2
```

wird die Multiplikation vor der Addition durchgeführt, da dieser Operator eine höhere Priorität hat, auch wenn er im Code erst nach der Addition auftaucht. Das Ergebnis im Debug Ausgabefenster sollte '23' sein.

Um die Priorität von Operatoren festzulegen, können Sie Klammern verwenden, mit denen Sie Teile des Codes kapseln. Diese Teile haben dann eine höhere Priorität. Wenn wir zum Beispiel die Addition zuerst ausführen möchten, müssen wir den Ausdruck so umschreiben:

```
Debug (3 + 10) * 2
```

Nun ist der Wert im Debug Ausgabefenster '26'.

Berechnungsregeln für Ausdrücke

Wenn PureBasic Ausdrücke mit Integer- und Fließkommazahlen berechnet, wechseln manche Ausdrucks-Komponenten zweckmäßig ihren Typ. Wenn der Ausdruck einen Float Wert enthält, dann werden alle Teile des Ausdrucks nach Float konvertiert, bevor die Berechnung durchgeführt wird und das Ergebnis zurückgeliefert wird. Abb. 14 zeigt, wie PureBasic Ausdrücke unter bestimmten Bedingungen berechnet.

Wenn Sie den Eindruck haben, dass ein Ergebnis einer Berechnung einen seltsamen Wert hat oder nicht dem erwarteten Typ entspricht, ist es eine gute Idee Ihre Ausdrücke daraufhin zu Prüfen, dass der Compiler nicht folgende Regeln beachtet:

Ausdrucks- Berechnungsregeln

Beispielausdruck	Berechnungsregeln
`a.l = b.l + c.l`	'b' und 'c' bleiben beide vor und während der Berechnung vom Typ Long. Der zurückgegebene Long Wert wird dann 'a' zugewiesen.
`a.l = b.l + c.f`	Da dieser Ausdruck einen Float Wert enthält, wird 'b' vor der Berechnung in einen Float Wert gewandelt. 'b' wird dann zu 'c' addiert. Der zurückgegebene Float Wert wird dann nach Long konvertiert und 'a' zugewiesen.
`a.f = b.l + c.l`	'b' und 'c' bleiben beide vor und während der Berechnung vom Typ Long. Der zurückgegebene Long Wert wird dann nach Float konvertiert und 'a' zugewiesen.
`a.l = b.f + c.f`	'b' und 'c' bleiben beide vor und während der Berechnung vom Typ Float. Der zurückgegebene Float Wert wird dann nach Long konvertiert und 'a' zugewiesen.

Abb. 14

Operatoren Schnellübersicht

Operator	Beschreibung
=	Gleich. Dieser bietet zwei Verwendungsmöglichkeiten. Die Erste ist den Wert des Ausdrucks RVO der Variablen LVO zuzuweisen. Die zweite Möglichkeit ist die Verwendung des Ergebnisses des Operators in einem Ausdruck, um zu Prüfen ob der Wert des Ausdrucks LVO und RVO identisch ist (wenn sie identisch sind, gibt der Gleich Operator True zurück, anderenfalls gibt er False zurück).
+	Plus. Addiert den Wert des Ausdrucks RVO zum Wert des Ausdrucks LVO und gibt das Ergebnis zurück. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert des Ausdrucks RVO direkt in die Variable LVO addiert.
–	Minus. Subtrahiert den Wert des Ausdrucks RVO vom Wert des Ausdrucks LVO. Wenn sich LVO kein Ausdruck befindet, erhält der Wert des Ausdrucks RVO ein negatives Vorzeichen. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert des Ausdrucks RVO direkt aus der Variable LVO subtrahiert. (Dieser Operator kann nicht mit Strings verwendet werden).
*	Multiplikation. Multipliziert den Wert des Ausdrucks LVO mit dem Wert des Ausdrucks RVO. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert des Ausdrucks RVO direkt in der Variable LVO multipliziert. (Dieser Operator kann nicht mit Strings verwendet werden).
/	Division. Dividiert den Wert des Ausdrucks LVO durch den Wert des Ausdrucks RVO. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert in der Variable LVO direkt durch den Wert des Ausdrucks RVO dividiert. (Dieser Operator kann nicht mit Strings verwendet werden).
&	Bitweises AND. Sie sollten mit binären Zahlen vertraut sein, wenn Sie diesen Operator verwenden. Das Ergebnis dieses Operators ist der Wert des Ausdrucks LVO UND verknüpft mit dem Wert des Ausdrucks RVO. Diese Verknüpfung erfolgt Bit für Bit. Wenn das Ergebnis des Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird das Ergebnis direkt in dieser Variable gespeichert. (Dieser Operator kann nicht mit Strings verwendet werden).
|	Bitweises OR. Sie sollten mit binären Zahlen vertraut sein, wenn Sie diesen Operator verwenden. Das Ergebnis dieses Operators ist der Wert des Ausdrucks LVO ODER verknüpft mit dem Wert des Ausdrucks RVO. Diese Verknüpfung erfolgt Bit für Bit. Wenn das Ergebnis des Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird das Ergebnis direkt in dieser Variable gespeichert. (Dieser Operator kann nicht mit Strings verwendet werden).
!	Bitweises XOR. Sie sollten mit binären Zahlen vertraut sein, wenn Sie diesen Operator verwenden. Das Ergebnis dieses Operators ist der Wert des Ausdrucks LVO EXKLUSIV ODER verknüpft mit dem Wert des Ausdrucks RVO. Diese Verknüpfung erfolgt Bit für Bit. Wenn das Ergebnis des Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird das Ergebnis direkt in dieser Variable gespeichert. (Dieser Operator kann nicht mit Strings verwendet werden).
~	Bitweises NOT. Sie sollten mit binären Zahlen vertraut sein, wenn Sie diesen Operator verwenden. Das Ergebnis dieses Operators ist der NICHT verknüpfte Wert des Ausdrucks RVO, d.h. die Bits des Ergebnisses sind im Vergleich zum Wert des Ausdrucks invertiert. (Dieser Operator kann nicht mit Strings verwendet werden).
<	Kleiner als. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO kleiner ist als der Wert des Ausdrucks RVO, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
>	Größer als. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO größer ist als der Wert des Ausdrucks RVO, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
<=	Kleiner oder gleich. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO kleiner oder gleich dem Wert des Ausdrucks RVO ist, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
>=	Größer oder gleich. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO größer oder gleich dem Wert des Ausdrucks RVO ist, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
<>	Ungleich. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO gleich dem Wert des Ausdrucks RVO ist, dann gibt der Operator False zurück, anderenfalls gibt er True zurück.
And	Logisches AND. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO und der Wert des Ausdrucks RVO beide wahr sind, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
Or	Logisches OR. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO oder der Wert des Ausdrucks RVO wahr ist, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
Not	Logisches NOT. Dient zum negieren eines Booleschen Wertes. Mit anderen Worten, wenn ein Ausdruck den Wert True zurückgibt, wird dieser durch den Not Operator in False gewandelt. Wenn umgekehrt der Ausdruck RVO False zurückgibt, wird diesen in True gewandelt.
XOr	Logisches XOR. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn nur einer der Ausdrücke LVO oder RVO True zurückgibt, dann ist das Ergebnis True. Wenn beide Ausdrücke entweder True oder False sind, gibt der XOr Operator False zurück.
<<	Arithmetisches schieben nach links. Verschiebt jedes Bit im Wert des Ausdrucks LVO um die Anzahl der durch den Wert des Ausdrucks RVO definierten Stellen nach links. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert der Variable um die Anzahl der Stellen RVO geschoben. Es ist hilfreich, das Sie Binärzahlen verstanden haben wenn Sie diesen Operator verwenden. Jede verschobene Stelle wirkt wie eine Multiplikation mit dem Faktor 2.
>>	Arithmetisches schieben nach rechts. Verschiebt jedes Bit im Wert des Ausdrucks LVO um die Anzahl der durch den Wert des Ausdrucks RVO definierten Stellen nach rechts. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert der Variable um die Anzahl der Stellen RVO geschoben. Es ist hilfreich, das Sie Binärzahlen verstanden haben wenn Sie diesen Operator verwenden. Jede verschobene Stelle wirkt wie eine Division mit dem Faktor 2.
%	Modulo. Gibt den Teilungsrest der Division des Ausdrucks LVO durch den Ausdruck RVO zurück.
( )	Klammern. Sie können Klammern verwenden um einen Teil eines Ausdrucks bevorzugt oder in einer bestimmten Reihenfolge zu bearbeiten. Ausdrücke in Klammern werden vor jedem anderen Teil im Ausdruck bearbeitet. In verschachtelten Klammern werden die innersten zuerst aufgelöst und dann immer weiter nach außen.

RVO = Rechts vom Operator LVO = Links vom Operator Abb. 15

4. Bedingungen und Schleifen

In diesem Kapitel führe ich Sie in die Bedingungen und Schleifen ein. Sie sind ein sehr wichtiger Bestandteil in jedem Programm, und helfen den Programmfluss zu definieren. Ich werde mit einer Erklärung zu Booleschen Werten beginnen und erläutern wie PureBasic diese handhabt. Dann werde ich zu den Bedingungen wie 'If' und 'Select' übergehen, die dem Programm mitteilen, wie es sich bei Verzweigungen zu verhalten hat. Mit einer Erklärung der in PureBasic enthaltenen Schleifen und einigen Beispielen werde ich diese Kapitel abschließen. Wie immer erfolgt die ausführliche Erklärung anhand der Beispiele.

Boolesche Logik

Lassen Sie uns zuerst einmal die Geschichtsbücher ausgraben. George Boole war ein Mathematiker und Philosoph, der eine Form von Algebra entwickelte, welche heute als Boolesche Algebra bezeichnet wird. Die Logik hinter dieser Form der Algebra wurde zur Ehre von George Boole, "Boolesche Logik" genannt. Diese Form der Logik wurde zur Basis der modernen Computer Arithmetik. Erstaunlich ist, das George Boole diese Erfindung fast siebzig Jahre vor dem Bau des ersten Computers machte, der sie benutzte.

Zusammenfassend kann man sagen, das ganze System dreht sich um zwei Werte: 'Wahr' und 'Falsch' ('True' und 'False'). Diese zwei Werte (oder Zustände) benutzen logische Operationen um ein Ergebnis zu ermitteln. Die drei häufigsten Operatoren sind UND, ODER und NICHT (AND, OR und NOT). Diese drei Operatoren bildeten die Basis von Boole's Form der Algebra, jedoch sind es die einzigen, die für Vergleiche oder grundlegende Berechnungen benötigt werden. (Wie diese logischen Operatoren in PureBasic implementiert sind und wie sie benutzt werden, lesen Sie in Kapitel 3.)

PureBasic hat anders als z.B. C++ keinen Booleschen Datentyp (wie Sie Abb. 2 und Abb. 3 entnehmen können). In PureBasic werden Zahlen verwendet um 'Wahr' und 'Falsch' auszudrücken, '1' symbolisiert True und '0' symbolisiert False. Behalten Sie das im Hinterkopf, wenn Sie Ergebnisse auf Wahr oder Falsch überprüfen. Um Ihren Programmcode lesbarer zu gestalten, sollten Sie aber anstelle von '1' und '0' die in PureBasic eingebauten Konstanten verwenden.

Hier die beiden Konstanten:

```
#True
#False
```

'#True' hat den Wert '1' und '#False' hat den Wert '0'.

Nahezu alle Befehle in PureBasic geben einen Wert zurück. Manchmal ist es das Ergebnis einer mathematischen Funktion, ein anderes mal ist es der Status eines von Ihnen erstellten Fensters. Diese Werte werden zum überprüfen zurückgegeben. Je nach Ergebnis der Prüfung ist es möglich, unterschiedliche Aktionen zu starten. Schauen Sie auf dieses Stück Pseudo-Code:

```
wenn Fenstererstellung gleich Wahr
  dann zeichne Grafiken und Knöpfe auf das Fenster
sonst
  sage dem Benutzer das ein Problem vorhanden ist
  und beende das Programm
```

Dieser Code ist nicht ausführbar, aber er vermittelt die Logik. Ich vergewissere mich hier, ob das Fenster wirklich erstellt wurde. Wenn diese Prüfung erfolgreich ist, kann ich verschiedene Elemente auf das Fenster zeichnen. Wurde das Fenster nicht erstellt, informiere ich den Benutzer darüber, das ein Fehler aufgetreten ist und beende das Programm. Wenn ich diese Prüfung nicht vornehme, setze ich mich dem Risiko eines schweren Programmabsturzes aus, da ich eventuell versuche Elemente auf etwas nicht vorhandenes zu zeichnen.

Das war ein erster Einblick in die Prüfung von 'True' und 'False'. Dieser führt uns zum nächsten Abschnitt, in dem die 'If' Anweisung genauer erklärt wird.

Die 'If' Anweisung

Das 'If' Schlüsselwort wird zur Konstruktion von Anweisungen die den Programmfluss steuern verwendet. Es beeinflusst den weiteren Programmverlauf wenn eine bestimmte Bedingung auftaucht. Manchmal bekommt Ihr laufendes Programm ungewöhnliche Eingaben oder es treten Fehler auf. In solchen Fällen ist es praktisch, wenn Sie direkt in dieser Situation den Programmfluss entsprechend steuern, und Fehler abfangen können.

Die Konstruktion von 'If' Anweisungen

Eine 'If' Anweisung wird verwendet um einen Wert auf 'True' zu überprüfen. Wenn der Wert wahr ist, wird der Code unterhalb der der Zeile mit der 'If' Anweisung ausgeführt. Ist der Wert falsch, springt die Anweisung bis zu einem eventuell vorhandenen 'Else' Schlüsselwort weiter unten im Konstrukt und führt die darunterliegenden Anweisungen aus. Hier ein Beispiel:

```
a.i = 5

If a = 5
  Debug "Ein wahrer Wert wurde gefunden"
Else
  Debug "Es wurde kein wahrer Wert gefunden"
EndIf
```

Hier überprüft der 'If' Operator ob die Variable 'a' den Wert '5' enthält. Da dies der Fall ist, wird ein wahrer Wert zurückgegeben, so das die erste Zeile nach der 'If' Anweisung verarbeitet wird. Wäre der Rückgabewert falsch, würde der Code nach dem 'Else' Schlüsselwort ausgeführt werden. Zum beenden der 'If' Anweisung müssen Sie das 'EndIf' Schlüsselwort eingeben, welches das Konstrukt abschließt.

Alles ist wahr?

Wie Sie bereits vorher gelesen haben, interpretiert PureBasic den Wert '1' als True und den Wert '0' als False. 'If' Anweisungen verhalten sich etwas anders hinsichtlich der Interpretation von 'True'. Für sie ist jeder Wert außer '0' (Null) wahr. Bei Rückgabe von '0' ist der Wert falsch (außer Sie fragen explizit auf '0' ab). Dies ist praktisch zum überprüfen, ob eine Variable oder der Rückgabewert eines Befehls einen anderen Wert als '0' hat.

Das erste worauf Sie bei der Konstruktion von 'If' Anweisungen achten müssen ist der Ausdruck, der dem 'If' Schlüsselwort unmittelbar folgt. Dieser Ausdruck wird auf wahr überprüft. Hierbei kann es sich um eine einfache Variable sowie einen langen Ausdruck handeln. Das 'Else' Schlüsselwort ist optional und wurde hier nur verwendet um ein komplettes Beispiel zu zeigen. Wir können es komplett weglassen und unser obiges Beispiel auch folgendermaßen schreiben:

```
a.i = 5

If a = 5
  Debug "Ein wahrer Wert wurde gefunden"
EndIf
```

Die Kehrseite dieser Variante ist, dass bei Rückgabe eines 'False' Wertes keine Rückmeldung erfolgt. Es gibt keine Regel die vorschreibt, das Sie das 'Else' Schlüsselwort in einer 'If' Anweisung verwenden müssen, aber manchmal ist es sinnvoll eine False Situation der Vollständigkeit halber zu behandeln.

Schauen Sie auf folgendes Beispiel in dem wir eine Variable auf einen Wert testen:

```
Perlen.i = 5

If Perlen
  Debug "Die Variable hat einen Wert"
Else
  Debug "Die Variable hat keinen Wert"
EndIf
```

Nach dem 'If' Schlüsselwort habe ich nur eine Variable als Ausdruck zum testen verwendet. Diese Variable wird darauf überprüft, ob sie einen Wert enthält, was hier der Fall ist. Der Wert ist nicht '0' somit ist er automatisch wahr (siehe Infobox 'Alles ist wahr?') und der relevante Code wird ausgeführt. Versuchen Sie den Wert von 'Perlen' auf '0' abzuändern und starten sie das Programm erneut um das Ergebnis für einen False Wert zu sehen.

Lassen Sie uns den Blick auf einen komplizierteren Ausdruck in der 'If' Anweisung werfen. Lesen Sie noch einmal in Kapitel 3 nach, wenn Sie nicht alle verwendeten Operatoren in diesem Ausdruck verstehen.

```
Wert1.i = 10
Wert2.i = 5
Wert3.i = 1

If Wert1 >= 10 And (Wert2 / 5) = Wert3
  Debug "Der Ausdruck wurde als wahr befunden"
Else
  Debug "Der Ausdruck wurde als falsch befunden"
EndIf
```

In dieser 'If' Anweisung wird geprüft ob 'Wert1' größer oder gleich '10' ist und das 'Wert2' dividiert durch '5' gleich 'Wert3' ist. Wie Sie sehen, kann der zu prüfende Ausdruck richtig kompliziert und auch sehr spezifisch sein, welche Werte Sie testen.

Das 'ElseIf' Schlüsselwort

Ein weiteres Schlüsselwort das in einem 'If' Konstrukt auftauchen kann ist das 'ElseIf' Schlüsselwort. Wie der Name schon nahelegt, ist dieses Schlüsselwort eine Kombination von 'Else' und 'If'. Wie das 'Else' Schlüsselwort erweitert es die 'If' Anweisung um die Möglichkeit einen alternativen Code auszuführen, wenn der Originalausdruck False ergibt. Im Gegensatz zu 'Else' wird der Code unterhalb von 'ElseIf' allerdings nur ausgeführt, wenn der zu diesem Schlüsselwort gehörige Ausdruck True ergibt. Verwirrt? Hier ein Beispiel:

```
AnzahlPerlen.i = 10

If AnzahlPerlen < 5
  Debug "Die Variable hat einen Wert kleiner als '5'"
ElseIf AnzahlPerlen > 5
  Debug "Die Variable hat einen Wert größer als '5'"
Else
  Debug "Die Variable hat den Wert '5'"
EndIf
```

Hier testen wir den Wert der 'AnzahlPerlen' Variable. Das erste 'If' prüft ob der Wert kleiner als '5' ist. Weil dies nicht der Fall ist springt das Programm zur 'ElseIf' Anweisung. Diese gibt True zurück, da der Variablenwert größer als '5' ist.

Die 'ElseIf' Anweisung ist eine Großartige Möglichkeit eine 'If' Anweisung um die Prüfung mehrerer Werte zu erweitern. Die Anzahl der 'ElseIf' Einschübe ist nicht beschränkt, kann also beliebig oft erfolgen. Ein kleiner Nachteil ist, das es bei einer großen Anzahl von 'ElseIf' Anweisungen zu Problemen bei der Ermittlung ihrer Anordnung kommen kann. Bei einer großen Anzahl von Prüfungen ist manchmal eine 'Select' Anweisung zu bevorzugen.

Überspringen von Anweisungen

Wenn an irgendeiner Stelle in einem 'If' Konstrukt eine Prüfung den Wert True ergibt, wird der zu dieser Anweisung gehörige Code ausgeführt und danach der komplette Rest des Konstruktes übersprungen. Aufgrund dieses Verhaltens ist einige Vorsicht bei der Konstruktion von 'If' Anweisungen geboten.

Das letzte Beispiel hat diese überspringen bereits demonstriert. Der 'Else' Abschnitt wurde komplett ignoriert, da der 'ElseIf' Teil True zurückgegeben hat.

Die 'Select' Anweisung

Die 'Select' Anweisung ist das direkte Gegenstück zu der 'If' Anweisung, und zwar in der Weise, das sie eine weitere Möglichkeit bietet mehrere Prüfungen einer Variable oder eines Ausdrucks innerhalb eines Anweisungsblocks vorzunehmen. Obwohl die 'If' Anweisung sehr mächtig ist, ist in komplizierteren Fällen mit einer hohen Anzahl an zu prüfenden Anweisungen manchmal die 'Select' Anweisung zu bevorzugen. Lassen Sie mich Ihnen ein Beispiel in richtiger Syntax und mit Erklärung zur Benutzung zeigen.

```
Tage.i = 2

Select Tage
  Case 0
    Debug "0 Tage"
  Case 1
    Debug "1 Tage"
  Case 2
    Debug "2 Tage"
  Default
    Debug "Über 2 Tage"
EndSelect
```

Die 'Select' Anweisung beginnt mit dem 'Select' Schlüsselwort, welches an sich einen Ausdruck oder eine Variable zum testen auswählt, in diesem Fall die Variable 'Tage'. Die 'Case' Schlüsselwörter sind einzelne Äste die potentiell ausgeführt werden können, wenn der Wert der Variable 'Tage' gleich dem Ausdruck oder der Variable hinter der entsprechenden 'Case' Anweisung ist. Bezogen auf unser Beispiel heißt das, wenn die Variable 'Tage' den Wert '0' hat, wird der zugehörige Code von 'Case 0' ausgeführt. Hat die Variable den Wert '1' wird der Code von 'Case 1' ausgeführt, usw.

Sie werden bemerkt haben, das an der letzten Stelle an der ein 'Case' stehen müsste, ein neues Schlüsselwort auftaucht: 'Default'. Der zugehörige Code wird ausgeführt, wenn alle vorhergehenden 'Case' Anweisungen False zurückgemeldet haben. 'Default' ist vergleichbar mit dem 'Else' Schlüsselwort aus meinem 'If Konstrukt'.

Prüfen auf mehrerer Werte

Die 'Select' Anweisung ist in der Lage, viele verschiedene Werte zu prüfen und trotzdem bleibt der Code sauber und übersichtlich formatiert. Um die saubere Formatierung des Codes zu erleichtern bietet das 'Select' Konstrukt verschiedene Abkürzungsmöglichkeiten innerhalb der 'Case' Definitionen. Hier ein Beispiel:

```
Gewicht.i = 12

Select Gewicht
  Case 0
    Debug "Kein Gewicht"
  Case 1, 2, 3
    Debug "leicht"
  Case 4 To 15
    Debug "mittel"
  Case 16 To 30
    Debug "schwer"
  Default
    Debug "sehr schwer"
EndSelect
```

Hier sehen Sie einige Abkürzungen, die es ermöglichen den Bereich einer 'Case' Anweisung zu definieren. Benutzen Sie das 'To' Schlüsselwort um einen Wertebereich zu definieren oder verwenden Sie Kommata um verschiedene Einzelwerte mit einem 'Case' auszuwerten. Wenn Sie 'To' verwenden, muss der zweite Wert größer als der Erste sein. In diesem Beispiel habe ich Werte verwendet, diese können allerdings auch durch Ausdrücke und Variablen ersetzt werden um die Auswertemöglichkeiten präziser zu handhaben.

Hier ein weiteres Beispiel, welches 'Select' in einem Konsolenprogramm verwendet:

```
If OpenConsole()
  PrintN("1. Offizielle PureBasic Homepage")
  PrintN("2. Offizielles PureBasic Forum")
  PrintN("3. PureArea.net")
  PrintN("")
  PrintN("Geben Sie eine Zahl von 1 bis 3 ein und drücken Enter: ")
  Ziel.s = Input()
```

```
    Select Ziel
      Case "1"
        RunProgram("http://www.purebasic.com")
      Case "2"
        RunProgram("http://forums.purebasic.com")
      Case "3"
        RunProgram("http://www.purearea.net")
    EndSelect
  EndIf
  End
```

In diesem Beispiel habe ich einige neue Befehle verwendet, die Ihnen noch nicht bekannt sind aber ich denke dieses Beispiel demonstriert eine schöne Möglichkeit zur Verwendung der 'Select' Anweisung. Die neuen Befehle werden etwas später erklärt, aber ich denke Sie können das Programm aufgrund der beschreibenden Namen der Befehle gut nachvollziehen.

Das wesentliche in diesem Programm ist die Prüfung der 'Ziel' Variable durch die 'Select' Anweisung. Diese Variable bekommt einen String zugewiesen, der vom 'Input()' Befehl zurückgegeben wird, nachdem die Enter Taste gedrückt wurde. Die 'Case' Anweisungen werden dann ebenfalls mit Strings definiert, damit es zu einer Übereinstimmung kommen kann. Jeder PureBasic Typ oder ein Typ, der als Ergebnis eines Ausdrucks zurückgegeben wird, kann mit einem 'Select' Konstrukt verarbeitete werden.

Am Rande bemerkt, können Sie in diesem Beispiel auch eine 'If' Anweisung sehen, mit der ich überprüfe, dass der 'OpenConsole()' Befehl den Wert 'True' zurückgibt und somit das korrekte Öffnen eines Konsolenfensters meldet.

Schleifen

Um die Möglichkeit zu haben, Daten kontinuierlich zu empfangen und zu verarbeiten, benötigen Sie Schleifen. Alle Programme mit grafischen Benutzeroberflächen benutzen Schleifen für ihr Oberflächen-Management und die kontinuierliche Überwachung der Benutzereingaben. Die PureBasic IDE verwendet zum Beispiel viele Schleifen um Tastendrücke und Mausklicks zu überwachen, sowie die Oberfläche zu aktualisieren. Schleifen sind weiterhin eine gute Möglichkeit, bei großen Datenmengen in Arrays oder Listen, alle Elemente nacheinander zu durchlaufen.

'For' Schleifen

Die erste Schleife über die ich sprechen möchte ist wahrscheinlich die bekannteste und möglicherweise auch die meistgenutzte aller Schleifen - es ist die 'For' Schleife. Diese Schleifen, manchmal auch 'For/Next' Schleifen genannt, eignen sich besonders gut, wenn Sie eine sich automatisch erhöhende Variable als Zähler oder Index für ein bestimmtes Element in einem Array benötigen. Hier ein kleines Beispiel zum Einstieg:

```
For x.i = 1 To 10
  Debug x
Next x
```

In diesem Beispiel haben wir eine Schleife konstruiert, die das 'For' Schlüsselwort verwendet. Unmittelbar nach dem Schlüsselwort muss eine benutzerdefinierte numerische Variable stehen, in unserem Beispiel 'x'. Der 'x' zugewiesene Wert ist der Startwert der Schleife, hier habe ich den Wert '1' vorgegeben. Nach der Variablenzuweisung wird das 'To' Schlüsselwort dazu benutzt, einen Bereich zu definieren. Hier habe ich '10' als oberes, für 'x' erreichbares Limit eingegeben. Damit ist der komplette Kopf der Schleife definiert, jetzt müssen wir nur noch ein Schleifenende festlegen. Dies geschieht mit der Zeile 'Next x'. Diese letzte Zeile teilt dem Compiler mit, dass er nach jedem Schleifendurchlauf 'x' um eins erhöhen und dann wieder zum Beginn der Schleife springen soll. Wenn 'x' noch im definierten Wertebereich ist beginnt ein neuer Schleifendurchlauf.

Der Code zwischen den beiden Zeilen wird, abhängig von der Anzahl der Schritte zwischen Start- und Endwert, ständig wiederholt. Wenn 'x' das obere Limit (Wert nach 'To') erreicht hat, endet die Schleife und das Programm wird unterhalb der Schleife fortgesetzt.

Wenn Sie das obige Beispiel starten, werden Sie feststellen, dass im Debug Ausgabefenster alle Werte, die 'x' während des Schleifendurchlaufs hatte, zu sehen sind. Sie werden sehen, das die Schleife zehnmal durchlaufen wurde. In jedem Durchlauf wurde 'x' um '1' erhöht.

Hier ein weiteres Schleifenbeispiel, um einfach ein Array zu durchlaufen:

```
Dim Namen.s(4)

Namen(0) = "Andreas"
Namen(1) = "Constanze"
Namen(2) = "Tabea"
Namen(3) = "Johanna"
Namen(4) = "Manuel"

For x.i = 0 To 4
  Debug Namen(x)
Next x
```

Da auf alle Arraywerte über Indizes zugegriffen werden kann, und diese Indizes immer bei Null beginnen, sind 'For' Schleifen hervorragend geeignet um Operationen an allen Elementen in einem Array mit geringem Programmieraufwand zu realisieren. Wie Sie im letzten Beispiel sehen können, sind nur drei Zeilen Code notwendig um alle Elemente eines Arrays im Debug Ausgabefenster darzustellen, egal wie groß das Array ist. Ein größeres Array benötigt nur einen größeren Bereich, der in der ersten Zeile der 'For' Schleife definiert wird.

'For' Schleifen können ebenfalls mit Ausdrücken erstellt werden:

```
StartVar.i = 5
StoppVar.i = 10

For x = StartVar - 4 To StoppVar / 2
  Debug x
Next x
```

und, natürlich können Schleifen verschachtelt werden um mehrdimensionale Arrays zu bearbeiten:

```
Dim Zahlen.i(2, 2)

Zahlen(0, 0) = 1
Zahlen(0, 1) = 2
Zahlen(0, 2) = 3
Zahlen(1, 0) = 4
Zahlen(1, 1) = 5
Zahlen(1, 2) = 6
Zahlen(2, 0) = 7
Zahlen(2, 1) = 8
Zahlen(2, 2) = 9

For x = 0 To 2
  For y = 0 To 2
    Debug Zahlen(x, y)
  Next y
Next x
```

Solange die Zähler Variablen unterschiedliche Namen habe, können Sie beliebig viele 'For' Schleifen ineinander verschachteln. Die einzigartige Konfigurierbarkeit der 'For' Schleife macht Sie zu einer leistungsstarken und äußerst nützlichen Schleife für eine benutzerdefinierte Anzahl von Durchläufen.

Bisher haben Sie gesehen wie 'For' Schleifen den Zähler bei jedem Durchlauf um '1' erhöht haben. Dieses Hochzählen kann aber mit dem 'Step' Schlüsselwort manuell konfiguriert werden. Hier ein Beispiel:

```
For x.i = 0 To 10 Step 2
  Debug x
Next x
```

Wie Sie sehen, erscheint das 'Step' Schlüsselwort in der ersten Zeile der 'For' Schleife. Dieses Schlüsselwort kann nur in der 'For' Schleife verwendet werden, und das ist die einzige Verwendungsstelle in der Schleife. Unmittelbar nach dem 'Step' Schlüsselwort definieren Sie den Wert, um den die Variable nach jedem Schritt erhöht wird. In diesem Fall habe ich den Wert '2' benutzt, das erhöht die Variable 'x' nach jedem Schritt um den Wert '2'. Wenn Sie dieses Beispiel starten, sehen Sie, das alle Werte im Debug Ausgabefenster Vielfache von '2' sind.

'ForEach' Schleifen

Diese Art Schleifen unterscheidet sich dahingehend von anderen Schleifen, das sie nur mit Listen zusammenarbeitet. Die Syntax ähnelt sehr stark der 'For' Schleife, mit der Ausnahme, das sie keine Zählervariable benötigt. Hier ein einfaches Beispiel:

```
NewList Einkauf.s()

AddElement(Einkauf())
Einkauf() = "Bananen"

AddElement(Einkauf())
Einkauf() = "Teebeutel"

AddElement(Einkauf())
Einkauf() = "Käse"

ForEach Einkauf()
  Debug Einkauf()
Next
```

In diesem Beispiel erstelle ich eine Liste und füge dieser einige Elemente hinzu. Danach verwende ich eine 'ForEach' Schleife um alle Elemente der Liste im Debug Ausgabefenster anzuzeigen. Wie Sie sehen können, ist die Syntax klar und einfach verständlich. Die Schleife startet mit dem 'ForEach' Schlüsselwort gefolgt vom Namen der Liste. Das Ende der Schleife wird mit dem 'Next' Schlüsselwort definiert. Der Code zwischen diesen beiden Zeilen wird entsprechend der Anzahl der Listenelemente wiederholt. Wenn das Ende der Liste erreicht ist, wird die Schleife verlassen. Die 'ForEach' Schleife arbeitet mit allen Arten von Listen zusammen, auch mit strukturierten Listen. In Kapitel 5 werde ich die Listen etwas ausführlicher besprechen.

'While' Schleifen

Diese besondere Art von Schleifen benutzt einen Ausdruck, um zu bestimmen, ob sie starten soll und wie lange sie wiederholen soll. Wenn dieser Ausdruck True zurückliefert, wird die Schleife gestartet. Nach jedem Schleifendurchlauf wird der Ausdruck erneut geprüft. Wenn er wieder True zurückgibt, wird die Schleife fortgesetzt. Wenn der Ausdruck False zurückgibt, wird die Schleife verlassen. Schauen Sie auf dieses Beispiel:

```
Affen.i = 0

While Affen < 10
  Debug Affen
  Affen + 1
Wend
```

Diese Schleife ist sehr einfach zu konstruieren. Sie beginnt mit dem 'While' Schlüsselwort, dann folgt der Ausdruck zum steuern der Schleife, hier 'Affen < 10'. Die Schleife wird mit dem 'Wend' Schlüsselwort abgeschlossen. Der anfängliche Ausdruck wird darauf überprüft, ob die Variable 'Affen' kleiner als '10' ist. Wenn dies der Fall ist, startet die Schleife. Der Code innerhalb der Schleife wird solange wiederholt, bis der Ausdruck 'Affen < 10' False zurückgibt. Wenn Sie auf das Debug Ausgabefenster blicken, werden Sie feststellen, das der Ausdruck False zurückgibt wenn 'Affen' gleich '10' ist (ab da ist der Ausdruck nicht mehr kleiner als '10') und die Schleife endet.

Sie sollten beachten, das 'While' Schleifen deren anfänglicher Ausdruck False ergibt, niemals gestartet werden. Diesen Umstand demonstriert das folgende Beispiel:

```
Affen.i = 20

While Affen < 10
  Debug "Dieser Code wird niemals ausgeführt"
Wend
```

'Repeat' Schleifen

Diese Art von Schleifen ist so ziemlich das Gegenteil von 'While' Schleifen. 'Repeat' Schleifen beginnen mit dem 'Repeat' Schlüsselwort und enden mit einer von zwei Möglichkeiten. Die erste Möglichkeit ist das 'Until' Schlüsselwort in Verbindung mit einem Ausdruck. Die zweite Möglichkeit ist die Verwendung des 'ForEver' Schlüsselwortes. Ich werde beide Möglichkeiten vollständig erklären und beginne mit dem 'Until' Schlüsselwort.

Schauen Sie auf dieses Beispiel:

```
Bananen.i = 0

Repeat
  Debug Bananen
  Bananen + 1
Until Bananen > 10
```

Im Gegensatz zur 'While' Schleife befindet sich der Steuerausdruck am Ende der Schleife, und dieser wird darauf überprüft, ob er False zurückgibt. Wenn dies der Fall ist, wird die Schleife fortgesetzt. Gibt der Ausdruck True zurück, wird die Schleife beendet.

Weiterhin sollten Sie beachten, das 'Repeat' Schleifen, im Gegensatz zu 'While' Schleifen, immer mindestens einmal durchlaufen werden, bis der Ausdruck geprüft wird. Das wird hier demonstriert:

```
Bananen.i = 20

Repeat
  Debug Bananen
Until Bananen > 10
```

Sie können sehen, das die Schleife einmal durchlaufen wird, obwohl 'Bananen' größer als '10' ist. Die ist der Fall, weil der Ausdruck erst nach dem ersten Durchlauf der Schleife geprüft wird. Nachdem der Ausdruck geprüft wurde, gibt dieser True zurück, weil 'Bananen' größer als '10' ist, und die Schleife endet.

'Repeat' Schleifen bieten noch eine alternative Möglichkeit. Mit einem anderen Schlüsselwort können sie zu Endlosschleifen umfunktioniert werden. Um eine Endlosschleife zu konstruieren, verwenden Sie das 'ForEver' Schlüsselwort anstelle der 'Until' 'Ausdruck' Kombination, wie hier:

```
Zaehler.i = 0

Repeat
  Debug Zaehler
  Zaehler + 1
ForEver
```

Das ist Praktisch, wenn Sie eine Schleife Endlos laufen lassen wollen, oder Sie nicht sicher sind, welche Bedingung erfüllt sein muss, um die Schleife zu beenden, oder Sie haben mehrere Bedingungen die erfüllt sein müssen um aus der Schleife zu springen. Benutzen Sie den 'Programm beenden' Befehl (Menü: Debugger → Programm beenden) um dieses Beispiel zu schließen.

Manuelles beenden von Endlosschleifen

Wenn Sie in Ihrem Programm Schleifen verwenden, kann es manchmal vorkommen, das Sie ungewollt eine Endlosschleife erstellt haben. Das kann ein Problem für unser Programm sein, da diese Schleifen das komplette Programm blockieren, bis sie beendet sind. Das größte Kopfzerbrechen bereitet das beenden eines Programmes, in dem eine Endlosschleife läuft.

Die PureBasic IDE erledigt das auf einfache Weise für Sie. Wenn Sie ein Programm manuell beenden wollen, drücken Sie auf die 'Programm beenden' Schaltfläche in der Werkzeugleiste oder benutzen Sie den Menübefehl 'Menü: Debugger → Programm beenden'. Dieser Befehl beendet nicht nur die Schleife, sondern das komplette Programm.

Kontrollieren von Schleifen mit 'Break' und 'Continue'

All diese Schleifen können jederzeit mit zwei Schlüsselwörtern kontrolliert werden. Diese zwei Schlüsselwörter sind 'Break' und 'Continue'. Zunächst werde ich 'Break' erklären.
Wenn das 'Break' Schlüsselwort an irgendeiner Stelle in einer Schleife steht, wird die Schleife unmittelbar bei erreichen des Schlüsselwortes verlassen. Für ineinander verschachtelte Schleifen gibt es einen optionalen Parameter hinter dem 'Break' Schlüsselwort. Mit ihm kann die Anzahl der zu verlassenden Schleifen definiert werden. Schauen wir auf ein Beispiel mit dem 'Break' Schlüsselwort:

```
For x.i = 1 To 10
  If x = 5
    Break
  EndIf
  Debug x
Next x
```

In dieser 'For' Schleife habe ich die Schleife vorzeitig mit dem 'Break' Schlüsselwort (wenn 'x' gleich '5' ist) verlassen. Sie werden feststellen, das die Schleife abgebrochen wird, bevor '5' ins Debug Ausgabefenster geschrieben wird. Hier noch ein Beispiel zum beenden von verschachtelten Schleifen mit dem optionalen Parameter für das 'Break' Schlüsselwort:

```
For x.i = 1 To 10
  Zaehler.i = 0
  Repeat
    If x = 5
      Break 2
    EndIf
    Zaehler + 1
  Until Zaehler > 1
  Debug x
Next
```

In diesem Beispiel werden beide Schleifen verlassen wenn 'x' gleich '5' ist, ausgelöst durch die 'Break 2' Anweisung.

Als nächstes das 'Continue' Schlüsselwort. Dieses ermöglicht es ihnen jederzeit, den aktuellen Schleifendurchlauf zu unterbrechen und direkt den Nächsten Durchlauf in der derzeitigen Schleife zu starten zu starten. Das ist leichter als es klingt:

```
For x.i = 1 To 10
  If x = 5
    Continue
  EndIf
  Debug x
Next
```

Wenn 'x' gleich '5' (im fünften Durchlauf) wird in diesem Beispiel das 'Continue' Schlüsselwort verwendet. An dieser Stelle wird der fünfte Durchlauf unterbrochen und die Schleife beginnt wieder am Anfang mit dem sechsten Durchlauf, in dem 'x' dann den Wert '6' hat. Aufgrund dieses Sprunges und der Fortsetzung der Schleife wird im Debug Ausgabefenster der Wert '5' nicht erscheinen, da der Sprung zum Schleifenanfang vor dem 'Debug x' Befehl erfolgt.

Schleifen können für viele Dinge in der Computerprogrammierung verwendet werden, hauptsächlich zur Reduzierung von stumpfsinnigem Code und zum schnellen Durchlaufen großer Datenmengen. Ich hoffe, Sie haben einen guten Einblick bekommen, wie sie benutzt werden.

5. Andere Datenstrukturen

In diesem Kapitel erkläre ich Ihnen, wie Sie andere Systeme, zum speichern und organisieren von Daten, erstellen und benutzen, wie zum Beispiel die benutzerdefinierte Struktur, Arrays und Listen. Datenstrukturen wie diese sind essentiell für die Anwendungs- und Spieleentwicklung, da sie es erlauben, schneller und einfacher auf viele Werte artverwandter und nicht artverwandter Daten zuzugreifen. Wie immer werde ich alles ausführlich erklären und viele Beispiele mitliefern.

Strukturen

In Kapitel 2 habe ich Ihnen die eingebauten Typen (Byte, Character, Word, Long, Integer, Quad, Float, Double und String) erklärt. Mit dem 'Structure' Schlüsselwort haben Sie die Möglichkeit, Ihren eigenen strukturierten Datentyp zu entwerfen und diesen dann einer Variable zuzuweisen. Ihre eigene strukturierte Variable zu erstellen ist recht einfach, da Sie nur eine Gruppe von gängigen Variablennamen unter dem Dach einer Struktur zusammenfassen. Verwirrt? Dann lassen Sie uns auf eine Beispielstruktur mit mehreren Feldern schauen:

```
Structure PERSONENDETAILS
  Vorname.s
  Nachname.s
  Wohnort.s
EndStructure

Ich.PERSONENDETAILS

Ich\Vorname  = "Gary"
Ich\Nachname = "Willoughby"
Ich\Wohnort  = "zuhause"

Debug "Vorname: "  + Ich\Vorname
Debug "Nachname: " + Ich\Nachname
Debug "Wohnort: "  + Ich\Wohnort
```

Hier wurde die Struktur 'PERSONENDETAILS' mit dem 'Structure' Schlüsselwort erstellt. Anschließend wurden die Strukturelemente definiert, und zwar genau so, wie normale Variablen definiert werden. Das 'EndStructure' Schlüsselwort markiert das Ende der neuen Struktur. Nachdem die Struktur deklariert wurde, ist sie bereit zur Verwendung. Wir weisen einer Variable diesen strukturierten Typ genau so zu, wie wir jeden anderen Typ zuweisen würden. So wie hier:

```
Ich.PERSONENDETAILS
```

Hier ist der Variablenname 'Ich' und ihr Typ ist 'PERSONENDETAILS'. Um den einzelnen Variablen (manchmal auch Felder genannt), innerhalb der 'Ich' Strukturvariable, Werte zuweisen zu können, benutzen wir das '\' Zeichen. Wenn Sie auf das große Beispiel oben schauen, sehen Sie, dass das '\' Zeichen auch zum auslesen der Daten aus den individuellen Feldern benutzt wird. So wie hier:

```
Vater.PERSONENDETAILS
Vater\Vorname = "Peter"
Debug Vater\Vorname
```

In diesem kleinen Beispiel erstellen wir eine neue strukturierte Variable mit Namen 'Vater' mit dem benutzerdefinierten strukturierten Typ 'PERSONENDETAILS'. Wir weisen dem 'Vorname' Feld in der Variable 'Vater' den Wert 'Peter' zu. Dann geben wir diesen Wert im Debug Ausgabefenster aus.

Ich kann es Ihnen nicht Einprügeln, aber Strukturen sind eine unglaublich wichtige Sache. In Anwendungen können sie helfen alles mögliche zu definieren, vom Personen Datensatz bis zu Fenster Koordinaten. In Spielen unterstützen sie bei der Definition von Kugeln oder Raumschiffen, mit allen dazugehörigen Werten.

Speicher Betrachtungen

Die Größe einer strukturierten Variable im Speicher ist abhängig von den Feldvariablen die innerhalb der Initialstruktur definiert wurden. In der 'PERSONENDETAILS' Struktur sind drei Feldvariablen vom Typ String definiert worden, die jeweils 4 Byte im Speicher belegen (siehe Abb. 3 in Kapitel 2 für Speicherbelegung von Stringtypen). Deshalb belegt die Variable 'Ich' 12 Byte (3 x 4 Bytes) im Speicher. Das ganze können wir prüfen, wenn wir den Rückgabewert des 'SizeOf()' Befehls anzeigen.

```
Structure PERSONENDETAILS
  Vorname.s
  Nachname.s
  Wohnort.s
EndStructure
Debug SizeOf(PERSONENDETAILS)
```

Hier gibt 'SizeOf()' den Wert '12' zurück, was dem Wert in Bytes entspricht, den die Struktur im Speicher belegt.

Der 'SizeOf()' Befehl

Dieser Befehl gibt die Größe jeder Struktur oder definierten Variable in Bytes zurück. Er funktioniert nicht mit Arrays, Listen oder Interfaces. Dieser Befehl hat einen unschätzbaren Wert für die Windows Programmierung, da einige Win32 API Funktionen die Größe einer bestimmten Struktur oder Variable, als Parameter benötigen. In Kapitel 13 erfahren Sie mehr über das Windows Application Programming Interface (Win32 API).

Erben von anderen Strukturen

Strukturen können Felder von anderen Strukturen mittels des 'Extends' Schlüsselwortes erben.

```
Structure PERSONENDETAILS
  Vorname.s
  Nachname.s
  Wohnort.s
EndStructure

Structure ALLEDETAILS Extends PERSONENDETAILS
  Adresse.s
  Land.s
  Postleitzahl.s
EndStructure

Benutzer.ALLEDETAILS

Benutzer\Vorname      = "Andreas"
Benutzer\Nachname     = "Schweitzer"
Benutzer\Wohnort      = "zuhause"
Benutzer\Adresse      = "eine Straße"
Benutzer\Land         = "Deutschland"
Benutzer\Postleitzahl = "12345"

Debug "Vorname: "       + Benutzer\Vorname
Debug "Nachname: "      + Benutzer\Nachname
Debug "Wohnort: "       + Benutzer\Wohnort
Debug "Adresse: "       + Benutzer\Adresse
Debug "Land: "          + Benutzer\Land
Debug "Postleitzahl: "  + Benutzer\Postleitzahl
```

In diesem Beispiel erweitert die Struktur 'ALLEDETAILS' die Struktur 'PERSONENDETAILS'. Während ihrer Erstellung erbt die Struktur alle Felder von der 'PERSONENDETAILS' Struktur. Diese Felder erscheinen dann am Anfang der neuen Struktur. Diesen neu erstellten strukturierten Typ weisen wir der Variable 'Benutzer' zu. Dann weisen wir allen Feldern in 'Benutzer' Werte zu und Prüfen diese, in dem wir die Werte im Debug Ausgabefenster ausgeben.

Struktur Verbände (StructureUnion)

Struktur Verbände sind eine Möglichkeit Speicher zu sparen, indem man mehrere Variablen in einer Struktur zusammenfasst, die den selben Speicherbereich benutzen. Das Thema ist zum momentanen Zeitpunkt wahrscheinlich noch etwas zu fortgeschritten, als dass Sie es verstehen, aber der Vollständigkeit halber habe ich es an dieser Stelle hinzugefügt. Wenn Sie Kapitel 13 (Zeiger) lesen, werden Sie besser verstehen, wie Struktur Verbände funktionieren. Hier ein einfaches Beispiel:

```
Structure UNIONSTRUKTUR
  StructureUnion
    Eins.l
    Zwei.l
    Drei.l
  EndStructureUnion
EndStructure

Debug SizeOf(UNIONSTRUKTUR)

UnionVariable.UNIONSTRUKTUR

UnionVariable\Eins = 123
Debug UnionVariable\Eins

UnionVariable\Drei = 456
Debug UnionVariable\Eins
```

Nach der Deklaration der Struktur 'UNIONSTRUKTUR' haben wir die Schlüsselwörter 'StructureUnion' und 'EndStructureUnion' benutzt um die Variablen zu kapseln, die den gleichen Speicherbereich verwenden sollen. Wenn wir das Beispiel starten, meldet die erste Debug Anweisung '4' (Bytes) zurück. Dies ergibt sich daraus, dass die drei Variablen innerhalb der Struktur sich den gleichen Speicherplatz teilen und somit nur die Größe einer Long Variable zurückgegeben wird.

Weiter unten im Programm erstellen wir die Variable 'UnionVariable' vom Typ 'UNIONSTRUKTUR' und weisen dem Feld 'UnionVariable\Eins' den Wert '123' zu. Den Inhalt dieses Feldes geben wir dann im Debug Ausgabefenster aus. Dann weisen wir einen komplett neuen Wert dem Variablenfeld 'UnionVariable\Drei' zu, da dieses aber den gleichen Speicherplatz wie die anderen Felder belegt, können wir mit einem anderen Feldnamen auf diesen Wert zugreifen. In diesem Fall geben wir erneut den Wert von 'UnionVariable\Eins' im Debug Ausgabefenster aus und es ist, wie vorhersehbar, der gemeinsame Wert von 'UnionVariable\Drei'.

Strukturen können auch das enthalten, was als Statische Arrays bekannt ist, aber ich muss erst die Arrays erklären, bevor wir dieses Wissen auf Strukturen anwenden können. Arrays und Statische Arrays werden vollständig im nächsten Kapitel erklärt.

Arrays

In PureBasic kann ein Array eine benutzerdefinierte Menge an Variablen des selben Typs enthalten. Individuelle Variablen im Array werden über einen Index angesprochen, der aus einem aufsteigenden Bereich von Ganzzahlen besteht. Arrays können auch mit strukturierten Typen (nicht nur PureBasic Standardtypen) aufgebaut werden. Dieses Kapitel zeigt Ihnen alles, was Sie über Arrays in PureBasic wissen müssen.

Das 'Dim' Schlüsselwort

Arrays werden in PureBasic mittels des 'Dim' Schlüsselwortes erstellt, wie hier zum Beispiel:

```
Dim LongArray.l(2)
```

Lassen Sie mich diese Codezeile etwas genauer erklären. Zuerst benutzen wir das 'Dim' Schlüsselwort, um dem Compiler mitzuteilen, das wir ein Array definieren wollen. Dann geben wir dem Array einen Namen, ich habe hier den fantasievollen Namen 'LongArray' gewählt. Nach dem Namen weisen wir dem Array, wie bei normalen Variablen, mittels Suffix einen Typ zu. Ich habe hier das '.l' Suffix verwendet um dieses Array vom Typ Long zu definieren. Im Anschluss müssen wir die Anzahl der Indizes angeben. Wir benutzen Klammern um die letzte Indexnummer zu definieren. In unserem obigen Beispiel haben wir als letzten Indexwert '2' eingegeben, weshalb unser Array drei Indizes hat, da der Index eines Arrays immer mit Null beginnt. Wenn dieses Array erstellt ist, enthält jeder Indize eine Long Variable.

Dieses einfache Array wird auch als Eindimensionales Array bezeichnet, da es nur einen Index benötigt um jedem darin enthaltenen Element einen Wert zuzuweisen oder diesen abzurufen. In diesem erweiterten Beispiel erstellen wir ein Array und weisen den einzelnen Indizes Werte zu:

```
Dim LongArray.l(2)

LongArray(0) = 10
LongArray(1) = 25
LongArray(2) = 30
```

```
Debug LongArray(0) + LongArray(1)
Debug LongArray(1) * LongArray(2)
Debug LongArray(2) - LongArray(0)
```

Nachdem wir die Werte zugewiesen haben, geben wir ein paar Tests mit den in den Indizes gespeicherten Werten im Debug Ausgabefenster aus. Das erste ausgegebene Ergebnis ist zum Beispiel die Berechnung von '10 + 25', was den gespeicherten Werten in den Indizes '0' und '1' entspricht. Dann werden die Ergebnisse von '25 * 30' und '30 - 10' angezeigt. Array Indizes können auch durch Variablen oder Ausdrücke repräsentiert werden.

```
LetzterIndex.i = 2
ErsterIndex.i  = 0
Dim StringArray.s(LetzterIndex)

StringArray(ErsterIndex)     = "Einer ist keiner"
StringArray(ErsterIndex + 1) = "Zwei Enden hat die Wurst"
StringArray(ErsterIndex + 2) = "Drei sind aller Guten Dinge"

Debug StringArray(ErsterIndex)
Debug StringArray(ErsterIndex + 1)
Debug StringArray(ErsterIndex + 2)
```

Hier haben wir ein Array mit drei Indizes definiert und jeder enthält eine String Variable (beachten Sie das angehängte '.s' Suffix am Array Namen in der Zeile mit dem 'Dim' Befehl). Wir verwendeten die Variable 'LetzterIndex' um den letzten Index des Arrays zu definieren. Dann haben wir die Variable 'ErsterIndex' dazu verwendet, dem ersten Index des Arrays einen String zuzuweisen. In den späteren Zuweisungen haben wir dann Ausdrücke mit dem Plus Operator verwendet. Die gleiche Technik (einen Ausdruck als Index verwenden) wurde verwendet um die Werte der einzelnen Indizes des Arrays im Debug Ausgabefenster anzuzeigen. Siehe Abb. 16 für eine grafische Darstellung des obigen Arrays.

Eindimensionales String-Array

Index	Wert
0	Einer ist keiner
1	Zwei Enden hat die Wurst
2	Drei sind aller Guten Dinge

Abb. 16

Da Arrays sauber mittels Indizes geordnet sind, ist es sehr einfach diese sehr schnell mittels Schleifen zu durchlaufen. Um Ihren Appetit zu wecken, hier ein Beispiel Array mit tausend Indizes, von denen jeder mittels einer 'For' Schleife einen Wert zugewiesen bekommt. Mittels einer zweiten 'For' Schleife geben wir dann alle Indizes im Debug Ausgabefenster aus.

```
Dim TestArray.i(999)

For x = 0 To 999
  TestArray(x) = x
Next x

For x = 0 To 999
  Debug TestArray(x)
Next x
```

Starten Sie dieses Beispiel und schauen Sie ins Debug Ausgabefenster. Wie sie sehen können ist es mit Arrays ein leichtes tausend Werte zuzuweisen und abzurufen.

Mehrdimensionale Arrays

Der beste Weg ein mehrdimensionales Array zu beschreiben, ist der Vergleich mit einer Tabelle, die Spalten und Reihen enthält. Um ein mehrdimensionales Array zu erstellen, müssen Sie einfach die Anzahl der Spalten und Reihen, die Sie benötigen, definieren. Im folgenden Beispiel werden wir ein Array 'Tiere' erstellen, welches drei Indizes besitzt und jeder dieser drei Indizes besitzt weitere drei Indizes.

```
Dim Tiere.s(2, 2)

Tiere(0, 0) = "Das Schaf"
Tiere(0, 1) = "4 Beine"
Tiere(0, 2) = "Määh"

Tiere(1, 0) = "Die Katze"
Tiere(1, 1) = "4 Beine"
Tiere(1, 2) = "Miau"

Tiere(2, 0) = "Der Papagei"
Tiere(2, 1) = "2 Beine"
Tiere(2, 2) = "Kraah"

Debug Tiere(0, 0) + " hat " + Tiere(0, 1) + " und sagt " + Tiere(0, 2)
Debug Tiere(1, 0) + " hat " + Tiere(1, 1) + " und sagt " + Tiere(1, 2)
Debug Tiere(2, 0) + " hat " + Tiere(2, 1) + " und sagt " + Tiere(2, 2)
```

Nach der Definition des Arrays weisen wir den Indizes Werte zu. Da das 'Tiere' Array zwei Indexe hat, mittels derer die Daten zugewiesen und ausgelesen werden, bezeichnet man diese Art von Arrays als zweidimensionale Arrays. Zweidimensionale Arrays können einfach verstanden werden, wenn man sie in einer Tabelle mit Spalten und Reihen abbildet. Abb. 17 zeigt das 'Tiere' Array in einer ähnlichen Weise, wie Abb. 16 das Eindimensionale Array zeigt. Sie zeigt die Spalten und Reihen auf, die mittels der Indizes den Zugriff auf die einzelnen Felder des zweidimensionalen Arrays ermöglichen.

Zweidimensionales String-Array

Index	0	1	2
0	Das Schaf	4 Beine	Määh
1	Die Katze	4 Beine	Miau
2	Der Papagei	2 Beine	Kraah

Abb. 17

Wenn wir Abb. 17 als Referenz benutzen, können wir sehen, wie einfach es ist auf die Werte der verschiedenen Indizes zuzugreifen. Wenn Sie z. B. den Wert vom Reihenindex '1' und dem Spaltenindex '2' im Debug Ausgabefenster anzeigen wollen, tippen Sie einfach:

```
Debug Tiere(1, 2)
```

Dies sollte den Text 'Miau' ausgeben. Wenn Sie eine Reihe ersetzen wollen, können Sie das so erledigen:

```
Tiere(0, 0) = "Das Dreibein"
Tiere(0, 1) = "3 Beine"
Tiere(0, 2) = "Oo-la"
```

Das ersetzt die Strings 'Das Schaf', '4 Beine' und 'Määh' mit 'Das Dreibein', '3 Beine' und 'Oo-la' im Reihenindex '0' des 'Tiere Arrays'. Abb. 18 sieht aus wie Abb. 17 mit modifizierter erster Reihe.

Zweidimensionales String-Array (modifiziert)

Index	0	1	2
0	Das Dreibein	3 Beine	Oo-la
1	Die Katze	4 Beine	Miau
2	Der Papagei	2 Beine	Kraah

Abb. 18

Eine weitere Möglichkeit mehrdimensionale Arrays zu beschreiben ist, diese als Arrays in Arrays zu betrachten. Stellen Sie sich einfach vor dass jeder Array Indize ein weiteres Array enthält und schon haben Sie ein mehrdimensionales Array. Wie viele Arrays in jedem Indize enthalten sind, ist abhängig davon wie das Array am Anfang definiert wurde.

Im folgenden Beispiel zeige ich Ihnen, wie man ein-, zwei-, drei-, vier- und fünfdimensionale Arrays definiert.

```
Dim Tiere.s(5)
Dim Tiere.s(5, 4)
Dim Tiere.s(2, 5, 3)
Dim Tiere.s(1, 5, 4, 5)
Dim Tiere.s(2, 3, 6, 2, 3)
```

Nach zwei Dimensionen werden die Dinge etwas anstrengender für unseren Kopf, wenn Sie aber die Array im Array Theorie im Hinterkopf behalten, sollten Sie verstehen wie es funktioniert. Auch wenn ein Array maximal Zweihundertfünfundfünfzig (255) Dimensionen haben kann, sind Arrays mit mehr als zwei oder drei Dimensionen im Programmieralltag eher unüblich.

Arrays mit Strukturierten Typen

Bis jetzt haben wir gesehen, wie man verschiedene Arrays mit den PureBasic Standardtypen erstellt, Wir können aber auch Arrays mit benutzerdefinierten strukturierten Typen erstellen. Schauen Sie auf dieses einfache Beispiel eines eindimensionalen Arrays:

```
Structure FISCH
  Art.s
  Gewicht.s
  Farbe.s
EndStructure

Dim FischImGlas.FISCH(2)

FischImGlas(0)\Art     = "Clown Fisch"
FischImGlas(0)\Gewicht = "120 g"
FischImGlas(0)\Farbe   = "Rot, Weiß und Schwarz"

FischImGlas(1)\Art     = "Kasten Fisch"
FischImGlas(1)\Gewicht = "30 g"
FischImGlas(1)\Farbe   = "Gelb"

FischImGlas(2)\Art     = "Seepferdchen"
FischImGlas(2)\Gewicht = "60 g"
FischImGlas(2)\Farbe   = "Grün"

Debug FischImGlas(0)\Art + " " + FischImGlas(0)\Gewicht + " " + FischImGlas(0)\Farbe
Debug FischImGlas(1)\Art + " " + FischImGlas(1)\Gewicht + " " + FischImGlas(1)\Farbe
Debug FischImGlas(2)\Art + " " + FischImGlas(2)\Gewicht + " " + FischImGlas(2)\Farbe
```

Hier erstellen wir, nachdem wir die Struktur 'FISCH' definiert haben, mittels des 'Dim' Schlüsselwortes ein Array und benutzen 'FISCH' als Typ. Dies erfolgt auf die gleiche Weise, wie wir dem 'Tiere' Array den Typ '.s' (String) zugewiesen haben. Weiterhin habe ich '2' als letzten Arrayindex angegeben. Den einzelnen Indizes Werte zuzuweisen ist unglaublich einfach. Wir verquicken einfach die Syntax der Wertzuweisung von Arrays und Strukturen, wie hier:

```
FischImGlas(0)\Art = "Clown Fisch"
```

Lassen Sie uns das ganze in einfacher zu verstehende Teile zerlegen. Zuerst kommt der Name des Arrays, in diesem Fall 'Fisch im Glas'. Dann folgt, in Klammern eingeschlossen, der momentane Index, in diesem Fall Index '0'. Danach benutzen wir das '\' Zeichen um auf das Feld 'Art' innerhalb der 'FISCH' Struktur zuzugreifen. Diesen Typ hatten wir dem 'FischImGlas' Array zugewiesen. Dann benutzen wir den '=' Operator um diesem Feld einen Stringwert zuzuweisen. Anspruchslos! Um den gerade zugewiesenen Wert wieder abzurufen, benutzen wir exakt die gleiche Syntax aber ohne den Zuweisungsteil, wie hier:

```
Debug FischImGlas(0)\Art
```

Wenn wir einen Wert von einem anderen Index zuweisen oder auslesen wollen, tun wir das wie bei einem Array:

```
Debug FischImGlas(0)\Art
Debug FischImGlas(1)\Art
Debug FischImGlas(2)\Art
```

Das listet die 'Art' Felder aller Indizes des 'FischImGlas' Arrays auf. Um die anderen Felder zuzuweisen oder auszulesen, benutzen wir deren Namen:

```
Debug FischImGlas(1)\Art
Debug FischImGlas(1)\Gewicht
Debug FischImGlas(1)\Farbe
```

Hier geben wir alle Felder mit dem Index '1' im Debug Ausgabefenster aus. Um die Dinge besser zu verstehen schauen Sie auf Abb. 19, die das 'FischImGlas' Array grafisch darstellt.

Eindimensionales Strukturiertes Array

Index	'FISCH' Struktur		
0	*Art:* Clown Fisch	*Gewicht:* 120 g	*Farbe:* Rot, Weiß und Schwarz
1	*Art:* Kasten Fisch	*Gewicht:* 30 g	*Farbe:* Gelb
2	*Art:* Seepferdchen	*Gewicht:* 60 g	*Farbe:* Grün

Abb. 19

Wie bei den Standardtyp Arrays können Sie auch Mehrdimensionale Arrays mit strukturierten Typen definieren. Sie haben zugriff auf jedes einzelne Strukturfeld in jedem Indize des mehrdimensionalen Arrays. Mehrdimensionale strukturierte Arrays werden auf die gleiche Weise wie Eindimensionale Strukturierte Arrays erstellt. Wir fügen einfach mehrere Dimensionen hinzu.

Hier ein Beispiel, wie ein zweidimensionales strukturiertes Array definiert wird:

```
Structure FISCH
  Art.s
  Gewicht.s
  Farbe.s
EndStructure

Dim FischImGlas.FISCH(2, 2)
...
```

Ich werde jetzt keinen weiteren Code mehr an diese zweidimensionale Array anfügen, da dies den Rahmen sprengen würde. Schauen sie stattdessen auf Abb. 20 in der ein fiktives zweidimensionales strukturiertes Array dargestellt wird.

Zweidimensionales strukturiertes Array

Index	0	1	2
0	***Art:*** Clown Fisch ***Gewicht:*** 120 g ***Farbe:*** Rot,Weiß und Schwarz	***Art:*** Kasten Fisch ***Gewicht:*** 30 g ***Farbe:*** Gelb	***Art:*** Seepferdchen ***Gewicht:*** 60 g ***Farbe:*** Grün
1	***Art:*** Papageien Fisch ***Gewicht:*** 150 g ***Farbe:*** Rot	***Art:*** Engelfisch ***Gewicht:*** 120 g ***Farbe:*** Orange	***Art:*** Shrimp ***Gewicht:*** 30 g ***Farbe:*** Rosa
2	***Art:*** Goldfisch ***Gewicht:*** 60 g ***Farbe:*** Orange	***Art:*** Löwen Fisch ***Gewicht:*** 240 g ***Farbe:*** Schwarz und Weiß	***Art:*** Hai ***Gewicht:*** 2 kg ***Farbe:*** Grau

Abb. 20

Um einen Wert aus dieser Art Array herauszuholen, benötigen wir zwei Indizes und einen Feldnamen:

```
Debug FischImGlas(1, 1)\Art
```

Dies würde den Text 'Engelfisch' im Debug Ausgabefenster anzeigen. Wenn wir diesen oder andere Werte ändern wollen, benutzen wir die gleiche Methode wie beim auslesen aus diesem Bereich des Arrays:

```
FischImGlas(1, 1)\Art     = "Teufelfisch"
FischImGlas(1, 1)\Gewicht = "180 g"
FischImGlas(1, 1)\Farbe   = "Dunkelrot"
```

Das ändert alle Felder der 'FISCH' Struktur im mittleren Bereich des Arrays, mit der genauen Indexposition '1, 1'. Das Ergebnis sehen Sie in Abb. 21.

Zweidimensionales strukturiertes Array (modifiziert)

Index	0	1	2
0	***Art:*** Clown Fisch ***Gewicht:*** 120 g ***Farbe:*** Rot,Weiß und Schwarz	***Art:*** Kasten Fisch ***Gewicht:*** 30 g ***Farbe:*** Gelb	***Art:*** Seepferdchen ***Gewicht:*** 60 g ***Farbe:*** Grün
1	***Art:*** Papageien Fisch ***Gewicht:*** 150 g ***Farbe:*** Rot	***Art:*** Teufelfisch ***Gewicht:*** 180 g ***Farbe:*** Dunkelrot	***Art:*** Shrimp ***Gewicht:*** 30 g ***Farbe:*** Rosa
2	***Art:*** Goldfisch ***Gewicht:*** 60 g ***Farbe:*** Orange	***Art:*** Löwen Fisch ***Gewicht:*** 240 g ***Farbe:*** Schwarz und Weiß	***Art:*** Hai ***Gewicht:*** 2 kg ***Farbe:*** Grau

Abb. 21

Wie Sie sehen können, sind diese Arten von Arrays sehr praktisch, wenn auch etwas komplex (besonders wenn Sie beginnen, mit dreidimensionalen Arrays zu arbeiten), aber das Wissen über ihre Funktion ist sehr nützlich für Ihre spätere Programmierarbeit.

In Ihren jetzigen Programmen werden Sie nur Eindimensionale strukturierte Arrays verwenden, aber das Wissen um Mehrdimensionale strukturierte Arrays gibt Ihnen ein besseres Verständnis für fortgeschritteneren Code.

Redefinierung von bereits erstellten Arrays

Standard Arrays in PureBasic sind nicht komplett statisch, das heißt, sie können auf zwei verschiedene Arten redefiniert werden. Die erste Möglichkeit ist noch einmal das 'Dim' Schlüsselwort zu verwenden, womit das Array neu definiert wird. Allerdings gehen dabei alle bereits zugewiesenen Daten verloren. Die zweite Möglichkeit ist das 'ReDim' Schlüsselwort zu verwenden, welches das Array redefiniert ohne die enthaltenen Daten zu zerstören. Hier zwei Beispiele, die beide Verhaltensweisen demonstrieren. Zuerst schauen wir auf die Redefinierung mit dem 'Dim' Befehl:

```
Dim Hunde.s(2)

Hunde(0) = "Jack Russell"
Hunde(1) = "Alaska Husky"
Hunde(2) = "Border Collie"

Debug Hunde(0)
Debug Hunde(1)
Debug Hunde(2)

Dim Hunde.s(2)

Debug Hunde(0)
Debug Hunde(1)
Debug Hunde(2)
```

Hier habe ich nach den Erstellen des Arrays und der Wertzuweisung erneut den 'Dim' Befehl verwendet, um das Array mit der gleichen Größe zu redefinieren. Nach der zweiten Definition werden Sie feststellen, das der 'Debug' Befehl keine Werte mehr vom neu erstellten Array zurückliefert. Durch das redefinieren sind alle Daten im Array zerstört worden. Diese Datenvernichtung kann auch ein Gutes haben. Wenn Sie zum Beispiel den von einem Array belegten Speicher freigeben möchten, können Sie es durch redefinieren einfach mit Nullen (0) überschreiben. Hierdurch wird das Array auf eine minimale Größe im Speicher reduziert. Wenn Sie Arrays auf diese Weise redefinieren, müssen Sie darauf achten, immer den selben Typ zu verwenden, sonst erscheint eine Fehlermeldung.

Hier ein Beispiel, in dem die Daten Intakt bleiben, während wir das Array mit dem 'ReDim' Befehl redefinieren:

```
Dim Hunde.s(2)

Hunde(0) = "Jack Russell"
Hunde(1) = "Alaska Husky"
Hunde(2) = "Border Collie"

For x.i = 0 To 2
  Debug Hunde(x)
Next x

Debug ""

ReDim Hunde.s(4)

Hunde(3) = "Yorkshire Terrier"
Hunde(4) = "Windhund"

For x.i = 0 To 4
  Debug Hunde(x)
Next x
```

Hier habe ich den 'ReDim' Befehl zum redefinieren des Arrays verwendet, aber dieses mal habe ich dem Array bei der Redefinierung zwei zusätzliche Indizes hinzugefügt. Danach habe ich den beiden neuen Indizes ('3' und '4') Werte zugewiesen und den Inhalt des kompletten Arrays im Debug Ausgabefenster ausgegeben. Achten Sie darauf, dass die Daten der ersten Erstellung nicht verloren gegangen sind. Sie müssen trotzdem vorsichtig sein. Wenn Sie den 'ReDim' Befehl mit einer kleineren Zahl von Indizes verwenden, dann gehen die Daten in den überzähligen Indizes verloren. Wenn Sie ein mehrdimensionales Array mit dem 'ReDim' Befehl verändern wollen, können Sie nur die letzte Dimension verändern. Das ist für einen Basic Befehl wie diesen Standardverhalten.

Regeln für die Arraybenutzung

Auch wenn Arrays sehr flexibel sind, gibt es doch einige Regeln bei der Arbeit mit ihnen zu beachten. Diese Regeln sollten eingehalten werden, wenn Sie Arrays in Ihren Programmen verwenden.

1). Wenn ein Array mit dem 'Dim' Befehl re-/definiert wird, gehen alle enthaltenen Daten verloren.
2). Wenn ein Array mit dem 'ReDim' Befehl re-/definiert wird, werden die enthaltenen Daten erhalten.
3). Arrays können nur aus einem Variablentyp erstellt werden (Standardtyp oder strukturierter Typ).
4). Arrays können 'Global', 'Protected', 'Static' und 'Shared' sein. Siehe Kapitel 6 (Geltungsbereiche im Programm).
5). Die Größe eines Arrays wird nur durch den installierten RAM Speicher beschränkt.
6). Mehrdimensionale Arrays können bis zu 255 Dimensionen haben.
7). Arrays können dynamisch, mittels Variable oder Ausdruck, definiert werden.
8). Wenn Sie die Größe einer Dimension definieren, geben Sie nur die letzte Indexnummer ein (alle Indizes beginnen mit '0').
9). Dimensionen in mehrdimensionalen Arrays können unterschiedlich groß sein.

Statische Arrays in Strukturen

Statische Arrays in Strukturen weichen ein wenig von den normalen, zuvor beschriebenen, Arrays ab. Es liegt in der Natur der Sache, das Statische Arrays statisch sind und deshalb nach ihrer Definition nicht mehr modifiziert werden können. Diese Art von Arrays existiert nur in Strukturen.

Bei der Arbeit mit Statischen Arrays gibt es ebenfalls ein paar Regeln zu beachten:

1). Wenn ein Statisches Array in einer Struktur definiert wurde, kann es nicht mehr verändert werden.
2). Statische Arrays können (wie Strukturen) nicht redefiniert werden.
3). Sie können nur aus einem Variablentyp erstellt werden (Standardtyp oder strukturierter Typ).
4). Die Größe eines Arrays wird nur durch den installierten RAM Speicher beschränkt.
5). Statische Arrays können nur eine Dimension haben.
6). Die Dimensionsgröße kann dynamisch, mittels Variable oder Ausdruck, definiert werden.
7). Als Dimensionsgröße definieren Sie die Anzahl der enthaltenen Elemente, nicht den letzten Index.
8). Statische Arrays können nur über die Strukturvariable, in der sie definiert wurden, angesprochen werden.

So, nun habe ich Ihnen die Hauptregeln vorgestellt, und es folgt ein Beispiel für was sie verwendet werden:

```
Structure FAMILIE
  Vater.s
  Mutter.s
  Kinder.s[3]
  Familienname.s
EndStructure

Familie.FAMILIE

Familie\Vater        = "Andreas"
Familie\Mutter       = "Constanze"
Familie\Kinder[0]    = "Tabea"
Familie\Kinder[1]    = "Johanna"
Familie\Kinder[2]    = "Manuel"
Familie\Familienname = "Schweitzer"

Debug "Familienmitglieder:"
Debug Familie\Vater     + " " + Familie\Familienname
Debug Familie\Mutter    + " " + Familie\Familienname
Debug Familie\Kinder[0] + " " + Familie\Familienname
Debug Familie\Kinder[1] + " " + Familie\Familienname
Debug Familie\Kinder[2] + " " + Familie\Familienname
```

In diesem Beispiel hat die Struktur 'FAMILIE' ein Feld mit dem Namen 'Kinder', welches ein statisches Array ist. Als wir dieses Array definiert haben, haben wir die Zahl '3' benutzt. Das definiert, das Statische Array mit drei Indizes. Dieses Verhalten weicht komplett von den Standard Arrays ab, da diesen bei der Erstellung immer der letze Index übergeben wird. In unserem neuen statischen Array haben wir nun drei Indizes ('0', '1' und '2'). Später in diesem Beispiel weise ich allen Feldern in der Strukturvariable 'Familie' Werte zu, einschließlich den drei Indizes im statischen Array 'Kinder'. Sie werden bemerkt haben, das statische Arrays eine geringfügige Abweichung in der Syntax aufweisen, wenn wir Werte zuweisen und abrufen. Sie verwenden eckige Klammern anstelle der sonst üblichen Runden.

Datenzuweisung an ein statisches String-Array (Verwendung von eckigen Klammern)

```
Familie\Kinder[0] = "Tabea"
```

Datenzuweisung an ein Standard Long-Array (Verwendung von runden Klammern)

```
LongArray(0) = 10
```

Sie werden auch bemerkt haben, das Sie kein Schlüsselwort wie 'Dim' zum definieren des Arrays gebraucht haben. Sie fügen einfach eckige Klammern am Ende des Feldnamens hinzu. Innerhalb der eckigen Klammern definieren Sie die gewünschte Anzahl der Indizes für das neu erstellte statische Array. In der obigen Struktur 'FAMILIE' haben wir den String Typ zur Erstellung des statischen Arrays ausgewählt, Sie können aber auch jeden anderen in PureBasic eingebauten Typ oder eine weitere Struktur verwenden.

Lassen Sie uns auf ein weiteres einfaches Beispiel schauen:

```
Structure MITARBEITER
  MitarbeiterName.s
  MitarbeiterPersonalNummer.l
  MitarbeiterAdresse.s
  MitarbeiterKontaktNummern.l[2]
EndStructure

Dim Firma.MITARBEITER(9)

Firma(0)\MitarbeiterName              = "Bruce Dickinson"
Firma(0)\MitarbeiterPersonalNummer    = 666
Firma(0)\MitarbeiterAdresse           = "22 Acacia Avenue"
Firma(0)\MitarbeiterKontaktNummern[0] = 0776032666
Firma(0)\MitarbeiterKontaktNummern[1] = 0205467746

Firma(1)\MitarbeiterName              = "Adrian Smith"
Firma(1)\MitarbeiterPersonalNummer    = 1158
...
```

Hier habe ich eine benutzerdefinierte Struktur mit Namen 'MITARBEITER' definiert, um einen kleinen Firmenmitarbeiter Datensatz zu beschreiben und dann habe ich ein Standard Array erstellt, das zehn dieser Einträge enthält (erinnern Sie sich daran, das in einem Standard Array der letzte Index Eintrag angegeben wird und Dieser mit '0' beginnt). Innerhalb der 'MITARBEITER' Struktur habe ich ein Statisches Array vom Typ Long definiert, um zwei Kontakt Telefonnummern zu speichern. Dann habe ich damit begonnen die individuellen Mitarbeiter Einträge zu definieren, beginnend mit 'Firma(0)\...' und danach 'Firma(1)\...' und so weiter. Ich habe dieses Beispiel hier nicht vervollständigt, damit ich nicht anfange abzuschweifen, aber Sie können sehen wo das ganze hinführt und verstehen, wie es funktioniert.

Listen

Listen sind den Arrays in der Weise ähnlich, als dass sie in der Lage sind jede Menge Daten unter einem Namen zu verwalten. Der Unterschied zu den Arrays besteht darin, dass sie keinen Index zum zuweisen und auslesen von Daten benötigen.

Diese Listen sind wie Bücher die Daten vom Anfang bis zum Ende durchblättern oder einfach zu einer Seite springen und von dort weiterlesen. Listen sind weiterhin völlig dynamisch, das heißt sie wachsen und schrumpfen in Abhängigkeit der ihnen anvertrauten Menge an Daten. Wenn die Größe einer Liste wächst, werden die enthaltenen Daten in keiner Weise beschädigt oder verändert, weshalb Sie bedenkenlos an jeder benötigten Stelle in der Liste neue Elemente einfügen können.

Listen sind eine großartige Möglichkeit zum speichern und organisieren von Daten mit unbekannter Länge und es gibt verschiedene Möglichkeiten diese Daten zu Ordnen. In PureBasic gibt es eine eingebaute 'Linked List' Bibliothek, die verschiedene Befehle zu hinzufügen, löschen und tauschen von Listenelementen bereitstellt. Innerhalb der 'Sort' Bibliothek sind außerdem zwei Funktionen enthalten, die nur dem Sortieren von Listen dienen. Ich werde das an späterer Stelle noch erwähnen. Ein genereller Überblick mit Einführung in die mit PureBasic mitgelieferten Befehle erfolgt in Kapitel 7.

Das 'NewList' Schlüsselwort

Listen werden in PureBasic mit dem 'NewList' Schlüsselwort erstellt, wie in diesem Beispiel:

```
NewList Frucht.s()
```

Das definieren einer Liste ist dem definieren eines Arrays mit dem 'Dim' Befehl sehr ähnlich. Zuerst benutzen wir das 'NewList' Schlüsselwort, um dem Compiler mitzuteilen, das wir eine neue Liste erstellen wollen. Nachdem wir den Namen vergeben haben, definieren wir den Typ, welcher hier auch wieder '.s' (String) ist. Die angehängten Klammern schließen die Listen Definition ab. Sie werden festgestellt haben, das innerhalb der Klammern keine Indizes definiert wurden. das ist so, weil Listen diese nicht benötigen. Sie sind dynamisch und wachsen, wenn Sie Elemente hinzufügen. Lassen Sie uns einen Blick darauf werfen, wie wir unserer neuen Liste Elemente hinzufügen. Hier ein etwas vollständigeres Beispiel:

```
NewList Frucht.s()

AddElement(Frucht())
Frucht() = "Banane"

AddElement(Frucht())
Frucht() = "Apfel"
```

Da Listen keine Indizes haben, ist es zu Beginn etwas fremdartig mit ihnen zu arbeiten, da Sie nicht wissen können, wo Sie sich gerade in der Liste befinden. Im obigen Beispiel habe ich der Liste 'Frucht()' zwei neue Elemente hinzugefügt. Um dies zu realisieren, habe ich den 'AddElement()' Befehl aus der eingebauten Listen Bibliothek verwendet. Wenn Sie ein neues Element mit dieser Funktion hinzufügen wird nicht nur ein neues Element erstellt, sondern der Name der Liste zeigt auf das neuerstellte, leere Element. Deshalb benutzen wir einfach den Namen der Liste um Daten zuzuweisen. Beachten Sie, das wir immer Klammern benutzen:

```
Frucht() = "Banane"
```

Wenn wir ein weiteres Element mit dem 'AddElement()' Befehl hinzufügen, dann läuft exakt der gleiche Prozess ab. Zuerst wird das neue Element erstellt, dann zeigt der Listenname auf das neu erstellte, leere Element. Anschließend fügen wir dem neuen Element auf die gleiche Weise Daten hinzu:

```
Frucht() = "Apfel"
```

Eine Bemerkung über die Verwendung des Begriffs 'zeigen'

In dieser Einführung und Erklärung der Listen habe ich häufiger das Wort 'zeigen' verwendet. Wenn ich es hier verwende, sollten Sie es nicht mit dem 'zeigen' oder dem 'Zeiger' aus der Computertechnik verwechseln. Der Begriff 'Zeiger' (oder Pointer) aus der Computertechnik meint, auf einen bestimmten Bereich im Speicher des Computers zu zeigen. Es handelt sich also um eine Variable, die eine Speicheradresse enthält. Wenn ich diesen Begriff hier verwende, namentlich wenn ich erwähne, der Name der Liste zeigt auf das aktuelle Element, dann tue ich das in einem beschreibenden Sinn und meine nicht damit, dass der Name buchstäblich auf den Speicherbereich dieses Elementes zeigt. Für eine etwas ausführlichere Beschreibung der Zeiger (im Sinne der Computertechnik) verweise ich auf Kapitel 13 (Zeiger).

Sie denken vielleicht, dass hier ein Fehler vorliegt, da wir den Text 'Apfel' dem gleichen Namen zuweisen wie zuvor den Text 'Banane'. Da wir der Liste aber ein neues Element hinzugefügt haben, zeigt der Name 'Frucht()' auf das neue Element in der Liste. Wir können auch jederzeit überprüfen wie viele Elemente in unserer Liste enthalten sind, indem wir den eingebauten 'ListSize()' Befehl verwenden, wie hier:

```
Debug ListSize(Frucht())
```

Wenn wir obigen Code ausführen, dann wird die Anzahl der in der Liste 'Frucht()' enthaltenen Elemente im Debug Ausgabefenster angezeigt. In diesem Fall '2'.

Lassen Sie uns ein paar weitere Elemente zu dieser Liste hinzufügen und dann alle Werte der Elemente im Debug Ausgabefenster anzeigen. Hier wieder ein vollständiges Beispiel:

```
NewList Frucht.s()

AddElement(Frucht())
Frucht() = "Banane"

AddElement(Frucht())
Frucht() = "Apfel"

AddElement(Frucht())
Frucht() = "Birne"

AddElement(Frucht())
Frucht() = "Orange"

ForEach Frucht()
  Debug Frucht()
Next
```

In diesem größeren Beispiel haben wir eine Liste mit dem Namen 'Frucht()' erstellt, ihr vier Elemente hinzugefügt und diesen individuelle Werte zugewiesen. Dann haben wir diese Liste mit einer 'ForEach' Schleife durchlaufen und alle Elemente im Debug Ausgabefenster angezeigt. Das 'ForEach' Schlüsselwort dient zum definieren einer Schleife, die nur mit Listen verwendet werden kann.

Abb. 22 liefert eine kurze Übersicht der verfügbaren Befehle die in der 'Linked List' Bibliothek enthalten sind. Diese Tabelle ist keine komplette Referenz, wird hier aber als kurzer Leitfaden angeführt, um zu zeigen welche Befehle verfügbar sind, wenn der Bedarf dafür vorhanden ist. Die etwas fortgeschritteneren Befehle können Sie der PureBasic Hilfe entnehmen.

Die eingebaute 'Linked List' Bibliothek

Funktion	Beschreibung
`AddElement(Liste())`	Fügt der Liste ein Element hinzu.
`ClearList(Liste())`	Löscht alle Elemente der Liste.
`ListSize(Liste())`	Zählt die Elemente in der Liste.
`DeleteElement(Liste())`	Löscht das aktuelle Element in der Liste.
`FirstElement(Liste())`	Springt zum ersten Element in der Liste.
`InsertElement(Liste())`	Fügt ein weiteres Element vor dem aktuellen Element oder zu Beginn der Liste, wenn diese leer ist, ein.
`LastElement(Liste())`	Springt zum letzten Element in der Liste.
`ListIndex(Liste())`	Gibt die Position des aktuellen Elementes in der Liste zurück (Elementepositionen beginnen bei '0').
`NextElement(Liste())`	Springt zum nächsten Element in der Liste.
`PreviousElement(Liste())`	Springt zum vorherigen Element in der Liste.
`ResetList(List())`	Setzt die Listenposition auf '0' und macht das erste Element zum aktuellen Element.
`SelectElement(Liste(), Position)`	Macht das Element 'Position' zum aktuellen Element.

Abb. 22

Strukturierte Listen

Nachdem ich nun die Standard Listen erklärt habe, lassen Sie uns zu den strukturierten Listen übergehen. Diese sind den strukturierten Arrays darin sehr ähnlich, dass sie mit einer Struktur anstelle eines eingebauten Variablentyps definiert werden. Sie haben dann effektiv eine dynamische, sich in der Größe anpassende Liste, maskiert als ein strukturiertes Array, das abhängig von den gespeicherten Informationen selbständig wächst oder schrumpft. Lassen Sie uns auf ein früheres Beispiel schauen, aber diesmal unter Verwendung einer strukturierten Liste.

```
Structure FISCH
  Art.s
  Gewicht.s
  Farbe.s
EndStructure

NewList FischImGlas.FISCH()

AddElement(FischImGlas())
FischImGlas()\Art     = "Clown Fisch"
FischImGlas()\Gewicht = "120 g"
FischImGlas()\Farbe   = "Rot, Weiß und Schwarz"

AddElement(FischImGlas())
FischImGlas()\Art     = "Kasten Fisch"
FischImGlas()\Gewicht = "30 g"
FischImGlas()\Farbe   = "Gelb"

AddElement(FischImGlas())
FischImGlas()\Art     = "Seepferdchen"
FischImGlas()\Gewicht = "60 g"
FischImGlas()\Farbe   = "Grün"

ForEach FischImGlas()
  Debug FischImGlas()\Art + " " + FischImGlas()\Gewicht + " " + FischImGlas()\Farbe
Next
```

In diesem Beispiel können Sie sehen, dass nach dem erstellen der Liste das Zuweisen und Auslesen von Daten eine große Ähnlichkeit mit den strukturierten Arrays hat. Der Hauptunterschied besteht darin, dass die Array typischen Indizes nicht verwendet werden. Beachten Sie, dass bei der Verwendung des

'AddElement(FischImGlas())' Befehls ein neues Element erstellt wird, das die Definition der Initial Struktur verwendet. Dieser Befehl schiebt die aktuelle Position in der Liste auf das neue Element. Dann ist es sicher, dem neuen strukturierten Element Daten zuzuweisen, wie hier:

```
FischImGlas()\Art     = "Clown Fisch"
FischImGlas()\Gewicht = "120 g"
FischImGlas()\Farbe   = "Rot, Weiß und Schwarz"
```

Weil der Name 'FischImGlas()' nun auf das neue Element zeigt, ist es nicht nötig einen Index zu verwenden. Um auf die Felder innerhalb des Elementes zuzugreifen benutzen Sie auch hier wieder das '\' Zeichen. Am Ende des Beispiels verwenden wir eine 'ForEach' Schleife, um schnell und effizient die Daten der Liste im Debug Ausgabefenster anzuzeigen.

Pro und Kontra von Listen

Listen sind hervorragend zum speichern von Daten, unbekannter Größe und Menge, geeignet. In der Vergangenheit habe ich zum Beispiel ein Programm zum Verwalten von Haushaltsausgaben geschrieben und dafür eine strukturierte Liste zum speichern der detaillierten Ausgaben verwendet. Die Verwendung einer Liste macht es einfacher als ein Array, Daten hinzuzufügen, zu löschen oder diese zu sortieren.

Während ich das Programm schrieb, dachte ich, dass ich das Programm flexibel Gestalten muss, um für neu hinzu kommende Ausgaben gerüstet zu sein und die Möglichkeit habe alte Positionen wieder zu löschen (wenn ich, wie in diesem Fall ein neues Auto kaufe und/oder einen Kredit abbezahle, usw.). Das alles lässt sich sehr schön mit Listen erledigen. Wenn ich einen neuen Eintrag benötige, verwende ich den 'AddElement()' Befehl und wenn ich einen Eintrag löschen muss, verwende ich den 'DeleteElement()' Befehl. Nach dem Hinzufügen und Löschen von Daten in der Liste, transferiere ich alle diese Daten in eine Grafische Benutzeroberfläche (Graphical User Interface [GUI]), damit der Benutzer sie sehen und bearbeiten kann. In Kapitel 9 berichte ich ausführlicher über 'GUIs'.

Listen sind flexibler als Arrays, da sie einfach wachsen und schrumpfen können, aber sie benötigen bei gleichem Inhalt mehr RAM Speicher als Arrays. Das ist so, weil Arrays fortlaufende Speicherbereiche sind, die für jeden Indize immer nur den Platz im Speicher benötigen, der vom entsprechenden Erstelltyp benötigt wird. Listen unterscheiden sich dahingehend, dass sie pro Element annähernd die dreifache Menge (in Bezug zur Typ Größe) im Speicher belegen. Der Grund dafür ist, dass Listen keinen fortlaufenden Speicherbereich verwenden und deshalb Informationen mit abspeichern müssen, wo sich die anderen Elemente der Liste im Speicher befinden. Diesen Sachverhalt sollten Sie im Hinterkopf behalten wenn Sie mit riesigen Mengen von Daten hantieren. Der gesamt Speicherverbrauch kann bis zu dreimal so hoch sein wie der eigentliche Bedarf der Nutzdaten.

Sortieren von Arrays und Listen

Arrays und Listen sind großartig zum speichern von Daten aller Art geeignet und diese Datenstrukturen können einfach durchlaufen werden um schnell Daten abzurufen. Manchmal kann es aber nötig sein, die Daten in einem Array oder einer Liste so zu reorganisieren, dass sie in alphabetischer oder numerischer Reihenfolge vorliegen. Hier ein paar Beispiele wie die Befehle der 'Sort' Bibliothek (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Sort) verwendet werden, um Arrays und Listen zu sortieren.

Sortieren eines Standard Arrays

Ein Standard Array zu sortieren ist sehr einfach. Zuerst benötigen Sie ein mit Werten gefülltes Array, dann verwenden Sie den 'SortArray()' Befehl um es zu sortieren. Hier ein Syntax Beispiel:

```
SortArray(Array(), Optionen [, Start, Ende])
```

Der erste Parameter ist das zu sortierende Array. Beachten Sie die runden Klammern nach dem Array Namen, diese werden benötigt um das Array korrekt als Parameter zu übergeben. Der zweite Parameter ist eine Option, mit der Sie definieren wie das Array sortiert werden soll. Für den zweiten Parameter gibt es folgende Möglichkeiten:

'0': Sortiert das Array in aufsteigender Richtung und berücksichtigt Groß-/Kleinschreibung (a <> A)
'1': Sortiert das Array in absteigender Richtung und berücksichtigt Groß-/Kleinschreibung (a <> A)
'2': Sortiert das Array in aufsteigender Richtung und berücksichtigt keine Groß-/Kleinschreibung (a = A)
'3': Sortiert das Array in absteigender Richtung und berücksichtigt keine Groß-/Kleinschreibung (a= A)

Die eckigen Klammern um die letzten zwei Parameter deuten an, dass diese optional sind und nicht übergeben werden müssen wenn Sie den Befehl verwenden. Diese Parameter dienen zum definieren eines Bereiches im Array, der sortiert werden soll.

Wenn wir die obigen Informationen umsetzen, können wir ein vollständiges Array in absteigender Reihenfolge unter ignorieren der Groß-/Kleinschreibung sortieren. Verwenden Sie den Befehl wie hier:

```
Dim Frucht.s(3)

Frucht(0) = "banane"
Frucht(1) = "Apfel"
Frucht(2) = "Birne"
Frucht(3) = "Orange"

SortArray(Frucht(), 3)

For x.i = 0 To 3
  Debug Frucht(x)
Next x
```

Sortieren eines Strukturierten Arrays

Dieser Vorgang ist ein wenig komplizierter und erfordert einen etwas komplizierteren Sortierbefehl: 'SortStructuredArray()'. Hier ein Syntax Beispiel:

```
SortStructuredArray(Array(), Optionen, Offset, Typ [, Start, Ende])
```

Der erste Parameter ist der Array Name, komplett mit Klammern. Im zweiten Parameter werden die Sortieroptionen übergeben, die denen des 'SortArray()' Befehls entsprechen. Der Dritte Parameter ist der Offset (eine Position in der Originalstruktur) des Feldes, nach dem Sie sortieren wollen. Diesen erhalten Sie, wenn Sie den 'OffsetOf()' Befehl verwenden. Der 'OffsetOf()' Befehl gibt die Position eines Feldes, vom Beginn der Struktur aus betrachtet, in Bytes zurück. Der vierte Parameter definiert, welchen Typ das mittels Offset übergebene Feld hat. Sie können folgende eingebauten Konstanten als vierten Parameter übergeben, um zu beschreiben nach welchem Typ von Variable Sie sortieren wollen:

```
#PB_Sort_Byte      : Das Strukturfeld, nach dem sortiert werden soll, ist ein Byte (.b)
#PB_Sort_Word      : Das Strukturfeld, nach dem sortiert werden soll, ist ein Word (.w)
#PB_Sort_Long      : Das Strukturfeld, nach dem sortiert werden soll, ist ein Long (.l)
#PB_Sort_String    : Das Strukturfeld, nach dem sortiert werden soll, ist ein String (.s oder $)
#PB_Sort_Float     : Das Strukturfeld, nach dem sortiert werden soll, ist ein Float (.f)
#PB_Sort_Double    : Das Strukturfeld, nach dem sortiert werden soll, ist ein Double (.d)
#PB_Sort_Quad      : Das Strukturfeld, nach dem sortiert werden soll, ist ein Quad (.q)
#PB_Sort_Character : Das Strukturfeld, nach dem sortiert werden soll, ist ein Character (.c)
#PB_Sort_Integer   : Das Strukturfeld, nach dem sortiert werden soll, ist ein Integer (.i)
#PB_Sort_Ascii     : Das Strukturfeld, nach dem sortiert werden soll, ist ein ASCII-Zeichen (.u)
#PB_Sort_Unicode   : Das Strukturfeld, nach dem sortiert werden soll, ist ein Unicode-Zeichen (.u)
```

Die letzten beiden Parameter des Befehls sind optional und müssen bei der Benutzung nicht angegeben werden. Sie dienen zum definieren eines Bereiches im Array, der sortiert werden soll. Mit obiger Information können wir ein vollständiges strukturiertes Array in aufsteigender Richtung Nach dem 'Reichweite' Feld sortieren, wie hier:

```
Structure WAFFE
  Name.s
  Reichweite.l
EndStructure

Dim Waffen.WAFFE(2)

Waffen(0)\Name       = "Phased Plasma Rifle"
Waffen(0)\Reichweite = 40

Waffen(1)\Name       = "SVD-Dragunov Sniper Rifle"
Waffen(1)\Reichweite = 3800

Waffen(2)\Name       = "HK-MP5 Sub-Machine Gun"
Waffen(2)\Reichweite = 300
```

```
SortStructuredArray(Waffen(), 0, OffsetOf(WAFFE\Reichweite), #PB_Sort_Long)

For x.i = 0 To 2
  Debug Waffen(x)\Name + " : " + Str(Waffen(x)\Reichweite)
Next x
```

In diesem Beispiel habe ich das 'Reichweite' Feld ausgewählt, nach dem das strukturierte Array sortiert werden soll. Deshalb habe ich dem Sortierbefehl mitgeteilt, er soll den Offset 'OffsetOf(WAFFE\Reichweite)' verwenden und mit Übergabe der '#PB_Sort_Long' Konstante, das dieses Feld vom Typ Long ist.

Sortieren einer Standard Liste

Das sortieren einer Standard Liste ist extrem einfach. Zuerst benötigen Sie eine mit Werten gefüllte Liste und dann verwenden Sie den 'SortList()' Befehl um diese zu sortieren. Hier ein Syntax Beispiel:

```
SortList(ListenName(), Optionen [, Start, Ende])
```

Der erste Parameter ist die Liste, die sortiert werden soll. Beachten Sie die runden Klammern nach dem Listen Namen, diese werden benötigt um die Liste korrekt als Parameter zu übergeben. Der zweite Parameter ist die Sortieroption, welches die gleichen Optionen wie beim 'SortArray()' Befehl sind. Die letzten beiden Parameter können verwendet werden, um einen Sortierbereich in der Liste zu definieren.

Mit obiger Information können wir ein komplette Liste in absteigender Richtung unter Berücksichtigung von Groß- und Kleinschreibung sortieren, indem wir den Sortierbefehl wie hier verwenden:

```
NewList Frucht.s()

AddElement(Frucht())
Frucht() = "banane"

AddElement(Frucht())
Frucht() = "Apfel"

AddElement(Frucht())
Frucht() = "Orange"

SortList(Frucht(), 0)

ForEach Frucht()
  Debug Frucht()
Next
```

Sortieren einer Strukturierten Liste

Das sortieren einer strukturierten Liste ist ein wenig komplizierter, da es einen etwas komplizierteren Sortierbefehl benötigt: 'SortStructuredList()'. Hier ein Syntax Beispiel:

```
SortStructuredList(List(), Optionen, Offset, Typ [, Start, Ende])
```

Der erste Parameter ist der Listenname, komplett mit Klammern. Der Zweite ist die Sortieroption (die gleichen Optionen wie beim 'SortArray()' Befehl). Der dritte Parameter ist der Offset (eine Position in der Original Struktur) des Feldes, nach dem sortiert werden soll. Dieser wird mit dem 'OffsetOf()' Befehl ermittelt. Der vierte Parameter definiert den Typ der Variable, die sich am zuvor übergebenen Offset befindet. Sie können zur Typangabe auch hier wieder die gleichen Konstanten wie beim 'SortStructuredArray()' Befehl verwenden. Die letzten beiden Parameter können verwendet werden, um einen Sortierbereich in der Liste zu definieren.

Mit obiger Information können wir eine vollständige strukturierte Liste in aufsteigender Richtung, unter Beachtung von Groß- und Kleinschreibung, sortieren, wie hier:

```
Structure FRAU
  Name.s
  Gewicht.s
EndStructure

NewList Frauen.FRAU()
```

```
AddElement(Frauen())
Frauen()\Name    = "Mia"
Frauen()\Gewicht = "54 kg"

AddElement(Frauen())
Frauen()\Name    = "Rosie"
Frauen()\Gewicht = "120 kg"

AddElement(Frauen())
Frauen()\Name    = "Sara"
Frauen()\Gewicht = "63,5 kg"

SortStructuredList(Frauen(), 0, OffsetOf(FRAU\Name), #PB_Sort_String)

ForEach Frauen()
  Debug Frauen()\Name + " : " + Frauen()\Gewicht
Next
```

In diesem Beispiel habe ich das 'Name' Feld ausgewählt, nach dem die Liste sortiert werden soll. Deshalb habe ich dem Sortierbefehl mitgeteilt, er soll den Offset 'OffsetOf(FRAU\Name)' verwenden und mit Übergabe der '#PB_Sort_String' Konstante, das dieses Feld vom Typ String ist.

Sortieren Einfach gemacht

Alle vorhergehenden Sortierbeispiele können auf die gleiche Weise sowohl zum sortieren von allen numerischen Feldern als auch von Stringfeldern verwendet werden. Sie müssen einfach nur die richtigen Sortieroptionen angeben, wenn Sie die verschiedenen Sortierbefehle benutzen. Das Sortieren von Arrays und Listen, egal ob sie eine Struktur verwenden oder nicht, ist nur eine Frage der Verwendung der richtigen Optionen und Offsets. Probieren Sie ein paar eigene Beispiele aus, um das Sortieren mit diesen Befehlen zu üben.

6. Prozeduren und Unterroutinen

In diesem Kapitel werde ich auf Prozeduren und Unterroutinen eingehen. Prozeduren sind ein wesentlicher Teil jeder Programmiersprache. Sie bilden die Grundlage für sauber Strukturierten Code und ermöglichen dessen Wiederverwendung. PureBasic ist eine Strukturierte Programmiersprache und Prozeduren unterstützen den Programmierer bei der Erstellung der Programmstruktur. Ich werde in diesem Kapitel auch die Unterroutinen behandeln, aber diese sind in vielen Programmiersprachen kein wesentlicher Bestandteil. Weil sie ein Teil von PureBasic sind, werden sie hier der Vollständigkeit halber erwähnt.

Warum soll man Prozeduren und Unterroutinen verwenden?

In der Theorie benötigen Sie niemals Prozeduren oder Unterroutinen. Sie schreiben einfach einen riesigen Quelltext. In diesem würden Sie aber schnell den Überblick verlieren und ich denke Sie würden sich darin sehr häufig wiederholen. Prozeduren und Unterroutinen verschaffen eine Möglichkeit, Separate Codestücke jederzeit in Ihrem Hauptprogramm aufzurufen. Es gibt keine funktionellen Einschränkungen bei diesen Codestücken, Sie können damit alles Mögliche Programmieren. Sie können zum Beispiel eine Prozedur aufrufen, die einen Klang abspielt oder einen Bildschirm aktualisiert oder Sie rufen einfach eine Prozedur auf, die ein voll funktionsfähiges Programm enthält.

Die Verwendung von Prozeduren und Unterroutinen sind eine gute Möglichkeit, Ihren Code übersichtlich und klein zu halten, und sie geben Ihm eine klare Struktur. Gut strukturierter Code ist später immer besser zu lesen, besonders wenn Sie ihren Code später wieder öffnen, um ihn zu erweitern, oder Sie arbeiten in einem Team und viele Personen arbeiten am gleichen Code.

Unterroutinen

Unterroutinen werden in PureBasic selten verwendet und einige sagen ihnen nach, dass sie zu schlechtem Programmierstil verführen. Manchmal können sie aber auch sehr nützlich sein, wenn Sie zum Beispiel sehr schnell einen bestimmten Codeabschnitt aufrufen wollen. Unterroutinen bieten die Möglichkeit, an eine andere Stelle weiter unten in Ihrem Code zu springen, diesen auszuführen und danach wieder an die Stelle unmittelbar nach der Sprunganweisung zurückzukehren. Um mit dieser Methode an eine bestimmte Stelle in Ihrem Code zu springen, müssen Sie eine Sprungmarke definieren, die dann angesprungen wird. Dieses Sprungziel wird in Ihrem Code durch eine Unterroutinen Sprungmarke definiert.

Um eine solche Unterroutinen Sprungmarke zu definieren, können Sie jeden beliebigen Namen verwenden. Er muss einzig die Richtlinien für die Vergabe von Variablennamen einhalten. Statt eines Suffixes verwenden Sie zur Sprungmarkendefinition einen Doppelpunkt, wie hier:

```
SprungMarke:
```

Diese Unterroutinen Sprungmarke kann von jedem Punkt in Ihrem Code mit folgendem Befehl angesprungen werden:

```
Gosub SprungMarke
```

Beachten Sie, das wir im 'Gosub' Befehl keinen Doppelpunkt verwenden, wenn wir spezifizieren zu welcher Unterroutinen Sprungmarke wir springen wollen. Nach der Ziel-Sprungmarke fügen Sie ganz normalen PureBasic Code ein, den Sie in der Unterroutine ausführen möchten. Nach diesem Code müssen Sie dann angeben, dass Sie zum Haupt Code zurückspringen möchten. Hierzu verwenden Sie folgenden Befehl:

```
Return
```

Hier ist ein kleines Beispiel, das die Verwendung einer Unterroutine demonstriert:

```
a.i = 10

Debug a
Gosub Berechnung
Debug a
End

Berechnung:
  a = a * 5 + 5
Return
```

In diesem Beispiel weisen wir der Variable 'a' den Wert '10' zu und geben Deren Inhalt dann im Debug Ausgabefenster aus. Dann benutzen wir das 'Gosub' Schlüsselwort um zur Sprungmarke 'Berechnung' zu springen. Man kann sagen, wenn ein Doppelpunkt hinter einem Namen ist, handelt es sich um eine Sprungmarke. Dies können Sie in der IDE auch an der farbigen Markierung erkennen. Nachdem wir zu dieser Sprungmarke gesprungen sind, wird eine kleine Berechnung an der Variable 'a' ausgeführt und wir kehren mit dem 'Return' Schlüsselwort wieder ins Hauptprogramm zurück. Wenn wir zurückkehren, landen wir in der Programmzeile nach dem 'Gosub' Befehl. In diesem Fall kehren wir in die Zeile 'Debug a' zurück, die erneut den Wert von 'a' im Debug Ausgabefenster ausgibt. Dieser Wert hat sich nun, entsprechend der Berechnung in der Unterroutine, geändert.

Eine Anmerkung bezüglich der Position von Unterroutinen in Ihrem Code

Wenn Sie Unterroutinen in Ihrem Programm verwenden, müssen Sie einen wichtigen Punkt beachten: Unterroutinen dürfen nur nach einem 'End' Schlüsselwort definiert werden. Das 'End' Schlüsselwort beendet nicht nur unmittelbar das Programm, wenn es ausgeführt wird, sondern es definiert auch das Ende des Programms während des Kompilierens. Alle Unterroutinen müssen hinter diesem Schlüsselwort stehen, sonst kann es zu Fehlfunktionen in Ihrem Programm kommen.

Springen aus einer Unterroutine

Das ist eine generell geächtete, schlechte Programmiertechnik. Da ich nun aber schon über Unterroutinen berichte, folge ich dem Stil des restlichen Buches und beschreibe sie vollständig - und die richtige Syntax befähigt eine Unterroutine zu solchen Dingen.

Da alle Unterroutinen ein 'Return' Schlüsselwort zum Rücksprung in den Hauptcode enthalten, muss eine Unterroutine bei einem Sprung vorzeitig zurückkehren, um diese korrekt zu verlassen. Dies wird durch die Schlüsselwörter 'FakeReturn' und 'Goto' erreicht. Hier ein Beispiel:

```
a.i = 10

Debug a
Gosub Berechnung
Debug a
End

StarteErneut:
a.i = 20
Debug a
Gosub Berechnung
Debug a
End

Berechnung:
If a = 10
  FakeReturn
  Goto StarteErneut
Else
  a = a * 5 + 5
EndIf
Return
```

Wie Sie in diesem kleinen Beispiel sehen können, sind die Dinge komplizierter geworden und der Code springt kreuz und quer. Wie ich bereits vorher erwähnt habe, ist das schlechter Programmierstil und produziert hässlichen Code.

Dieses Beispiel ist sehr geradlinig, wenn Sie es aufmerksam verfolgen. Sie können in der 'Berechnung' Unterroutine sehen, das wir im Falle eines Sprunges den 'Goto' Befehl benötigen, vor dem aber ein 'FakeReturn' stehen muss. Wir springen zu einer neuen Sprungmarke (Unterroutine) mit Namen 'StarteErneut', die ein weiteres nahezu selbsterklärendes Programm mit einem weiteren 'End' Schlüsselwort enthält. Das ist nicht wirklich ideal.

Ich hoffe, dieses Beispiel gibt Ihnen einen Überblick zu den Unterroutinen und vermittelt wie im Programm herumgesprungen wird. Ich hoffe, dass Sie diese Praktiken ablehnen und nicht als regulären Weg zur Programmierung in Erwägung ziehen. Ich fühle, dass Sie ihren Code mit Prozeduren besser strukturieren können.

Grundlegendes zu Prozeduren

Der Begriff 'Strukturierte Programmierung' ist ein Begriff, der PureBasic und die Gesinnung die zu seiner Entwicklung geführt haben, sehr klar definiert. PureBasic kann als eine 'Prozedurale' und 'Strukturierte' Programmiersprache beschrieben werden. Die Architektur hinter der Strukturierten Programmierung wird durch Prozeduren geschaffen. Sie bilden buchstäblich die Struktur in der strukturierten Programmierung.

Prozeduren (in anderen Programmiersprachen auch 'Funktionen' genannt) sind der wichtigste Teil der Funktionalität, die PureBasic zur Verfügung stellt. In ihrer grundlegendsten Form sind sie einfach eine Hülle für ein paar Code Zeilen, die jederzeit in Ihrem Programm aufgerufen werden können (ähnlich den Unterroutinen). Prozeduren können so oft aufgerufen werden, wie Sie wollen und können, wie die Unterroutinen, jeden beliebigen PureBasic Code enthalten. Anders als bei den Unterroutinen können den Prozeduren Startparameter übergeben werden und sie können einen Wert (beliebiger in PureBasic eingebauter Typ) zurückgeben.

Lassen Sie uns mit einem einfachen Prozedur Beispiel beginnen:

```
Procedure KinderLied()
  Debug "Alle meine Entchen schwimmen auf dem See,"
  Debug "schwimmen auf dem See,"
  Debug "Köpfchen in das Wasser, Schwänzchen in die Höh'."
EndProcedure

KinderLied()
```

Damit Sie eine Prozedur in Ihrem Programm verwenden können, müssen Sie diese erst definieren. Hier habe ich eine Prozedur mit dem Namen 'KinderLied' mittels des 'Procedure' Schlüsselwortes definiert. Nach dieser Zeile habe ich den Code eingegeben, den die Prozedur enthalten soll. Das Ende der Prozedur wird durch das 'EndProcedure' Schlüsselwort definiert. Wir rufen dann die Codezeilen innerhalb der Prozedur jederzeit mit dem Prozedurnamen auf, wie hier:

```
KinderLied()
```

Bevor wir weitergehen möchte ich auf ein paar Dinge in diesem Beispiel hinweisen, die etwas näherer Erklärung bedürfen. Zuerst haben Sie wahrscheinlich bemerkt, dass sich hinter dem Prozedurnamen ein mysteriöser Satz von Klammern befindet. Innerhalb dieser Klammern werden alle der Prozedur zu übergebenden Parameter definiert. Auch wenn Sie keine Parameter übergeben (wie in unserem Beispiel der Fall), müssen Sie trotzdem die Klammern anfügen. Auch wenn Sie die Prozedur aufrufen, müssen Sie immer die Klammern als Teil des Prozeduraufrufs mit angeben. Die Übergabe von Parametern an Prozeduren werde ich an späterer Stelle in diesem Kapitel erklären.

Ein weiterer Punkt, auf den ich hinweisen möchte ist der, dass Prozedurnamen wie z.B. 'KinderLied()' keine Leerzeichen enthalten dürfen und den selben Namensrichtlinien entsprechen müssen, die auch für Variablen gelten.

Wenn Sie das obige Beispiel starten, sehen Sie im Debug Ausgabefenster das Kinderlied, so wie es in der 'KinderLied()' Prozedur programmiert wurde. Wenn Sie die Prozedur in diesem Beispiel wiederholen möchten, rufen Sie sie einfach immer wieder mit ihrem Namen auf.

```
Procedure KinderLied()
  Debug "Alle meine Entchen schwimmen auf dem See,"
  Debug "schwimmen auf dem See,"
  Debug "Köpfchen in das Wasser, Schwänzchen in die Höh'."
EndProcedure

For x.i = 1 To 5
  KinderLied()
Next x
```

Prozeduren können von jeder Stelle in Ihrem Programm aufgerufen werden, wie im obigen Beispiel, in dem ich die Prozedur 'KinderLied()' aus einer Schleife heraus aufgerufen habe. Prozeduren können aber auch aus Prozeduren heraus aufgerufen werden:

```
Procedure KinderLied2()
  Debug "schwimmen auf dem See,"
  Debug "Köpfchen in das Wasser, Schwänzchen in die Höh'."
EndProcedure
```

```
Procedure KinderLied1()
  Debug "Alle meine Entchen schwimmen auf dem See,"
  KinderLied2()
EndProcedure

KinderLied1()
```

Hier können Sie sehen, das 'KinderLied1()' einen Aufruf von 'KinderLied2()' enthält.

Eine Anmerkung bezüglich der Position von Prozeduren in Ihrem Code

Das obige Beispiel demonstriert schön, warum die Position von Prozeduren im Code ein Thema ist. Sie werden festgestellt haben, dass ich zuerst 'KinderLied2()' programmiert habe, vor 'KinderLied1()'. Das habe ich absichtlich so gemacht und es war notwendig, da Sie immer erst eine Prozedur deklarieren müssen, bevor Sie sie zum ersten mal verwenden können. Da 'KinderLied1()' die Prozedur 'KinderLied2()' aufruft, musste letztere zuerst definiert werden. Der PureBasic Compiler ist ein sogenannter 'One Pass' (ein Durchlauf) Compiler, das heißt der Quelltext wird während des Kompilierens nur einmal von Oben nach Unten durchlaufen. Er wird einen Fehler melden, wenn ihm ein Prozeduraufruf begegnet, zu dem bis zu diesem Zeitpunkt noch keine Definition vorliegt. Dieses einfache Beispiel zeigt damit gleich noch, warum die Prozeduren in den meisten Quelltexten oben stehen. Wie bei den meisten Dingen im Leben, gibt es auch hierzu eine Ausnahme. Sie können das 'Declare' Schlüsselwort verwenden um die Position einer Prozedurdefinition in Ihrem Quelltext zu ändern. Das 'Declare' Schlüsselwort definiert keine Prozedur, es teilt dem Compiler lediglich mit, welche Prozeduraufrufe vor der entsprechenden Definition auftauchen können. Das erlaubt es, jede Prozedur erst später im Quelltext zu definieren. Hier ein Beispiel:

```
Declare KinderLied1()
Declare KinderLied2()

KinderLied1()

Procedure KinderLied1()
  Debug "Alle meine Entchen schwimmen auf dem See,"
  KinderLied2()
EndProcedure

Procedure KinderLied2()
  Debug "schwimmen auf dem See,"
  Debug "Köpfchen in das Wasser, Schwänzchen in die Höh'."
EndProcedure
```

Wenn Sie das 'Declare' Schlüsselwort verwenden, definieren Sie die erste Zeile der Prozedur, genau so, wie Sie das mit dem 'Procedure' Schlüsselwort erledigen würden. Sie sind dann völlig frei, die Prozedur jederzeit in Ihrem Programm aufzurufen. Da dies aber nur eine Deklaration und keine Definition ist, müssen Sie an anderer Stelle in Ihrem Code natürlich auch noch die Prozedur definieren. Die Prozedur kann nun überall definiert werden, auch am Ende Ihres Quellcodes.

Wenn Sie obigen Quellcode betrachten, können Sie sehen, dass ich zwei Deklarationen mit dem 'Declare' Schlüsselwort durchgeführt habe. Danach bin ich frei, die 'KinderLied1()' Prozedur aufzurufen. Die eigentlichen Prozedurdefinitionen befinden sich am Ende des Quelltextes.

Geltungsbereiche im Programm

Wenn Sie Prozeduren verwenden, ist es auch wichtig zu wissen, welche Geltungsbereiche ein Programm haben kann. Dieser Abschnitt zeigt, welchen Geltungsbereich jede in Ihrem Programm verwendete Variable, jedes Array oder jede Liste hat. Lassen Sie mich mit einem einfachen Beispiel beginnen, das eine Variable verwendet:

```
a.i = 10

Procedure ZeigeWert()
  Debug a
EndProcedure

ZeigeWert()
```

Hier habe ich eine Integer Variable mit Namen 'a' definiert und ihr dann den Wert '10' zugewiesen. Außerdem habe ich eine Prozedur mit Namen 'ZeigeWert()' definiert und in diese Prozedur habe ich eine

Codezeile geschrieben die den Wert von 'a' ausgibt. Die letzte Zeile in dem Beispiel ruft die Prozedur auf. Wenn Sie nun das Beispiel starten, erwarten Sie dass der Wert von 'a' (welcher '10' ist) im Debug Ausgabefenster erscheint. Das ist aber nicht der Fall. Sie werden bemerken, dass stattdessen der Wert '0' ausgegeben wird. Schuld daran ist der Geltungsbereich des Programms.

Lassen Sie mich etwas tiefer in dieses Beispiel einsteigen und genau erklären was passiert ist. Das erste was Sie sich merken müssen ist die Tatsache, dass wenn Sie in PureBasic eine Variable definieren, diese standardmäßig einen lokalen Geltungsbereich hat. Das bedeutet, wenn wir in obigem Beispiel die erste Variablendefinition vornehmen ('a.i = 10'), hat diese Variable einen lokalen Geltungsbereich, in diesem Fall das Hauptprogramm. Solange die Variable nicht Global bekannt gemacht ist, hat keine Prozedur die Möglichkeit sie zu sehen oder zu verwenden.

In der 'ZeigeWert' Prozedur geben wir den Wert von 'a' aus, aber diese Variable ist nicht die gleiche wie die außerhalb der Prozedur. Sie haben den gleichen Namen, sind aber nicht die gleichen Variablen. Die Variable in der Prozedur ist in Dieser lokal definiert, weshalb keine der Variablen außerhalb ihres Geltungsbereiches angesprochen werden kann.

Das kann sehr verwirrend sein, besonders wenn beide lokalen Variablen, wie hier im Beispiel, den gleichen Namen haben aber ich denke, ich habe den lokalen Geltungsbereich gut erklärt.

Wie ich bereits gesagt habe, muss die Variable Global erreichbar gemacht werden. Dann können alle Prozeduren sie sehen und benutzen. Lassen Sie mich Ihnen zeigen wie das funktioniert.

```
Global a.i = 10

Procedure ZeigeWert()
  Debug a
EndProcedure

ZeigeWert()
```

Wie Sie sehen, habe ich nur eine Änderung am Quelltext vorgenommen. Ich habe das 'Global' Schlüsselwort vor der Variablendefinition verwendet. Dieses definiert die Variable 'a' als Global und alle Prozeduren können die Variable nun sehen und benutzen. Im Debug Ausgabefenster wird nun der korrekten Wert '10' angezeigt.

Wenn wir umgekehrt eine Variable in einer Prozedur definieren, gilt diese auch nur lokal in der Prozedur. Schauen Sie auf diese Beispiel:

```
Procedure ZeigeWert()
  a.i = 10
EndProcedure

ZeigeWert()

Debug a
```

Wie erwartet gibt die letzte Zeile 'Debug a' den Wert '0' im Debug Ausgabefenster aus. Wenn wir die Variable außerhalb der Prozedur in der sie definiert wurde, sehen und benutzen wollen, müssen wir sie global machen. Wie hier:

```
Procedure ZeigeWert()
  Global a.i = 10
EndProcedure

ZeigeWert()

Debug a
```

Wenn Sie diese letzten paar Beispiele durchgelesen haben, denken Sie vielleicht, warum definiert man nicht alle Variablen global? Das macht vielleicht für einige Personen Sinn, aber wenn Programme einen größeren Maßstab erreichen, können die Dinge durcheinander geraten oder verwirren wenn alle Variablen global sind. Sie werden auch merken, das Ihnen irgendwann die sinnvollen Variablennamen ausgehen. Wenn Sie mit getrennten Geltungsbereichen in Ihrem Programm arbeiten, haben Sie zum Beispiel die Möglichkeit temporäre Variablen für Berechnungen oder Schleifen gleich zu benennen, mit dem Wissen, dass die Variablen außerhalb dieses Geltungsbereiches nicht verändert werden. Manche Programmierer mühen sich richtig, so wenig globale Variablen wie möglich zu verwenden, da dies einen erheblich geringeren Aufwand

bei der Fehlersuche bedeutet. Verdächtige Werte und Variablen können viel schneller auf einen bestimmten Geltungsbereich heruntergebrochen werden wenn nur wenige Globale Variablen benutzt werden.

Wenn Sie Arrays und Listen mit Prozeduren verwenden, haben diese auch unterschiedliche Geltungsbereiche, wie die Variablen. Bis PureBasic Version 4 waren alle Arrays und Listen global, um sicherzustellen das sie alle durch Prozeduren bearbeitet werden können. Seit dem eintreffen von PureBasic Version 4 können Arrays und Listen Lokal, Global, Protected und Static sein, genau wie die Variablen. Sie benutzen die Schlüsselwörter zur Definition des Geltungsbereiches.

Im nächsten Abschnitt liste ich die Schlüsselwörter zur Definition der Geltungsbereiche auf und werde diese vollständig erklären. Für jedes wird es reichlich Beispiele geben, die Seine Verwendung mit Variablen, Listen und Arrays demonstrieren.

Das 'Global' Schlüsselwort

'Global' Variablen

Ich habe Ihnen bereits ein Beispiel für das 'Global' Schlüsselwort gegeben, während ich Ihnen die Geltungsbereiche erklärt habe aber hier ist es nochmal, um die Liste zu vervollständigen.

```
Global a.i = 10

Procedure ZeigeWert()
  Debug a
EndProcedure

ZeigeWert()
```

Das 'Global' Schlüsselwort wird vor einer Variablendefinition benutzt, um den Geltungsbereich der Variable global zu machen. Wenn eine Variable als Global definiert wurde, kann diese Variable von jeder in Ihrem Quelltext enthaltenen Prozedur gesehen und bearbeitet werden. Die Syntax ist sehr einfach, wie Sie in obigem Beispiel erkennen können.

'Global' Arrays

```
Global Dim Zahlen.l(1)

Procedure AendereWerte()
  Zahlen(0) = 3
  Zahlen(1) = 4
EndProcedure

AendereWerte()

Debug Zahlen(0)
Debug Zahlen(1)
```

In diesem Beispiel habe ich, ähnlich wie bei den Variablen, das 'Global' Schlüsselwort vor der Array Definition verwendet, weshalb es mir möglich war auf dieses aus der Prozedur heraus zuzugreifen. Ohne das 'Global' Schlüsselwort könnte die Prozedur das Array nicht sehen und auch nicht bearbeiten.

'Global' Listen

```
Global NewList Zahlen.l()

AddElement(Zahlen())
Zahlen() = 1

Procedure AendereWert()
  SelectElement(Zahlen(), 0)
  Zahlen() = 100
EndProcedure

AendereWert()

SelectElement(Zahlen(), 0)
Debug Zahlen()
```

In diesem Beispiel habe ich, ähnlich wie bei den Arrays, das 'Global' Schlüsselwort vor der Definition der Liste verwendet. So ist es mir möglich auf Diese aus der Prozedur heraus zuzugreifen. Ohne das 'Global' Schlüsselwort vor der Listendefinition würde die Prozedur die Liste weder sehen, noch bearbeiten können.

Das 'Protected' Schlüsselwort

'Protected' Variablen

Das 'Protected' Schlüsselwort zwingt eine Variable zu einem lokalen Dasein in der Prozedur, auch wenn der gleiche Variablenname im Haupt Quelltext schon einmal global deklariert wurde. Das ist sehr nützlich für die Definition von temporären Variablen in der Prozedur oder um sicherzustellen, das sich eine Prozedurvariable und eine globale Variable mit gleichem Namen niemals gegenseitig beeinflussen.

```
Global a.i = 10

Procedure AendereWert()
  Debug a
  Protected a.i = 20
  Debug a
EndProcedure

AendereWert()

Debug a
```

Wie Sie sehen können, hat die Variable in der Prozedur den gleichen Namen wie die Globale Variable. Es sind zwei unterschiedliche Variablen in verschiedenen Geltungsbereichen. Wenn Sie das obige Beispiel starten bekommen Sie drei Debug Ausgaben. Die erste Debug Ausgabe gibt den Wert '10' aus, da wir uns zwar in der Prozedur 'AendereWert()' befinden, aber noch keine 'Protected' Variablendefinition vorgenommen haben. Zu diesem Zeitpunkt hat die Globale Variable noch Gültigkeit. Nach der Variablendefinition gibt der Debug Befehl '20' zurück, da nun eine neue Variable mit anderem Geltungsbereich hinzugefügt wurde. Der Dritte Debug Befehl gibt wieder '10' aus, da wir uns nun außerhalb der Prozedur befinden und hier nur die Globale Variable gültig ist. Dieses Schlüsselwort ist essentiell, wenn Sie allgemeine Prozeduren schreiben möchten, die in vielen verschiedenen Quelltexten Verwendung finden und sicherstellen wollen, dass keine ungewünschten Beeinflussungen auftreten.

'Protected' Arrays

```
Global Dim Zahlen.l(1)

Procedure AendereWert()
  Protected Dim Zahlen.l(1)
  Zahlen(0) = 3
  Zahlen(1) = 4
EndProcedure

AendereWert()

Debug Zahlen(0)
Debug Zahlen(1)
```

In diesem Beispiel benutzen wir das 'Protected' Schlüsselwort auf die gleiche Weise wie im Variablen Beispiel. Wenn Sie obiges Beispiel starten, wird das Ergebnis im Debug Ausgabefenster '0' sein, da wir zwar die 'AendereWert()' Prozedur aufrufen, das darin enthaltene Protected Array aber nicht das Globale beeinflusst. Dieses Schlüsselwort ist großartig zum schützen von Arrays in Prozeduren, da es keine Konflikte gibt, auch wenn ein globales Array mit gleichem Namen existiert.

'Protected' Listen

```
Global NewList Zahlen.l()
AddElement(Zahlen())
Zahlen() = 1

Procedure AendereWert()
  Protected NewList Zahlen.l()
  AddElement(Zahlen())
  Zahlen() = 100
EndProcedure
```

```
AendereWert()

SelectElement(Zahlen(), 0)
Debug Zahlen()
```

Auch in diesem Beispiel wird der ausgegebene Wert im Debug Ausgabefenster '1' sein, da wir zwar wieder die 'AendereWert()' Prozedur aufrufen, das darin enthaltene Array aber nicht das Globale beeinflusst. Dieses Schlüsselwort ist großartig zum schützen von Listen in Prozeduren, da es keine Konflikte gibt, auch wenn eine globale Liste mit gleichem Namen existiert.

Das 'Shared' Schlüsselwort

'Shared' Variablen

Manchmal müssen Sie vielleicht aus einer Prozedur heraus auf eine externe Variable zugreifen, die nicht als Global definiert wurde. Dies ist mit dem 'Shared' Schlüsselwort möglich. Hier ein Beispiel:

```
a.i = 10

Procedure AendereWert()
  Shared a
  a = 50
EndProcedure

AendereWert()

Debug a
```

Hier können Sie, obwohl die Original Variable nicht als Global definiert wurde, trotzdem innerhalb der Prozedur auf die Variable zugreifen. Dies ist durch das 'Shared' Schlüsselwort möglich. Wenn das obige Beispiel gestartet wird, ändert die 'AendereWert()' Prozedur den Wert von 'a' obwohl es keine Globale Variable ist.

'Shared' Arrays

```
Dim Zahlen.l(1)

Procedure AendereWert()
  Shared Zahlen()
  Zahlen(0) = 3
  Zahlen(1) = 4
EndProcedure

AendereWert()

Debug Zahlen(0)
Debug Zahlen(1)
```

In diesem Beispiel kann ich mittels des 'Shared' Schlüsselwortes innerhalb der Prozedur auf das Array zugreifen, obwohl dieses nicht als Global definiert wurde. Wenn Sie angeben, welches Array Sie bereichsübergreifend teilen wollen, müssen Sie den Namen des Arrays mit angefügten Klammern angeben, wie hier: 'Zahlen()'. Es müssen keine Suffixe für den Typ, Indizes oder Dimensionen angegeben werden.

'Shared' Listen

```
NewList Zahlen.l()

Procedure AendereWert()
  Shared Zahlen()
  AddElement(Zahlen())
  Zahlen() = 100
EndProcedure

AendereWert()

SelectElement(Zahlen(), 0)
Debug Zahlen()
```

In diesem Beispiel kann ich mittels des 'Shared' Schlüsselwortes innerhalb der Prozedur auf die Liste zugreifen, obwohl diese nicht als Global definiert wurde, sehr ähnlich wie im Array Beispiel. Wenn Sie angeben, welche Liste Sie bereichsübergreifend teilen wollen, müssen Sie den Namen der Liste mit angefügten Klammern angeben, wie hier: 'Zahlen()'. Es muss kein Suffix für den Typ angegeben werden.

Das 'Static' Schlüsselwort

'Static' Variablen

Jedes Mal wenn eine Prozedur beendet wird, gehen die Werte der darin definierten Variablen verloren. Wenn Sie erreichen möchten, das Variablen ihren Wert nach jedem Prozeduraufruf behalten sollen, müssen Sie das 'Static' Schlüsselwort verwenden. Schauen Sie auf dieses Beispiel:

```
Procedure AendereWert()
  Static a.i
  a + 1
  Debug a
EndProcedure

For x.i = 1 To 5
  AendereWert()
Next x
```

Hier habe ich in der 'AendereWert()' Prozedur die Variable 'a' mittels des 'Static' Schlüsselwortes als statisch definiert. Danach habe ich den Wert der Variable 'a' um '1' erhöht und habe deren Wert im Debug Ausgabefenster ausgegeben. Dann habe ich diese Prozedur mit einer 'For' Schleife fünf mal aufgerufen. Wenn Sie die ausgegebenen Werte betrachten, fällt Ihnen auf, das sie alle unterschiedlich und um Eins erhöht sind. Das ist so, weil der Variablenwert zwischen den Prozeduraufrufen gespeichert wurde.

'Static' Arrays

Sie können mit dem 'Static' Schlüsselwort auch Array Werte zwischen Prozeduraufrufen intakt halten.

```
Procedure AendereWert()
  Static Dim Zahlen.l(1)
  Zahlen(0) + 1
  Zahlen(1) + 1
  Debug Zahlen(0)
  Debug Zahlen(1)
EndProcedure

For x.i = 1 To 5
  AendereWert()
Next x
```

In obigem Beispiel habe ich das 'Static' Schlüsselwort verwendet um die Array Werte zwischen den Prozeduraufrufen zu bewahren, genau wie bei den statischen Variablen. Wenn Sie die ausgegebenen Werte betrachten, sehen Sie dass alle unterschiedlich und um Eins erhöht sind. Das ist so, weil alle Array Werte zwischen den Prozeduraufrufen bewahrt wurden.

'Static' Listen

```
Procedure AendereWert()
  Static NewList Zahlen.l()
  If ListSize(Zahlen()) = 0
    AddElement(Zahlen())
  EndIf
  SelectElement(Zahlen(), 0)
  Zahlen() + 1
  Debug Zahlen()
EndProcedure

For x.i = 1 To 5
  AendereWert()
Next x
```

In diesem Beispiel habe ich das 'Static' Schlüsselwort auf die gleiche Weise wie bei den Arrays verwendet, um die Werte in der Liste zwischen den Prozeduraufrufen zu bewahren. Wenn Sie die ausgegebenen Werte betrachten, sehen Sie dass alle unterschiedlich und um Eins erhöht sind. Da diese Liste statisch ist und ich

in diesem Beispiel nur ein Element hinzufügen möchte, habe ich eine 'If' Anweisung eingebaut, um die Anzahl der in der Liste enthaltenen Elemente zu prüfen. Wenn die Liste kein Element enthält, füge ich ihr eines hinzu, anderenfalls ändere ich nur den Wert des existierenden Elementes in der statischen Liste. Wenn ich diese 'If' Anweisung nicht eingebaut hätte, würde ich bei jedem Prozeduraufruf der Liste ein neues Element hinzufügen, was aber nicht sein soll.

Übergabe von Variablen an Prozeduren

Wie ich bereits vorher schon erwähnt habe, haben Prozeduren die herausragende Eigenschaft, das sie Initiale Startparameter akzeptieren. Diese Parameter können Variablen von jedem eingebauten Typ, Arrays und Listen sein. Parameter werden als eine Möglichkeit genutzt, Werte vom Hauptprogramm in eine Prozedur zu schleusen, um sie dort zu verarbeiten. Die Prozeduren können dann immer wieder aus dem Programm heraus aufgerufen werden, auch mit wechselnden Startwerten. Hier sehen Sie, wie Sie eine Prozedur definieren müssen, die Parameter akzeptiert:

```
Procedure ZaehleZusammen(a.i, b.i)
  Debug a + b
EndProcedure

ZaehleZusammen(10,  5)
ZaehleZusammen( 7,  7)
ZaehleZusammen(50, 50)
```

Alle Parameter die an die Prozedur übergeben werden müssen innerhalb der Klammern stehen. Bei mehreren zu übergebenden Parametern müssen diese durch Kommata separiert werden. Wenn die Parameter in der Definition festgelegt werden, muss zu jedem Parameter auch der entsprechende Typ angegeben werden. Hier habe ich zwei Parameter definiert ('a' und 'b'), die beide vom Typ Integer sind.

Wenn Sie eine Prozedur mit Parametern definiert haben, müssen Sie darauf achten dass Sie beim Aufruf die entsprechenden Werte mit übergeben, wie hier:

```
ZaehleZusammen(10, 5)
```

Nach diesem besonderen Aufruf wird der Wert '10' in die Variable 'a' und der Wert '5' in die Variable 'b' geschoben. Die Prozedur addiert dann diese beiden Variablen und zeigt das Ergebnis im Debug Ausgabefenster an. Das passiert mit allen anderen Aufrufen genauso, der erste Parameter Wert wird in 'a' und der Zweite in 'b' geschoben.

Hier ein weiteres Beispiel, das Strings verwendet:

```
Procedure VerbindeString(a.s, b.s)
  Debug a + b
EndProcedure

VerbindeString("Alle meine Entchen ", "schwimmen auf dem See,")
VerbindeString("schwimmen auf ", "dem See,")
VerbindeString("Köpfchen in das Wasser, ", "Schwänzchen in die Höh'.")
```

Hier nutzen wir die zweite Verwendungsmöglichkeit des '+' Operators, wir benutzen ihn zur Verkettung von Strings statt zur Addition von Zahlen. Ich habe einfach die 'ZaehleZusammen(a.i, b.i)' Prozedur modifiziert. Ich habe die beiden Parameter vom Typ Integer durch String Typen ersetzt und den Namen in 'VerbindeString(a.s, b.s)' geändert. Nun kann ich die Prozedur von jeder Stelle in meinem Programm aufrufen und ihr zu den Parametern passende Strings übergeben.

Ein weiterer Punkt, den Sie sich hinsichtlich Prozeduren merken sollten ist, das die Parameter nicht alle den selben Typ haben müssen. Sie können so viele verschiedene Typen verwenden, wie sie wollen.

```
Procedure VieleTypen(a.b, b.w, c.l, d.f, e.s)
  Debug "Byte: "   + Str(a)
  Debug "Word: "   + Str(b)
  Debug "Long: "   + Str(c)
  Debug "Float: "  + StrF(d)
  Debug "String: " + e
EndProcedure

VieleTypen(104, 1973, 97987897, 3.1415927, "Alle meine Entchen")
```

Hier habe ich ein paar verschiedene Variablentypen als Parameter in meine 'VieleTypen()' Prozedur gepackt, um zu demonstrieren dass Parameter nicht alle den gleichen Typ haben müssen. Die der Prozedur übergebenen Werte werden den definierten Prozedurparametern zugewiesen ('a', 'b', 'c', 'd' und 'e'). Beachten Sie, dass die übergebenen Werte zu den entsprechenden Parametertypen passen müssen, sonst gibt es eine Fehlermeldung, die einen falschen Parameter meldet.

Sie haben wahrscheinlich auch gemerkt, das ich zwei in PureBasic eingebaute Funktionen verwendet habe, mit denen Sie noch nicht vertraut sind. Diese beiden Funktionen, 'Str()' und 'StrF()' werden detaillierter in Kapitel 7 (Beispiele für gängige Befehle) beschrieben. Auf die Schnelle erklärt: sie konvertieren numerische Typen in Strings.

Übergaberichtlinien von Werten an Prozeduren

Wenn Sie eine Prozedur aufrufen, müssen Sie sich vergewissern, das alle definierten Parameter mit dem richtigen Typ bedient werden, sonst bekommen Sie einen Syntax Fehler bei der Kompilierung. Wenn eine Prozedur zum Beispiel folgendermaßen definiert ist:

```
ZaehleZusammen(a.i, b.i)
```

dann dürfen nur Integer Werte als Parameter übergeben werden. Wenn als Parameter String Variablen definiert sind, dann dürfen Sie der Prozedur beim Aufruf auch nur Strings übergeben. Prozeduren die Arrays und Listen als Parameter verwenden müssen ebenfalls ordnungsgemäß aufgerufen werden. Alle Parameter müssen die richtige Reihenfolge und den richtigen Typ haben. Ich weiß, das scheint alles klar zu sein, aber ich denke es ist kein Fehler es noch einmal zu betonen.

Gerade durch diese einfachen Beispiele sollten die Stärken von Prozeduren offensichtlich werden. Sie sind nicht nur großartige Zeitsparer beim tippen von Code, sondern sie erlauben es die Funktionalität von PureBasic zu erweitern und geben die Möglichkeit diese erweiterte Funktionalität in jedem neuen Quelltext weiter zu verwenden. Dies vereinfacht auf großartige Weise die Wiederverwendung von Code, was sehr interessant ist, wenn Sie ein paar sehr hilfreiche und nützliche Prozeduren geschrieben haben.

Optionale Parameter

Ein neues Merkmal von PureBasic Version 4 war die Möglichkeit in Prozeduren optionale Parameter zu definieren. Diese sind sehr einfach zu erklären und zu demonstrieren, und sie sind extrem nützlich.

Grundlegend können Sie einem Parameter einen Initialwert zuweisen, der dann verwendet wird, wenn beim Prozeduraufruf für diesen Parameter kein Wert übergeben wird. Hier ein Beispiel:

```
Procedure ZeigeParameter(a.i, b.i, c.i = 3)
  Debug a
  Debug b
  Debug c
EndProcedure

ZeigeParameter(1, 2)
```

Wenn Sie in die Prozedur Definition in obigem Beispiel schauen, sehen Sie dass der letzte Parameter anders definiert wurde. Ihm wurde der Initialwert '3' zugewiesen. Wenn für den letzten Parameter beim Prozeduraufruf kein Wert übergeben wird, dann hat er diesen Standardwert. Wenn ich die Prozedur in obigem Beispiel folgendermaßen aufrufe: 'ZeigeParameter(1, 2)' und den letzten Parameter auslasse, dann wird der Standardwert verwendet und im Debug Ausgabefenster angezeigt.

Eine beliebige Anzahl von Parametern kann auf diese Weise optional definiert werden. Sie müssen nur darauf achten, das alle optionalen Parameter in der Definition rechts von den nicht optionalen stehen. Das ist so, weil alle Parameter die an eine Prozedur übergeben werden von links nach rechts zugewiesen werden und die optionalen können nicht übersprungen werden.

Übergabe von Arrays an Prozeduren

Wie Variablen, können Sie auch Arrays und Listen als Parameter an Prozeduren übergeben. Die Verwendung ist ebenfalls sehr einfach und schnell zu demonstrieren.

Um ein Array als Parameter übergeben zu können, müssen Sie zuerst ein Array mit dem 'Dim' Schlüsselwort definieren. Hier ist ein einfaches Eindimensionales Array in dem fünf Indizes definiert sind (erinnern Sie sich, das Indizes mit '0' beginnen). Ich habe es 'Kontinente' genannt:

```
Dim Kontinente.s(4)

Kontinente(O) = "Europa"
Kontinente(1) = "Afrika"
Kontinente(2) = "Asien"
Kontinente(3) = "Ozeanien"
Kontinente(4) = "Amerika"
```

Da wir das Array nun definiert haben, lassen Sie uns eine Prozedur definieren, der wir es übergeben:

```
Procedure ZeigeArray(Array MeinArray.s(1))
  For x.i = 0 To 4
    Debug MeinArray(x)
  Next x
EndProcedure
```

Um eine Prozedur so zu definieren, dass sie ein Array als Parameter akzeptiert, müssen Sie sich streng an die Regeln halten. Für Anfänger, Sie definieren ein Array als Parameter genau so wie jeden anderen Parameter innerhalb der Prozedur Klammern. Der eigentliche Array Parameter setzt sich aus vier Teilen zusammen.

Zuerst müssen wir definieren, das es sich bei dem Parameter um ein Array handelt. Dies erledigen wir mit dem 'Array' Schlüsselwort. Darauf folgt (getrennt durch ein Leerzeichen) der Name, der als Arrayname innerhalb der Prozedur verwendet wird. Dann folgt ein Suffix für den Arraytyp mit anschließenden Klammern. In diesen Klammern steht üblicherweise eine Zahl, die den Arrayindex repräsentiert. In diesem Fall wird aber die Anzahl der Dimensionen des übergebenen Arrays eingetragen. In obigem Beispiel habe ich ein Eindimensionales Array verwendet, deshalb sieht der vollständige Array Parameter so aus:

```
Array MeinArray.s(1)
```

Hier sehen Sie einen Array Parameter Namen, das Typ Suffix des Arrays und zwischen den Klammern die Anzahl der definierten Dimensionen im erwarteten Array.

Sie werden bemerkt habe, das ich als Array Parameter Namen 'MeinArray' verwendet habe. Das muss nicht so sein, Sie können Ihre Parameter nennen wie Sie wollen. Sie müssen nur die gleichen Namensregeln wie bei den Variablen einhalten, wie z.B. keine Leerzeichen, keine Schlüsselwort Namen, usw.

Innerhalb der Prozedur können Sie nun auf 'MeinArray()' ganz normal zugreifen, so wie Sie es mit jedem anderen Array machen. Sie können Werte zuweisen und auslesen indem Sie die Standard Indizes verwenden. In obigem Beispiel habe ich eine einfache 'For' Schleife benutzt um mich durch das Array zu bewegen und die Werte im Debug Ausgabefenster anzuzeigen. Sie können sich aber auch eigene kreative Ideen zum verarbeiten von Arrays ausdenken.

Wenn die Prozedur richtig definiert ist, können Sie sie aufrufen und ihr das 'Kontinente' Array übergeben, wie hier:

```
ZeigeArray(Kontinente())
```

Beachten Sie, wenn wir das Array 'Kontinente()' übergeben, werden nur die Klammern an den Arraynamen angehängt, fast wie beim Prozeduraufruf selbst. Es werden keine Typen, keine Indizes und keine Dimensionen beim Prozeduraufruf übergeben.

Wenn Sie dieses Beispiel starten, sollten Sie die fünf Kontinente der Erde im Debug Ausgabefenster angezeigt bekommen.

Übergabe von Mehrdimensionalen Arrays

Sie können auch Mehrdimensionale Arrays als Parameter übergeben, wie dieses Beispiel zeigt:

```
Dim Kontinente.s(4, 1)

Kontinente(0, 0) = "Europa"
Kontinente(0, 1) = "738 Millionen"
```

```
Kontinente(1, 0) = "Afrika"
Kontinente(1, 1) = "999 Millionen"

Kontinente(2, 0) = "Asien"
Kontinente(2, 1) = "4117 Millionen"

Kontinente(3, 0) = "Ozeanien"
Kontinente(3, 1) = "36 Millionen"

Kontinente(4, 0) = "Amerika"
Kontinente(4, 1) = "920 Millionen"

Procedure ZeigeArray(Array MeinArray.s(2))
  For x.i = 0 To 4
    Debug MeinArray(x, 0) + "  -  " + "Einwohner: " + MeinArray(x, 1)
  Next x
EndProcedure

Debug "Weltbevölkerung, Stand: August 2009"
ZeigeArray(Kontinente())
```

Obwohl es sich hier um ein Zweidimensionales Array handelt, verwenden wir trotzdem die gleichen Regeln bei der Parameterdefinition, die wir auch bei den Eindimensionalen Arrays verwendet haben. Sie werden bemerkt haben, dass wir in dieser Array Parameterdefinition den Wert '2' innerhalb der Klammern angegeben haben. Damit definieren wir, dass das zu erwartende Array zwei Dimensionen hat, wie hier:

```
Array MeinArray.s(2)
```

Auch wenn der Parameter für ein Zweidimensionales Array definiert ist, erfolgt der Prozeduraufruf mit Übergabe des Arrays wie zuvor ohne Typen, Indizes oder Dimensionen.

```
ZeigeArray(Kontinente())
```

Übergabe von Strukturierten Arrays

Strukturierte Arrays können ebenfalls sehr einfach an Prozeduren übergeben werden. Sie erledigen das exakt so, wie es auf der vorherigen Seite dargestellt wurde. Sie müssen lediglich den Array Parametertyp so abändern, das er zur Struktur Definition des verwendeten Arrays passt. Schauen Sie auf dieses Beispiel:

```
Structure KONTINENT
  Name.s
  Einwohner.s
EndStructure

Dim Kontinente.KONTINENT(4)

Kontinente(0)\Name      = "Europa"
Kontinente(0)\Einwohner = "738 Millionen"

Kontinente(1)\Name      = "Afrika"
Kontinente(1)\Einwohner = "999 Millionen"

Kontinente(2)\Name      = "Asien"
Kontinente(2)\Einwohner = "4117 Millionen"

Kontinente(3)\Name      = "Ozeanien"
Kontinente(3)\Einwohner = "36 Millionen"

Kontinente(4)\Name      = "Amerika"
Kontinente(4)\Einwohner = "920 Millionen"

Procedure ZeigeArray(Array MeinArray.KONTINENT(1))
  For x.i = 0 To 4
    Debug MeinArray(x)\Name + "  -  " + "Einwohner: " + MeinArray(x)\Einwohner
  Next x
EndProcedure

Debug "Weltbevölkerung, Stand: August 2009"
ZeigeArray(Kontinente())
```

Hier habe ich ein strukturiertes Array verwendet, das mit der 'KONTINENT' Struktur definiert wurde. Um diesen Typ von Array zu übergeben, müssen Sie darauf achten, das Sie bei der Definition des Array Parameters die gleiche Struktur verwenden, die Sie beim definieren des erwarteten Arrays verwendet haben. In diesem Fall war es die 'KONTINENT' Struktur. Sie sehen den Parameter in der obigen Prozedur Definition:

```
Array MeinArray.KONTINENT(1)
```

Dieser zeigt das Array Schlüsselwort, den Parameternamen, das Typ Suffix des strukturierten Arrays und die Anzahl der Dimensionen des übergebenen Arrays. Wenn all dies korrekt programmiert wurde, dann wird ein Eindimensionales, 'KONTINENT' strukturiertes Array an die Prozedur übergeben.

Übergabe von Listen an Prozeduren

Wenn Sie mit PureBasic programmieren, möchten Sie vielleicht auch einmal eine Liste an eine Prozedur übergeben. Das ähnelt stark der Array Übergabe. Schauen Sie zuerst auf dieses Beispiel:

```
NewList Zahlen.l()

AddElement(Zahlen())
Zahlen() = 25

AddElement(Zahlen())
Zahlen() = 50

AddElement(Zahlen())
Zahlen() = 75

Procedure ZeigeListe(List MeineListe.l())
  ForEach MeineListe()
    Debug MeineListe()
  Next
EndProcedure

ZeigeListe(Zahlen())
```

Hier habe ich eine Standard Liste mit dem Namen 'Zahlen' erstellt, die einige Elemente vom Typ Long enthält. Nachdem ich der Liste drei Elemente hinzugefügt habe, habe ich die 'ZeigeListe()' Prozedur definiert, der ich diese Liste übergeben will. Der Listen Parameter wird so ähnlich wie der Array Parameter definiert. Er beginnt mit dem 'List' Schlüsselwort. Dann folgt getrennt durch ein Leerzeichen der Parameter Name, mit anschließendem Typ Suffix. Den Abschluss bildet ein Satz Klammern. Listen haben keine Indizes, weshalb Sie keinen Wert in die Klammern schreiben müssen. Wenn das alles getan ist, können Sie die Prozedur aufrufen und die Liste wie ein Array übergeben, ohne einen Typ, usw. Wie hier:

```
ZeigeListe(Zahlen())
```

Natürlich enthält die Liste 'MeineListe()' innerhalb der 'ZeigeListe()' Prozedur, alle Informationen die in der übergebenen Liste enthalten waren und kann wie jede andere Standard Liste verwendet werden. In der 'ZeigeListe()' Prozedur habe ich eine 'ForEach' Schleife verwendet, die sich schnell durch die Liste arbeitet und die enthaltenen Werte im Debug Ausgabefenster anzeigt.

Übergabe von Strukturierten Listen

Strukturierte Listen können ebenfalls sehr einfach als Parameter an eine Prozedur übergeben werden. Auch hier müssen Sie wieder darauf achten, dass (wie bei der Übergabe von strukturierten Arrays an Prozeduren) die Struktur, die im Prozedur Parameter definiert wird, die gleiche ist, die zum definieren der erwarteten Liste verwendet wurde.

```
Structure FLAGGEN
  Land.s
  Flagge.s
EndStructure

NewList Info.FLAGGEN()

AddElement(Info())
Info()\Land   = "Großbritannien"
Info()\Flagge = "Union Jack"
```

```
AddElement(Info())
Info()\Land   = "USA"
Info()\Flagge = "Stars And Stripes"

AddElement(Info())
Info()\Land   = "Frankreich"
Info()\Flagge = "Trikolore"

Procedure ZeigeListe(List MeineListe.FLAGGEN())
  ForEach MeineListe()
    Debug MeineListe()\Land + "'s Flagge: " + MeineListe()\Flagge
  Next
EndProcedure

ZeigeListe(Info())
```

Wie Sie obigem Beispiel entnehmen können, sieht der Listen Parameter so aus:

```
List MeineListe.FLAGGEN()
```

Ich habe mich vergewissert, dass der Listen Parameter nun den Typ 'FLAGGEN' enthält, damit die strukturierte Liste korrekt übergeben wird.

Arrays und Listen werden per Referenz übergeben

Anders als Variablen, werden Arrays und Listen per Referenz übergeben. Das bedeutet, dass die Werte des Arrays oder der Liste nicht in die Prozedur überführt werden und dort als Kopie vorgehalten werden. Stattdessen wird ein interner Zeiger übergeben, der zum manipulieren des Arrays oder der Liste verwendet wird.

Das passiert alles intern, so das Sie keinen Unterschied zur Verwendung von Variablen bemerken. Das einzige, was Sie berücksichtigen müssen, ist die Tatsache, dass beim übergeben eines Arrays oder einer Liste keine lokale Kopie in der Prozedur existiert. Egal welchen Parameternamen Sie dem Array oder der Liste geben, Sie verändern immer das übergebene Original des Arrays oder der Liste.

Wenn Sie zum Beispiel ein Array mit dem Namen 'Hunde()' als Parameter an eine Prozedur übergeben, dann kann dieser Parameter einen anderen Namen haben, zum Beispiel 'Katzen()'. Innerhalb der Prozedur können Sie dem Array 'Katzen()' verschiedene Werte zuweisen. Wenn Sie die Prozedur verlassen, werden Sie feststellen, dass das Array 'Hunde()' auf die gleiche Weise verändert wurde, wie Sie das Array 'Katzen()' verändert haben. Das ist so, weil der 'Katzen()' Parameter nur eine Referenz zum Original übergebenen 'Hunde()' Array ist.

Das trifft nur für Arrays und Listen zu, Variablen werden ganz normal als Wert übergeben, das heißt, ihr aktueller Wert wird in den Variablen Parameter kopiert. Wenn Sie den Variablen Parameter in der Prozedur manipulieren, hat das keinen Einfluss auf die Original Variable, die Sie übergeben haben.

Rückgabe eines Wertes aus Prozeduren

Eine weitere herausragende Fähigkeit von Prozeduren ist die Möglichkeit Werte zurückzugeben. Dieser Wert kann jeden in PureBasic eingebauten Typ haben. Die Rückgabe von Werten sind eine gute Möglichkeit, Daten zu berechnen oder zu manipulieren und dann etwas Sinnvolles zurückzugeben. Das kann ein Berechnungsergebnis sein, eine Fehlernummer oder auch ein beschreibender String, was die Prozedur gerade getan oder erreicht hat. Schauen Sie auf Abb. 2 und Abb. 3 für eine Liste der Typen, die von einer Prozedur zurückgegeben werden können.

Die Syntax zur Rückgabe von Werten aus einer Prozedur ist ebenfalls sehr einfach. Zuerst müssen Sie definieren, welchen Typ die Prozedur zurückgeben soll. Dies geschieht durch eine kleine Erweiterung der Prozedur Definition. Schauen Sie auf dieses Beispiel:

```
Procedure.i ZaehleZusammen(a.i, b.i)
  ProcedureReturn a + b
EndProcedure

Debug ZaehleZusammen(7, 5)
```

Der obigen Prozedurdefinition wurde ein neues Suffix am Ende des 'Procedure' Schlüsselwortes hinzugefügt. Das ist die Stelle, an der Sie definieren, welchen Typ die Prozedur zurückgeben wird. Der obigen Definition wurde das '.i' Suffix hinzugefügt, was bedeutet, dass die Prozedur einen Integer Wert zurückgeben wird. Damit die Prozedur einen Wert zurückgibt, müssen wir das 'ProcedureReturn' Schlüsselwort verwenden. Der Wert oder Ausdruck der diesem Schlüsselwort folgt wird von der Prozedur zurückgegeben. In diesem Fall wird das Ergebnis der Berechnung von 'a + b' zurückgegeben.

Sie können ebenfalls sehen, dass wenn wir obige Prozedur mit 'ZaehleZusammen(7, 5)' aufrufen, direkt ein Wert zurückgegeben wird, der selbst wieder in einem Ausdruck oder einer Anweisung verwendet werden kann. In diesem Fall habe ich ihn als Ausdruck für die 'Debug' Anweisung verwendet. Der Wert '12' wird vom Prozeduraufruf zurückgegeben und dann direkt mittels des 'Debug' Schlüsselwortes im Debug Ausgabefenster angezeigt.

Wenn Sie die Prozedur dazu bringen wollen einen String zurückzugeben, müssen Sie den Rückgabetyp in '.s' ändern, wie hier gezeigt:

```
Procedure.s VerbindeString(a.s, b.s)
  ProcedureReturn a + b
EndProcedure

Debug VerbindeString("Roter ", "Lastwagen")
Debug VerbindeString("Gelber ", "Lastwagen")
```

Beachten Sie, das der Rückgabetyp im 'Procedure.s' Schlüsselwort spezifiziert wurde. Dies legt fest, das die Prozedur einen Stringwert zurückgibt. In diesem Beispiel habe ich der Prozedur zwei Strings übergeben, die dann von der Zeile 'ProcedureReturn a + b' miteinander verbunden, und anschließend als ein String zurückgegeben wurden. Ich rufe diese Prozedur zweimal mit dem 'Debug' Schlüsselwort auf um die zurückgegebenen Werte im Debug Ausgabefenster anzuzeigen und die zurückgegebenen Werte zu demonstrieren.

Prozeduren die Werte zurückgeben, können an jeder Stelle an der ein Ausdruck stehen kann verwendet werden. Sie können zum Beispiel in einem weiteren Prozeduraufruf stehen, wie hier:

```
Procedure.i ZaehleZusammen(a.i, b.i)
  ProcedureReturn a + b
EndProcedure

Debug ZaehleZusammen(ZaehleZusammen(2, 3), ZaehleZusammen(4, 1))
```

Hier habe ich die 'ZaehleZusammen(a.i, b.i)' Prozedur, verwendet um Werte zurückzugeben, die von einem separaten Aufruf der gleichen Prozedur addiert und als Gesamtergebnis zurückgegeben werden. Zuerst wird die Prozedur 'ZaehleZusammen(2, 3)' aufgerufen, die '5' zurückgibt. Danach wird 'ZaehleZusammen(4, 1)' aufgerufen, die eben falls '5' zurückliefert. Beide Prozeduraufrufe wurden aber als Parameter an eine dritte Prozedur übergeben. Diese addiert letztendlich die beiden zurückgegebenen Werte und gibt den Wert zurück, der im Debug Ausgabefenster erscheint.

Parameter beeinträchtigen nicht den Rückgabe Typ

Wenn Sie Rückgabetypen mit Prozeduren verwenden, sollten Sie sich eine Sache merken: Dieser muss nicht der Gleiche sein wie der, den Sie zur Parameterdefinition verwendet haben. Zum Beispiel:

```
Procedure.s ZeigeWert(a.i, b.s)
  ProcedureReturn Str(a) + b
EndProcedure

x.i = 5

While x >= 0
  Debug ZeigeWert(x, " Grüne Flaschen hängen an der Wand.")
  x - 1
Wend
```

Obwohl ich einen Integer Typ und einen String Typ als Parameter in der Prozedur 'ZeigeWert(a.i, b.s)' verwendet habe, kann diese trotzdem einen String zurückgeben, so wie es durch das 'Procedure.s' Schlüsselwort definiert wurde. Rückgabe Typen und Parameter Typen können Sie beliebig mischen.

Die Regeln für Rückgabe Typen sind genauso streng wie die für Variablen. Wenn Sie eine Prozedur so definiert haben, dass sie einen String zurückgibt und Sie wollen einen Integer Wert zurückgeben, dann müssen Sie den Wert zuerst in den richtigen Typ (String) wandeln, bevor Sie ihn zurückgeben können sonst bekommen Sie einen Syntax Fehler. Dies können Sie in obigem Beispiel sehen. Ich habe den eingebauten Befehl 'Str()' verwendet um 'a' in einen String zu wandeln, bevor ich die beiden Werte verbunden und zurückgegeben habe.

Einschränkungen von Rückgabewerten

Bei der Verwendung von Prozeduren um Ergebnisse zurückzugeben gibt es zwei Einschränkungen. Erstens, Prozeduren können nur einen Wert pro Aufruf zurückgeben. Das heißt, Sie können nicht zwei oder mehr Werte mit einem Prozeduraufruf zurückgeben. Zweitens, Sie können keine Arrays, Listen oder benutzerdefinierte Typen zurückgeben. Sie können nur die in PureBasic eingebauten Typen zurückgeben. Es ist möglich diese Einschränkungen mittels Rückgabe eines Zeigers zu umgehen. Siehe Kapitel 13 (Zeiger) für eine Erklärung dieser Technik.

7. Verwendung der eingebauten Befehle

Programmiersprachen sind nichts ohne die mit nützlichen Befehlen gefüllten Bibliotheken. So verfügt PureBasic über mehr als achthundert eingebaute Befehle die Sie in Ihren Programmen verwenden können. Diese reichen von String Manipulation, über mathematische Funktionen, Dateiverarbeitung bis zu grafischen Befehlen. Sie decken nahezu jede denkbare Aufgabenstellung, und für den Fall, das Sie einmal nicht fündig werden, haben Sie die Möglichkeit eigene Prozeduren zu schreiben.

In diesem Kapitel werde ich Ihnen die meist genutzten eingebauten Befehle, die PureBasic zu bieten hat, vorstellen und erklären. Da einige meinen, die Liste sei nicht komplett - diese Einführung will nur die gängigsten Befehle ausführlich erklären, die Ihnen bei Ihrer täglichen Arbeit begegnen.

Dieses Kapitel beginnt mit einer Beschreibung, wie Sie die Befehls-Syntax in der PureBasic Hilfe lesen und geht dann über zu den Befehlsbeschreibungen und Erklärungen. Ich beende das Kapitel mit einem Abschnitt über Dateiverarbeitung, wie Sie Dateien laden, Lese- und Schreiboperationen durchführen, sowie Dateien speichern. All dies geschieht mit den in PureBasic eingebauten Befehlen.

Benutzen der PureBasic Hilfe

Hier werde ich einen gründlichen Blick in die PureBasic Hilfe werfen und erklären, wie die Hilfe geordnet ist, wie Sie sie am besten lesen, und was am wichtigsten ist, wie Sie die dort beschriebenen eingebauten Befehle benutzen.

Wenn Sie eine Befehls-Seite in der PureBasic Hilfe öffnen werden Sie mit einem Standardlayout konfrontiert, das sich folgendermaßen zusammensetzt:

Syntax
Beschreibung
Ein Beispiel
Unterstützte Betriebssysteme

Die letzten drei Abschnitte jeder Hilfeseite sind weitestgehend selbsterklärend, also lassen Sie uns eine Minute im Syntax Abschnitt verweilen. Der Syntax Teil im oberen Abschnitt auf jeder Hilfeseite ist ein Beispiel dafür, wie der Befehl in der IDE eingefügt werden muss und welche Parameter er erwartet. Lassen Sie uns zum Beispiel einen Blick auf die Hilfeseite des 'SaveImage()' Befehls werfen. So navigieren Sie zur richtigen Seite: (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Image → SaveImage). Auf dieser Seite, ganz oben unter der Syntax Überschrift erscheint folgendes Syntax Beispiel:

```
Ergebnis = SaveImage(#Image, DateiName$ [, ImagePlugin [, Flags [, Tiefe]]])
```

Das erste was Sie sehen ist 'Ergebnis =', was bedeutet, das dieser Befehl einen numerischen Rückgabewert hat. Dann folgt der eigentliche Befehlsname, in diesem Fall 'SaveImage', mit einem angehängten Satz Klammern und wenn der bestimmte Befehl Parameter akzeptiert, werden diese innerhalb der Klammern angezeigt. Das 'SaveImage()' Syntax Beispiel zeigt, das dieser Befehl fünf Parameter akzeptieren kann, weshalb alle fünf innerhalb der Klammern stehen. Sie werden auch bemerkt haben, das die letzten drei Parameter in eckigen Klammern stehen. Dies zeigt an, das diese drei Parameter optional sind und nicht zwingend übergeben werden müssen, wenn Sie den 'SaveImage()' Befehl aufrufen. Wenn wir das im Hinterkopf behalten, kann der 'SaveImage()' Befehl zweckmäßig so verwendet werden, wie es in diesen vier Beispielen dargestellt wird:

```
SaveImage(1, "Bild.bmp")
SaveImage(1, "Bild.jpg", #PB_ImagePlugin_JPEG)
SaveImage(1, "Bild.jpg", #PB_ImagePlugin_JPEG2000, 10)
SaveImage(1, "Bild.jpg", #PB_ImagePlugin_JPEG, 0, 24)
```

Das erste Beispiel wird ein Bild mit Namen 'Bild.bmp' im Standard 24-Bit Bitmap Format abspeichern. Das zweite Beispiel wird ein Bild im JPEG Format mit der Standardkompression abspeichern. Das dritte Beispiel speichert ein Bild im JPEG2000 Format in der höchsten Qualitätsstufe ('10') ab. Das vierte Beispiel speichert ein Bild im JPEG Format mit der höchsten Kompression ('0' = niedrigste Qualität) und einer Farbtiefe von 24-Bit ab. Ganz einfach!

Was bedeuten die eckigen Klammern in den Beispielen?

Die eckigen Klammern die im Syntax Beispiel zu sehen sind werden niemals eingegeben, wenn Sie den Befehl benutzen. Sie werden nur in die Beispiele eingefügt, um zu zeigen, welche Parameter (wenn es welche gibt) optional sind, wenn Sie diesen entsprechenden Befehl verwenden. Eckige Klammern werden nur dann in einem Quelltext eingegeben, wenn Sie Statische Arrays verwenden. Für mehr Informationen über statische Arrays lesen Sie Kapitel 5 (Arrays).

Das erste was Ihnen bewusst wird, wenn Sie die eingebauten Befehle verwenden ist, dass sie aussehen und agieren wie normale Prozeduraufrufe. Alle eingebauten Befehle sind einfach Aufrufe von zuvor geschriebenen Prozeduren, die als ein Teil des PureBasic Paketes definiert wurden. Der Trick sie richtig zu verwenden ist, sie einfach mit den richtigen Parametern (wie sie im Syntax Beispiel definiert sind) zu versorgen, sonst bekommen Sie einen Syntax Fehler vom Compiler gemeldet.

Die Struktur des obigen Syntax Beispiels ist direkt aus der mitgelieferten Hilfe gegriffen. Wenn Sie verstanden haben, wie Sie dieses Beispiel lesen müssen, haben Sie verstanden, wie Sie die Syntax Beispiele aller anderen eingebauten Befehle lesen müssen.

PureBasic Nummern und Betriebssystem Identifikatoren

Wenn Sie die eingebauten Befehle verwenden, ist es wichtig, die Rolle von PureBasic's Objektnummerierungssystem und dem der Betriebssystem Identifikatoren zu verstehen, da diese direkt benutzt werden müssen, wenn Sie Ereignisse in Ihrem Programm kontrollieren möchten. Beide sind nichts mehr als Zahlen, aber beide werden benötigt um Objekte im Programm zu identifizieren. Teile von Grafischen Benutzeroberflächen oder verschiedene Bilder lassen sich zum Beispiel über diese Nummern identifizieren. Das Wissen, wie und wann sie zu verwenden sind ist essentiell, nicht nur für die PureBasic Programmierung, sondern für das Programmieren allgemein.

Die in der PureBasic Hilfe Datei enthaltenen Informationen sind von unschätzbarem Wert, aber sie können ein wenig verwirren, wenn Sie zum ersten mal versuchen PureBasic's Objektnummerierungssystem und das der Betriebssystem Identifikatoren zu verstehen. Das ist so, weil die Hilfe manchmal beide als Identifikatoren bezeichnet. Lassen Sie mich genau erklären, um was es sich bei beiden handelt, dann werden Sie die Hilfe Datei besser verstehen.

PB Nummern erklärt

PureBasic arbeitet mit einem aus Zahlen bestehenden System, um jedes Objekt das Sie in Ihrem Programm erstellen zu identifizieren. Ein PureBasic Objekt kann unter anderem ein Fenster, ein Gadget oder auch ein Bild sein. Diese Zahlen, wir nennen Sie PB Nummern, werden später von Ihrem Programm benutzt um auf die Objekte zu verweisen, wenn Sie verschiedene Aktionen mit diesen durchführen möchten.

Vielen PureBasic Befehlen müssen ein oder zwei PB Nummern als Parameter übergeben werden damit sie korrekt funktionieren. Diese werden in der PureBasic Hilfe innerhalb des Syntax Beispiels des entsprechenden Befehls angezeigt. Wenn ein Befehl eine PB Nummer benötigt, wird dies üblicherweise durch eine Konstante gekennzeichnet. Diese beginnt mit einer Raute (#) und anschließendem Namen der Bibliothek in der der Befehl hinterlegt ist. So wird zum Beispiel '#Image' für eine Bildnummer und '#Window' für eine Fensternummer verwendet, usw.

Hier ist ein Beispiel für einen eingebauten Befehl aus der 'Image' Bibliothek, der eine PB Nummer verwendet: (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Image → CreateImage).

```
CreateImage(#Image, Breite, Höhe [, Tiefe])
```

Wie Sie sehen können, erwartet dieser Befehl eine PB Nummer als ersten Übergabe Parameter, was durch '#Image' verdeutlicht wird. Obwohl diese im Syntax Beispiel als Konstante angezeigt wird, bedeutet das nicht zwangsläufig, das eine Konstante übergeben werden muss, sondern dass üblicherweise eine Konstante übergeben wird. Tatsache ist, dass jeder Integer Wert als PB Nummer für ein Objekt verwendet werden kann. Das einzige worauf Sie achten müssen ist, dass in einer Gruppe gleicher Objekte jedes eine einmalige Nummer haben muss. Diese Nummer verweist dann bis zum Programmende auf dieses Objekt.

Die selbe Art von Objekten kann sich keine PB Nummern teilen

PureBasic Nummern sind eine großartige Möglichkeit auf alles zu verweisen, was PureBasic im einzelnen erstellt hat. Deshalb können Sie keine zwei gleichen Objekte haben, die die selbe PB Nummer verwenden.

Wenn Sie ein Bild mit der Nummer '1' erstellen und später erstellen Sie noch ein Bild mit der Nummer '1', dann zerstört PureBasic automatisch das Erste und gibt dessen Speicher frei bevor es das zweite Bild erstellt. Das verdeutlicht, dass zwei Objekte aus der gleichen Bibliothek sich keine PB Nummer teilen können. Diese Eigenschaft kann aber auch praktisch sein, wenn Sie zum Beispiel zu einem bestimmten Zeitpunkt ein Objekt ersetzen wollen.

Gleiche PB Nummern können aber in unterschiedlichen Bibliotheken mehrfach verwendet werden. Wenn Sie zum Beispiel ein Fenster mit der Nummer '1' erstellen und anschließend eine Schaltfläche mit der Nummer '1' erstellen, dann ist das ohne Probleme möglich. Zwischen diesen beiden Nummern gibt es keine Konflikte, da sie auf Objekte aus unterschiedlichen Bibliotheken verweisen.

Wenn Sie verschiedene Befehle zum erstellen von Objekten verwenden, kann es kompliziert werden die Übersicht über alle vergebenen Nummern zu behalten. Um dieses Problem zu vermeiden, verwenden viele PureBasic Programmierer (mich eingeschlossen) Konstanten für diese PB Nummern. Diese Konstanten werden in einem 'Enumeration' Block definiert um die Nummern sequentiell zu halten. Hier ein Beispiel:

```
Enumeration
  #BILD_HUND
  #BILD_KATZE
  #BILD_VOGEL
EndEnumeration

CreateImage(#BILD_HUND, 100, 100)
CreateImage(#BILD_KATZE, 250, 300)
CreateImage(#BILD_VOGEL, 450, 115)
```

Nachdem die Konstanten definiert wurden, kann ich diese verwenden um obige Bilder (oder jedes andere Objekt) zu erstellen und muss mir keine Gedanken über irgendwelche Wertekonflikte machen. Bis zum Programmende kann ich diese Konstanten auch verwenden, um das Bild direkt mit dem Namen anzusprechen. Ich kann zum Beispiel auf das erste Bild mit dem Namen "#BILD_HUND' zugreifen. Diese Methode der Konstanten Verwendung für PureBasic Nummern garantiert gut organisierten und lesbaren Code.

Dynamische PB Nummern

Als Alternative zur Verwendung von 'Enumeration' Blöcken um PB Nummern zu verwalten, können Sie auch eine Spezielle Konstante für alle verwenden. Diese Konstante ist:

```
#PB_Any
```

Diese Konstante können Sie an jeder Stelle verwenden, an der, bei der Objekt Erstellung, eine PB Nummer benötigt wird. Der Wert der Konstante ist die nächste verfügbare PB Nummer. So hat sie zum Beispiel bei der ersten Verwendung die Nummer '1' und wenn Sie das nächste mal verwendet wird möglicherweise den Wert '2'. Dieses Verhalten eignet sich gut für die Erstellung von dynamischen Programmen, in denen Sie möglicherweise noch nicht wissen, wie viele Objekte Sie erstellen.

Das einzige was sie beachten müssen, wenn Sie '#PB_Any' als PB Nummer verwenden, ist die Tatsache, dass Sie bei einem Objektzugriff wissen müssen, welchen Wert '#PB_Any' zum Zeitpunkt der Objekterstellung hatte. Das lässt sich mit PureBasic einfach bewerkstelligen. Sie benutzen die Konstante als Parameter in einem Befehl und dieser Befehl gibt die PB Nummer des neu erstellten Objektes zurück. Hier ein Beispiel für dynamische PB Nummern, unter Verwendung des 'CreateImage()' Befehls:

```
BildHund.i = CreateImage(#PB_Any, 100, 100)
```

Hier gibt der 'CreateImage()' Befehl die PB Nummer des neu erstellten Bildes zurück, da wir die '#PB_Any' Konstante als PB Nummer Parameter übergeben haben. Das funktioniert auf die gleiche Weise mit allen anderen Befehlen die Objekte erstellen können.

PB Nummern sind einzigartig in PureBasic und sind eine schnelle und einfache Möglichkeit auf verschiedene Objekte in Ihrem Programm zuzugreifen. Gute Beispiele zur Verwendung finden Sie in Kapitel 9.

Betriebssystem Identifikatoren (Handles)

Manchen Objekten die Sie mit PureBasic erstellen müssen Sie auch eine Nummer des Betriebssystems zuweisen. Das sind die Nummern, mit deren Hilfe das aktuelle Betriebssystem die einzelnen Objekte verwaltet, wie zum Beispiel Fenster, Schriftarten, Bilder, usw. Diese Nummern werden auch als 'Handles' bezeichnet und werden sehr häufig bei der Win32 API Programmierung benötigt (Siehe Kapitel 13, 'Das

Windows Application Programming Interface'). Die PureBasic Hilfe nennt diese OS Identifikatoren (OS = Operating System = Betriebssystem) einfach IDs. Während der Arbeit mit PureBasic werden Sie feststellen, das einige Befehle OS Identifikatoren zurückgeben, während andere Befehle diese als Parameter benötigen.

Hier einige Beispiele von Befehlen die OS Identifikatoren von (mit PB Nummern als Parameter definierten) Objekten zurückgeben:

```
WindowOSId.i = WindowID(#Window)
GadgetOSId.i = GadgetID(#Gadget)
ImageOSId.i  = ImageID(#Image)
```

Ich habe hier exemplarisch drei aufgelistet, aber es gibt noch einige mehr. Befehle die mit '...ID()' enden, geben üblicherweise einen OS Identifikator zurück. Wie bereits zuvor erwähnt, hier ein Beispiel Befehl der einen OS Identifikator als Parameter benötigt:

```
UseGadgetList(WindowOSId)
```

Dieser Befehl wird verwendet um eine bestehende Gadget Liste zu öffnen (z.B. von einem Fenster, um dann weitere Elemente hinzufügen zu können), und benötigt einen OS Identifikator um zu ermitteln wo sich diese Liste befindet. In diesem Befehl wird anstelle der PB Nummer ein OS Identifikator verwendet um die maximale Kompatibilität zu wahren, da es in PureBasic auch möglich ist Fenster direkt über das API des Betriebssystems zu erzeugen. Wenn Sie die Gadgetliste eines mit PureBasic Nummern erstellten Fensters öffnen wollen, dann können Sie folgendermaßen vorgehen:

```
UseGadgetList(WindowID(#Window))
```

Beachten Sie dass wir den Befehl 'WindowID()' als Parameter verwenden, der den OS Identifikator des mit PB Nummern erstellten Fensters zurückgibt.

Jedes Betriebssystem verfügt über ein eingebautes 'Application Programming Interface' oder kurz 'API'. Das ist ein eingebauter Befehlssatz, den alle Programmiersprachen benutzen dürfen um das Betriebssystem zu steuern und Oberflächen für dieses zu erstellen. OS Identifikatoren werden vom Betriebssystem verwendet, um die Kontrolle über die einzelnen Objekte zu behalten und um auf diese zuzugreifen. Alle OS Identifikatoren sind einmalig im System und sie existieren für alle Objekte, auch die nicht mit PureBasic erstellten. Da ein Betriebssystem oft Tausende von Objekten zu verwalten hat, sind die Zahlen die es verwendet oft sehr hoch. Seien Sie nicht verwundert wenn Ihnen OS Identifikatoren mit mehr als 8 Stellen in der Länge begegnen.

OS Identifikatoren müssen nicht verwendet werden um ein voll funktionsfähiges Programm mit PureBasic zu erstellen, aber für Anfänger ist es trotzdem gut, zu verstehen um was es sich dabei handelt, auch wenn sie diese nicht verwenden. OS Identifikatoren spielen eine sehr große Rolle wenn Sie das API eines Betriebssystems verwenden, besonders das Windows API das wir in Kapitel 13 etwas näher betrachten.

Beispiele für gängige Befehle

In diesem Abschnitt werde ich ihnen einige der gängigsten Befehle von PureBasic vorstellen. Diese Befehle sind in fast jedem PureBasic Programm zu finden, weshalb das Erlernen ihrer Syntax und das Wissen über ihren Verwendungszweck vorteilhaft ist. Alle diese Befehle sind in verschiedenen Bibliotheken hinterlegt, und sind nicht notwendigerweise miteinander verwandt. Sie sind sehr nützlich für generelle Programmieraufgaben. Hier sind sie in alphabetischer Reihenfolge hinterlegt, komplett mit Fundort in der Hilfe und Beispielen zur Verwendung.

Asc()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → String → Asc)

Syntax Beispiel:

```
AsciiWert.i = Asc(Zeichen.s)
```

Dieser Befehl gibt den ASCII Wert eines Zeichens aus der Standard ASCII Tabelle zurück. Im Standard ASCII Zeichensatz werden die Zahlen von '0' bis '255' zum repräsentieren von Zeichen und Computer Steuercodes verwendet. Der Übergabe Parameter ist ein String mit einer Länge von einem Zeichen, zu dem der Befehl den verknüpften ASCII Wert zurückgibt.

```
Text.s = "Das ist ein Test"

For x.l = 1 To Len(Text)
  Debug Mid(Text, x, 1) + "  :  " + Str(Asc(Mid(Text, x, 1)))
Next x
```

Das obige Beispiel nutzt eine 'For' Schleife und den eingebauten 'Mid()' Befehl, um den in der Variable 'Text' enthaltenen Text in einzelne Zeichen zu separieren. Während des Schleifendurchlaufs wird jedes einzelne Zeichen dem 'Asc()' Befehl übergeben, der seinerseits den entsprechend zugeordneten ASCII Wert zurückgibt. Diese Werte, mit den zugeordneten Zeichen, werden dann im Debug Ausgabefenster angezeigt. Ich habe in diesem Beispiel einige neue Befehle verwendet, aber keine Angst, diese werden etwas später in diesem Abschnitt erklärt. Schauen Sie in Anhang B (Hilfreiche Tabellen), wo Sie eine ASCII Tabelle mit allen Zeichen und ihren zugeordneten Werten von '0' bis '255' finden.

Chr()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → String → Chr)

Syntax Beispiel:

```
Zeichen.s = Chr(AsciiWert.i)
```

Dieser Befehl gibt ein einzelnes Zeichen in String Form aus der Standard ASCII Tabelle zurück. Der zurückgegebene String ist das mit dem übergebenen Wert verknüpfte ASCII Zeichen. Im Standard ASCII Zeichensatz sind die Zeichen '33' bis '126' tatsächlich druckbare Zeichen, die Sie auf einer englischen Standard Tastatur finden.

Der Parameter der diesem Befehl übergeben wird: 'AsciiWert.i', ist die Zahl zu der Sie das zugeordnete Zeichen erhalten wollen.

```
Text.s = Chr(68)  + Chr(97)  + Chr(115) + Chr(32)
Text.s + Chr(105) + Chr(115) + Chr(116) + Chr(32)
Text.s + Chr(101) + Chr(105) + Chr(110) + Chr(32)
Text.s + Chr(84)  + Chr(101) + Chr(115) + Chr(116)
Debug Text
```

Das obige Beispiel konstruiert den String 'Das ist ein Test', indem es die Ergebnisse von vielen 'Chr()' Befehlen aneinander hängt. Der erste Befehl 'Chr(68)' zum Beispiel, gibt das Zeichen 'D' zurück. Der Rest wird dann an die 'Text' Variable angehängt. Schauen Sie in Anhang B (Hilfreiche Tabellen), wo Sie eine ASCII Tabelle mit allen Zeichen und ihren zugeordneten Werten von '0' bis '255' finden.

Delay()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Misc → Delay)

Syntax Beispiel:

```
Delay(Millisekunden)
```

Dieser Befehl pausiert sämtliche Programmaktivität für den Zeitraum, der im 'Millisekunden' Parameter spezifiziert wurde. Die Zeit wird in Millisekunden gemessen, was einer Tausendstel Sekunde entspricht.

```
Debug "Start..."
Delay(5000)
Debug "Das wird 5 Sekunden später ausgeführt"
```

Wenn obiges Beispiel gestartet wird, wird die zweite 'Debug' Anweisung fünf Sekunden nach der Ersten ausgeführt, da ich den Wert '5000' Millisekunden (5 Sekunden) mit dem 'Delay()' Befehl verwendet habe.

ElapsedMilliseconds()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Misc → ElapsedMilliseconds)

Syntax Beispiel:

```
Ergebnis.i = ElapsedMilliseconds()
```

Dieser Befehl gibt die Anzahl von Millisekunden zurück, die seit dem Einschalten des Computers vergangen sind.

```
Debug "Start..."
StartZeit.i = ElapsedMilliseconds()
Delay(5000)
Debug "Pausiert für " + Str(ElapsedMilliseconds() - StartZeit) + " Millisekunden."
```

Dieser Befehl ist großartig für Zeitoperationen in Ihrem Programm geeignet. Wenn Sie zum Beispiel die verstrichenen Millisekunden seit einem bestimmten Ereignis messen wollen. Zuerst speichern Sie die Startzeit mit diesem Befehl in einer Integer Variable. Wenn Sie dann diesen 'Timer' stoppen wollen, speichern Sie erneut die Zeit mit dem 'ElapsedMilliseconds()' Befehl. Um das Ergebnis zu erhalten müssen Sie dann die Startzeit von der Endzeit abziehen und Sie erhalten die verstrichene Zeit in Millisekunden, die zwischen dem ausführen der beiden Befehle vergangen ist. Das wird alles in obigem Beispiel dargestellt. Wenn Sie den Befehl zum ersten mal starten, sind Sie vermutlich von der großen Zahl des Ergebniswertes geschockt. Sie müssen aber bedenken, das dies die Anzahl der seit dem Systemstart vergangenen Millisekunden ist. Keine Angst bei den großen Zahlen, merken Sie sich, dass Sie nur einen Zeitpunkt festhalten und mit der obigen Methode die vergangene Zeit bis zu einem anderen Zeitpunkt berechnen können.

FindString()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → String → FindString)

Syntax Beispiel:

```
Position.i = FindString(String.s, GesuchterString.s, StartPosition.i)
```

Dieser Befehl versucht den 'GesuchterString' Parameter innerhalb des 'String' Parameters zu finden, beginnend an der Position die durch den 'StartPosition' Parameter vorgegeben wird. Wenn der String gefunden wurde, wird die Position des ersten Auftretens zurückgegeben. Dieser Wert bezieht sich auf die Anzahl der Zeichen vom Beginn des Strings gerechnet, beginnend mit '1'.

```
String.s          = "Ich steh' auf PureBasic Pur!"
GesuchterString.s = "Pur"

ErstesAuftreten.i  = FindString(String, GesuchterString, 1)
ZweitesAuftreten.i = FindString(String, GesuchterString, ErstesAuftreten + 1)
Debug "Index des ersten Auftretens: "  + Str(ErstesAuftreten)
Debug "Index des zweiten Auftretens: " + Str(ZweitesAuftreten)
```

Dieses Beispiel zeigt, wie Sie einen String in einem anderen String finden. Der erste 'FindString()' Befehl versucht den String 'Pur' ab der ersten Position zu finden, was er auch erfolgreich erledigt und dann der Variable 'ErstesAuftreten' den Wert '15' zuweist. Der Zweite 'FindString()' Befehl versucht den gleichen String zu finden, beginnt aber an einer anderen Position, nämlich 'ErstesAuftreten + 1' um zu vermeiden dass wir den ersten Fund erneut finden. Die Zweite Suche gibt als Ergebnis '25' zurück, welches unverzüglich der Variable 'ZweitesAuftreten' zugewiesen wird. Beide Variablen werden dann im Debug Ausgabefenster ausgegeben.

Len()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → String → Len)

Syntax Beispiel:

```
Laenge.i = Len(String.s)
```

Dieser Befehl gibt die Länge des Parameter 'String' in Anzahl der Zeichen zurück.

```
Alphabet.s     = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
StringLaenge.i = Len(Alphabet)

Debug StringLaenge
```

Dieses Beispiel ist sehr einfach, da der 'Len()' Befehl selbst sehr einfach ist. Das Alphabet wird einer String Variable mit Namen 'Alphabet' zugewiesen. Diese Variable wird dann dem 'Len()' Befehl übergeben, welcher den Wert '26' (das ist die Anzahl der Zeichen die die 'Alphabet' Variable enthält) zurückgibt. Dieser Wert wird der Variable 'StringLaenge' zugewiesen und dann im Debug Ausgabefenster angezeigt.

MessageRequester()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Requester → MessageRequester)

Syntax Beispiel:

```
Ergebnis.i = MessageRequester(Titel.s, Text.s [, Flags])

;Mögliche Flags:
#PB_MessageRequester_Ok
#PB_MessageRequester_YesNo
#PB_MessageRequester_YesNoCancel

;Mögliche Rückgabe Werte:
#PB_MessageRequester_Yes
#PB_MessageRequester_No
#PB_MessageRequester_Cancel
```

Dieser Befehl wird zum erstellen eines kleinen Fensters verwendet, das irgendeine Information in Ihrem Programm anzeigt. Er kann in jeder Art Programm verwendet werden, nicht nur in Programmen mit grafischer Benutzeroberfläche. Der erste Parameter dieses Befehls ist der Text, der in der Titelleiste des Fensters erscheint. Der Zweite Parameter ist der eigentliche Meldungstext, der auf dem Fenster erscheint. Der dritte und letzte Parameter sind die verfügbaren, optionalen Flags (Bitschalter). Mit der Verwendung verschiedener Flags, können Sie das Aussehen des Fensters beeinflussen. Hierdurch lassen sich dem Fenster unterschiedliche Schaltflächen hinzufügen. Ihr Programm wird so lange angehalten, bis Sie eine der Schaltflächen gedrückt haben. Um herauszufinden, welche Schaltfläche gedrückt wurde, können Sie den Rückgabewert folgendermaßen prüfen:

```
Titel.s     = "Information"
Nachricht.s = "Das ist der Standard Stil des Nachrichten Fensters"
MessageRequester(Titel, Nachricht, #PB_MessageRequester_Ok)

Nachricht.s = "In diesem Stil können Sie [Ja] oder [Nein] auswählen."
Ergebnis.i  = MessageRequester(Titel, Nachricht, #PB_MessageRequester_YesNo)

If Ergebnis = #PB_MessageRequester_Yes
  Debug "Sie haben [Ja] gedrückt"
Else
  Debug "Sie haben [Nein] gedrückt"
EndIf

Nachricht.s = "In diesem Stil können Sie [Ja], [Nein] oder [Abbrechen] auswählen."
Ergebnis.i  = MessageRequester(Titel, Nachricht, #PB_MessageRequester_YesNoCancel)

Select Ergebnis
  Case #PB_MessageRequester_Yes
    Debug "Sie haben [Ja] gedrückt"
  Case #PB_MessageRequester_No
    Debug "Sie haben [Nein] gedrückt"
  Case #PB_MessageRequester_Cancel
    Debug "Sie haben [Abbrechen] gedrückt"
EndSelect
```

Dieses Beispiel zeigt alle möglichen Wege, den 'MessageRequester' Befehl zu verwenden. Es zeigt außerdem, wie Sie den Rückgabewert auswerten können um die gedrückte Schaltfläche zu ermitteln. Die Verwendung der Konstanten erspart Ihnen die Mühe, sich numerische Werte merken zu müssen. Sie müssen nicht wissen welche Werte den Konstanten intern zugewiesen sind, wir sind nur daran interessiert ob sie gleich oder ungleich dem Rückgabewert des Befehls sind.

Mid()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → String → Mid)

Syntax Beispiel:

```
Ergebnis.s = Mid(String.s, StartPosition.i, Laenge.i)
```

Dieser Befehl gibt einen abgeschnittenen String zurück, der aus einem anderen String extrahiert wurde. Der Ausgangsstring wird als 'String' Parameter übergeben. Der extrahierte String kann von irgendeiner Position innerhalb des Ausgangsstring entnommen werden. Die genaue Position wird durch den 'StartPosition' Parameter vorgegeben. Die Länge des neuen extrahierten Strings wird durch den 'Laenge' Parameter definiert. Hier ein Beispiel:

```
AusgangsString.s = "Das Leben geht weiter"
ExtrahierterString.s = Mid(AusgangsString, 5, 5)
Debug ExtrahierterString
```

Hier habe ich den String 'Leben' extrahiert, indem ich die Startposition auf '5' gesetzt habe, und von dieser Stelle ab '5' Zeichen extrahiert habe. Wie beim 'FindString()' Befehl wird auch hier die Position im Ausgangsstring durch die Anzahl von Zeichen ermittelt.

Random()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Misc → Random)

Syntax Beispiel:

```
Ergebnis.i = Random(Maximum.i)
```

Dieser Befehl ist einfach zu demonstrieren, da er nur eine Zufalls Integer Zahl zwischen (und einschließlich) '0' und dem Wert, der als 'Maximum' Parameter definiert wurde, zurückgibt.

```
Debug Random(100)
```

In obigem Beispiel gibt der 'Random()' Befehl einen Wert zwischen '0' und '100' zurück.

Str(), StrF(), StrD()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → String → Str)

Syntax Beispiel:

```
Ergebnis.s = Str(Wert.q) [Alle Ganzzahl Typen von .b bis .q]
Ergebnis.s = StrF(Wert.f [, AnzahlDezimalstellen.i])
Ergebnis.s = StrD(Wert.d [, AnzahlDezimalstellen.i])
```

Diese Drei Befehle haben genau genommen die gleiche Aufgabe. Ihr Verwendungszweck ist es, numerische Werte in Strings umzuwandeln. Es sind drei verschiedene Befehle, um drei verschiedene Typen von numerischen Werten zu verarbeiten, 'Str()' zum verarbeiten von Ganzzahlen ('.b' bis '.q'), 'StrF()' zum verarbeiten von Float Werten und 'StrD()' zum verarbeiten von Double Werten. Sie müssen den richtigen 'Str()' Befehl für Ihren speziellen Anwendungszweck auswählen. Im Falle von 'StrF()' und 'StrD()' haben Sie noch einen zusätzlichen, optionalen Parameter mit Namen 'AnzahlDezimalstellen'. Mit diesem Parameter können Sie die Anzahl der Nachkommastellen bestimmen, auf die der Wert dann gerundet wird bevor er zum String konvertiert wird.

```
Debug "Long konvertiert nach String: "   + Str(2147483647)
Debug "Quad konvertiert nach String: "   + Str(9223372036854775807)
Debug "Float konvertiert nach String: "  + StrF(12.05643564333454646, 7)
Debug "Double konvertiert nach String: " + StrD(12.05643564333454646, 14)
```

Das obige Beispiel zeigt wie vier unterschiedliche Arten von numerischen Werten in Strings konvertiert werden. Ich habe einen wörtlichen String mit dem konvertierten String verbunden um ganze Sätze zu bilden, die ich dann im Debug Ausgabefenster anzeige. Eine wichtige Sache die Sie beachten sollten: Wenn Sie den 'AnzahlDezimalstellen' Parameter beim 'StrF()' und 'StrD()' Befehl auslassen, dann geben beide standardmäßig sechs Dezimalstellen zurück und Sie verlieren unter Umständen eine Menge Präzision.

Val(), ValF(), ValD()

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → String → Val)

Syntax Beispiel:

```
Ergebnis.q = Val(String.s)
Ergebnis.f = ValF(String.s)
Ergebnis.d = ValD(String.s)
```

Diese Drei Befehle haben genau genommen die Gleiche Aufgabe und sind das genaue Gegenteil des 'Str()' Befehls. Der Unterschied ist, dass diese Befehle einen String als Parameter entgegennehmen und einen numerischen Wert zurückgeben (abhängig davon, welchen Val() Befehl Sie verwenden). 'Val()' gibt zum Beispiel einen Quad Wert zurück, 'ValF()' einen Float Wert und 'ValD()' gibt einen Double Wert zurück. Hier ein einfaches Beispiel:

```
LongTypVariable.l   = Val("2147483647")
QuadTypVariable.q   = Val("92233720368547758O7")
FloatTypVariable.f  = ValF("12.05643564333454646")
DoubleTypVariable.d = ValD("12.05643564333454646")

Debug LongTypVariable
Debug QuadTypVariable
Debug FloatTypVariable
Debug DoubleTypVariable
```

In diesem Beispiel habe ich alle drei verschiedenen Arten des Val() Befehls verwendet, um vier Strings in die entsprechenden numerischen Typen zu konvertieren. Beachten Sie, dass der String abgeschnitten wird wenn er das numerische Limit des entsprechenden Typs überschreitet. Wenn Sie zum Beispiel in die Zeile 'ValF("12.05643564333454646")' des obigen Beispiels schauen, werden Sie feststellen dass diese Zahl zu groß (oder zu Präzise) für einen normalen Float Wert ist. Deshalb wird dieser String beim konvertieren passend auf den Zieltyp abgeschnitten.

Arbeiten mit Dateien

Die Verwendung von Dateien sind der natürliche Weg eines Programmes, um Daten zu speichern oder auf diese zuzugreifen. PureBasic bietet volle Unterstützung zum lesen und schreiben von Dateien und kann auf eine beliebige Anzahl von Dateien gleichzeitig Lese- und Schreiboperationen ausführen. Alle Befehle zum manipulieren von Dateien sind in der 'File' Bibliothek abgelegt, deren Beschreibung Sie in der Hilfe folgendermaßen lokalisieren:

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → File)

In diesem Abschnitt werden nicht alle Befehle ausführlich besprochen, aber ich werde Ihnen eine gute Einführung in das Lesen und Schreiben von Dateien geben, mit der Sie auch die übrigen Befehle gut verstehen können wenn Sie sie verwenden.

Schreiben in eine Datei

Lassen Sie uns mit einem Beispiel beginnen, in dem wir Daten in eine Datei schreiben. Hier werden wir ein paar Strings in eine einfache Text Datei schreiben.

```
#DATEI_KINDERLIED = 1

Dim Kinderlied.s(2)

Kinderlied(0) = "Alle meine Entchen schwimmen auf dem See,"
Kinderlied(1) = "schwimmen auf dem See,"
Kinderlied(2) = "Köpfchen in das Wasser, Schwänzchen in die Höh'."

If CreateFile(#DATEI_KINDERLIED, "Entchen.txt")
  For x.i = 0 To 2
    WriteStringN(#DATEI_KINDERLIED, Kinderlied(x))
  Next x
  CloseFile(#DATEI_KINDERLIED)
EndIf
```

Zu Beginn habe ich die Konstante '#DATEI_KINDERLIED' erstellt, die wir als PB Nummer für unser neues Datei Objekt verwenden, und ich habe ein String Array definiert, welches das Kinderlied enthält. Nachdem wir die Strings zum 'Kinderlied()' Array hinzugefügt haben, gehen wir dazu über unsere Datei zu schreiben. Wenn ich eine existierende Datei öffnen will, kann ich den 'OpenFile()' Befehl verwenden, aber in obigem Beispiel möchte ich eine neue Datei erstellen, deshalb verwende ich stattdessen den 'CreateFile()' Befehl.

Der 'CreateFile()' Befehl benötigt zwei Parameter. Der erste ist die PB Nummer mit der das neue Datei Objekt verknüpft wird und der zweite ist der eigentliche Dateiname den diese Datei erhält wenn sie erstellt wird. Ich habe den 'CreateFile()' Befehl in eine 'If' Anweisung eingebunden, um zu prüfen dass der Befehl True zurückgibt. Wenn das der Fall ist, dann wurde die Datei erfolgreich erstellt und ich kann mit meinem Programm weiter machen. Es ist immer gut, die Dateierstellung zu überprüfen, da sehr viele Fehler auftreten können, wenn Sie versuchen in eine ungültige Datei zu schreiben.

Um die eigentlichen Strings in die Datei zu schreiben habe ich den 'WriteStringN()' Befehl verwendet. Dieser benötigt zwei Parameter. Der erste ist die PB Nummer, die mit dem Datei Objekt verknüpft ist das ich beschreiben möchte. Der zweite Parameter ist der eigentlich String den ich in die Datei schreiben möchte. Ich verwende eine Schleife um mich durch das Array zu arbeiten und alle seine Inhalte in die Datei zu schreiben, und somit das Kinderlied zu schreiben. Der 'WriteStringN()' Befehl den ich in diesem Beispiel verwendet habe, ist eine erweiterte Version des 'WriteString()' Befehls. Tatsächlich ist der einzige Unterschied das 'N' vor den Klammern. Dieses 'N' bedeutet, das ein 'Neue Zeile' Zeichen nach dem String in die Datei geschrieben wird. Da ich möchte, dass nach jedem String den ich in die Datei schreibe eine neue Zeile beginnt, verwende ich den 'WriteStringN()' Befehl, sonst würde der komplette Text in einer Zeile stehen.

Verschiedene Wege zum erstellen oder öffnen von Dateien

Wenn Sie die Datei Befehle von PureBasic zum lesen oder schreiben von Dateien verwenden, müssen Sie darauf achten, dass Sie abhängig von der Dateiverwendung den richtigen Befehl verwenden. Die nachstehende Liste beschreibt Befehle die Lese- und Schreiboperationen ermöglichen, und wofür Sie verwendet werden.

'ReadFile()' öffnet die definierte Datei zum lesen, verhindert aber jegliche Schreiboperationen auf diese.
'OpenFile()' öffnet die definierte Datei zum lesen oder schreiben und erstellt sie, wenn sie nicht existiert.
'CreateFile()' erstellt eine leere Datei zum schreiben, existiert sie bereits, wird sie durch eine leere ersetzt.

Jeder Befehl arbeitet auf die gleiche weise und alle haben die gleichen zwei Parameter. Der erste Parameter ist die PB Nummer die mit dem Datei Objekt verknüpft ist und der zweite Parameter ist der eigentliche Dateiname auf dem Laufwerk.

Nachdem ich alle Strings in die Datei geschrieben, und die Schleife beendet habe, schließe ich die Datei mit dem 'CloseFile()' Befehl. Dieser benötigt nur einen Parameter, nämlich die PB Nummer des Datei Objektes das Sie schließen möchten.

Diese Datei mit dem Kinderlied sollte sich nun irgendwo auf ihrer Festplatte befinden. Wenn Sie eine Datei mit einem relativen Pfad erstellen (wie z.B. 'Entchen.txt'), dann befindet sich diese Datei im gleichen Verzeichnis wie Ihr Quelltext. Wenn Sie eine Datei an einem speziellen Ort ablegen wollen, müssen Sie einen absoluten Pfad angeben. Dieser muss den Laufwerksbuchstaben, die Verzeichnisse und den Dateinamen angeben, etwa wie hier:

```
...
If CreateFile(#DATEI_KINDERLIED, "C:\Mein Verzeichnis\Entchen.txt")
...
```

Wenn Sie diese Methode benutzen, müssen Sie dafür sorgen dass alle Verzeichnisse im spezifizierten Pfad auch existieren bevor Sie die Datei erstellen, sonst schlägt die Erstellung fehl.

Der 'CloseFile()' Befehl ist sehr wichtig

Wenn Sie Ihre Lese- und besonders Schreiboperationen an der Datei abgeschlossen haben, müssen Sie die Datei ordnungsgemäß mit dem 'CloseFile()' Befehl schließen. Dieser Befehl schließt nicht nur einfach die Datei sondern gibt diese auch wieder frei, damit sie bei Bedarf von einem anderen Programm geöffnet werden kann. Der 'CloseFile()' Befehl spielt eine weitere sehr wichtige Rolle, da er vor dem schließen noch alle Daten aus dem Datei Puffer in die Datei hineinschreibt. PureBasic verwendet ein Puffersystem um Dateizugriffe zu beschleunigen. Wenn Sie eine Datei in einem anderen Editor öffnen und Sie einige Daten vermissen, dann müssen Sie prüfen ob Sie die Datei ordnungsgemäß mit dem 'CloseFile()' Befehl geschlossen haben. Dieser Befehl stellt sicher, dass alle Daten aus dem Puffer auf die Festplatte geschrieben werden.

Die Datei Puffer sind für den regulären Benutzer völlig unsichtbar, deshalb müssen Sie sich um diese nicht zu viele Sorgen machen. Sie sollten nur daran denken, das Sie jede Datei nach Beendigung der Arbeit schließen, um irgendwelche Fehler zu vermeiden.

Wie schreibe ich Werte von anderen eingebauten Typen?

Das Schreiben der anderen eingebauten Datentypen in eine Datei ist genauso einfach wie das Schreiben von Strings. Da Sie nun wissen wie der 'WriteString()' Befehl funktioniert, sind die anderen Befehle einfach zu verstehen. Hier ein paar Syntax Beispiele zum schreiben der anderen eingebauten Typen:

```
WriteByte(#Datei, Wert.b)
WriteCharacter(#Datei, Wert.c)
WriteUnicodeCharacter(#Datei, Wert.u)
WriteWord(#Datei, Wert.w)
WriteLong(#Datei, Wert.l)
WriteInteger(#Datei, Wert.i)
WriteQuad(#Datei, Wert.q)
WriteFloat(#Datei, Wert.f)
WriteDouble(#Datei, Wert.d)
```

Diese Befehle können verwendet werden um jeden beliebigen eingebauten Typ in eine Datei zu schreiben, und sie können so häufig verwendet werden wie Sie wollen. Diese Werte, auch die Strings, können immer wieder (auch in gemischter Form) in eine einzelne Datei geschrieben werden, ganz so wie Sie es benötigen. Um die Sache einfach zu halten - diese Befehle werden alle auf die gleiche Art eingesetzt, wie der 'WriteString()' Befehl. Der erste Parameter ist das Datei Objekt in das geschrieben werden soll, und der zweite Parameter ist der zu schreibende Wert.

Lesen aus einer Datei

Lesen aus einer Datei ist genauso einfach wie in diese zu schreiben. PureBasic macht das extrem einfach mit leistungsfähigen Lesebefehlen. Schauen Sie auf das nächste Beispiel, dort lese ich Strings aus einer Datei mit dem Namen 'Report.txt' und platziere diese in einer Liste mit dem Namen 'DateiInhalt'. Der Inhalt der Liste wird dann im Debug Ausgabefenster ausgegeben, um die Stringdaten in der Liste zu zeigen. Sie müssen diese Daten nicht anzeigen, haben aber die Möglichkeit die Strings an jeder passenden Stelle in Ihrem Programm zu verwenden oder diese zu manipulieren.

```
#DATEI_REPORT = 1

NewList DateiInhalt.s()

If ReadFile(#DATEI_REPORT, "Report.txt")
  While Eof(#DATEI_REPORT) = #False
    AddElement(DateiInhalt())
    DateiInhalt() = ReadString(#DATEI_REPORT)
  Wend
  CloseFile(#DATEI_REPORT)
EndIf

ForEach DateiInhalt()
  Debug DateiInhalt()
Next
```

Zu Beginn des Codes habe ich eine Konstante definiert, die ich als PB Nummer für die geöffnete Datei verwende. Dann habe ich eine Liste mit dem Namen 'DateiInhalt' erstellt, in der ich die Strings speichere die

ich aus der Datei lese. Sie müssen keine Liste verwenden um diese Strings zu verwalten, Ich denke aber dass dies die Organisation etwas erleichtert. Ich habe hier eine Liste verwendet, da sie einfach zu verwenden sind und bei Bedarf mit der Anzahl der Daten wachsen, anders als Arrays, deren Größe vordefiniert werden muss.

Ich habe die Datei mittels des 'ReadFile()' Befehls geöffnet. Dieser öffnet die Datei mit einem nur Lese Status und verhindert alle Schreiboperationen auf die Datei, während sie geöffnet ist. Wie beim 'CreateFile()' Befehl ist der erste Parameter die PB Nummer mit dem diese neu geöffnete Datei verknüpft wird, während der zweite Parameter der Name der Datei ist, von der gelesen werden soll. Wie zuvor habe ich den Befehl in eine 'If' Anweisung eingebettet, um mich zu vergewissern dass die Operation True zurückgibt. Wenn dies der Fall ist, kann ich mir sicher sein dass die Datei ordnungsgemäß geöffnet ist und ich habe die Möglichkeit, lesenderweise auf diese zuzugreifen.

Wenn die Datei geöffnet ist, möchte ich die Informationen in ihr lesen. Ich habe hier eine 'While' Schleife verwendet, damit ich nicht ständig sich wiederholende Befehle eingeben muss, um die darin enthaltenen Strings zu lesen. Die Schleife verwendet den 'Eof()' Befehl, was für 'End of File' steht. Dieser Befehl gibt True zurück wenn er das Ende der geöffneten Datei erreicht hat. In der Schleife überprüfe ich vor jedem Durchlauf, ob der Rückgabewert dieses Befehles False ist. Wenn dies der Fall ist, habe ich das Ende der Datei noch nicht erreicht und kann weitere Daten aus der Datei lesen. Der 'Eof()' Befehl benötigt einen Parameter, nämlich die PB Nummer von der Datei die Sie prüfen möchten.

Während die Datei geöffnet ist, verwende ich den 'ReadString()' Befehl um bei jedem Schleifendurchlauf einen String zu lesen. Der String der zurückgegeben wird beginnt am Anfang der aktuellen Zeile, und endet wenn der Befehl auf ein 'Neue Zeile' Zeichen trifft. Dies bewegt auch den Dateizugriffszeiger eine Zeile weiter voran. Jedes Mal wenn ich einen String aus der Datei lese, erstelle ich ein neues Element in der 'DateiInhalt' Liste und weise diesem den gelesenen String zu. Das führe ich solange aus, bis der 'Eof()' Befehl True zurückgibt. Dann verlasse ich die Schleife.

Wenn die Schleife verlassen wurde schließe ich die Datei ordnungsgemäß mit dem 'CloseFile()' Befehl. Das Schließen der Datei ist sehr wichtig, und sollte so schnell wie möglich nach der letzten Dateioperation ausgeführt werden. Eine einfache 'ForEach' Schleife gibt dann die gelesenen Daten im Debug Ausgabefenster aus.

Wie lese ich die anderen Werte von eingebauten Typen?

Das letzte Beispiel hat mit Strings gearbeitet, Sie können aber genauso einfach auch andere Typen aus einer Datei lesen. PureBasic stellt spezifische Befehle für jeden eingebauten Typ zur Verfügung, um diese korrekt zu lesen.

```
ReadByte(#Datei)
ReadCharacter(#Datei)
ReadUnicodeCharacter(#Datei)
ReadWord(#Datei)
ReadLong(#Datei)
ReadInteger(#Datei)
ReadQuad(#Datei)
ReadFloat(#Datei)
ReadDouble(#Datei)
```

Um die Dinge einfach zu halten, verwenden alle diese Befehle den gleichen Parameter wie der 'ReadString()' Befehl. Dieser Parameter ist die PB Nummer des geöffneten Dateiobjektes, von dem gelesen wird. Der einzige Unterschied zwischen diesen Befehlen und 'ReadString()' ist der, dass sie nur einen einzigen Wert aus der Datei lesen, nämlich die zu dem entsprechenden Typ passende Anzahl an Bytes. Anders als der 'ReadString()' Befehl lesen sie nicht bis zum Ende der Zeile. Wenn ich zum Beispiel einen Wert vom Typ Long aus der Datei lesen möchte, verwende ich den 'ReadLong()' Befehl, der '4' Bytes aus der Datei liest und diese Bytes als Long-Wert zurückgibt. Diese Aktion bewegt den Dateizugriffszeiger innerhalb der Datei um '4' Bytes weiter vorwärts, bereit für den nächsten Befehl.

Der Dateizugriffszeiger

Zu jeder geöffneten Datei existiert ein unsichtbarer Dateizugriffszeiger. Diese imaginäre Position gibt an, wo Sie in der Datei eine Lese- oder Schreiboperation ausführen. Wenn Sie die Befehle 'ReadFile()', 'OpenFile()' oder 'CreateFile()' verwenden, steht der Dateizugriffszeiger am Beginn der Datei, bereit für die nächste Operation. Wenn Sie beginnen in der Datei zu lesen oder zu schreiben, wandert der Dateizugriffszeiger. Wenn Sie in die Datei schreiben, bewegt sich der Zeiger hinter den zuletzt geschriebenen Wert, bereit für den Nächsten. Wenn Sie aus einer Datei lesen, bewegt sich der Zeiger nach jeder Leseoperation auf die Position hinter dem gelesenen Wert, bis das Ende der Datei erreicht ist und der 'Eof()' Befehl True zurückgibt.

Lokalisieren des Dateizugriffszeigers

Jedes Mal wenn Sie eine Lese- oder Schreiboperation in der Datei durchgeführt haben, können Sie die Position des Dateizugriffszeigers mit dem 'Loc()' Befehl ermitteln. 'Loc()' steht für 'Location' (momentaner Standort). Der Rückgabewert wird in Byte gemessen.

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → File → Loc)

Syntax Beispiel:

```
Position.q = Loc(#Datei)
```

Dieser Befehl gibt die Position des Dateizugriffszeigers (in Byte) innerhalb des, als Parameter übergebenen, geöffneten Datei Objektes zurück. Um auch große Dateien zu unterstützen, wird die Position als Quad Wert zurückgegeben.

Bewegen des Dateizugriffszeigers

Der Dateizugriffszeigers kann jederzeit mit dem 'FileSeek()' Befehl bewegt werden

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → File → FileSeek)

Syntax Beispiel:

```
FileSeek(#Datei, NeuePosition.q)
```

Dieser Befehl benötigt zwei Parameter, der Erste ist die PB Nummer des Datei Objektes das modifiziert werden soll, und der Zweite ist die Neue Position (in Byte) des Dateizugriffszeigers. Der 'NeuePosition' Parameter ist vom Typ Quad, um große Dateien zu unterstützen.

Herausfinden der gegenwärtigen Dateigröße

Um die Größe der momentan verwendeten Datei zu ermitteln, verwenden Sie den 'Lof()' Befehl. 'Lof()' steht für 'Length of File' (Länge der Datei).

(Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → File → Lof)

Syntax Beispiel:

```
Laenge.q = Lof(#Datei)
```

Dieser Befehl gibt die Länge in Byte des über die PB Nummer spezifizierten Datei Objektes zurück. Die Länge wird als Quad Wert zurückgegeben, um Große Dateien zu unterstützen.

Loc, Lof und FileSeek Beispiel

In diesem nächsten Codeschnipsel demonstriere ich Ihnen die Verwendung von 'Loc()', 'Lof()' und 'FileSeek()' in einer Form, die sehr häufig verwendet wird. Im nachfolgenden Code lese ich eine MP3 Musikdatei und prüfe, ob diese ein 'ID3(v1) Tag' enthält. Diese 'Tags' enthalten Informationen wie z.B.: Künstler, Liedname, Genre, usw.

Ich habe mich im Internet über die Spezifikationen von 'MP3 ID3 Tags' belesen, und herausgefunden dass diese Informationen immer ans Ende der regulären MP3 Datei angehängt sind und immer eine Länge von '128' Byte haben. Die Spezifikation sieht vor, dass die ersten '3' Bytes dieses Eintrages die Zeichen 'TAG' sind. Deshalb werden wir auf diese Besonderheit hin prüfen. Hier ist der Code:

```
#DATEI_MP3 = 1

MP3Datei.s = "Test.mp3"

If ReadFile(#DATEI_MP3, MP3Datei)
  FileSeek(#DATEI_MP3, Lof(#DATEI_MP3) - 128)
  For x.i = 1 To 3
    Text.s + Chr(ReadByte(#DATEI_MP3))
  Next x
  CloseFile(#DATEI_MP3)
  If Text = "TAG"
    Debug "'" + MP3Datei + "' enthält ein ID3v1 Tag."
  Else
    Debug "'" + MP3Datei + "' enthält kein ID3v1 Tag."
  EndIf
EndIf
```

Nach dem Öffnen der Datei mit dem 'ReadFile()' Befehl, habe ich den Dateizugriffszeiger an die richtige Stelle bewegt, die ich durch die Methode 'Länge der Datei minus 128 Byte' ermittelt habe. Wie hier:

```
FileSeek(#DATEI_MP3, Lof(#DATEI_MP3) - 128)
```

Nachdem dieser Befehl seine Arbeit beendet hat, befindet sich der Dateizugriffszeiger an der richtigen Stelle. Dann benutzte ich eine Schleife, um dreimal eine 'ReadByte()' Operation durchzuführen. Die Ergebniswerte habe ich jeweils einem 'Chr()' Befehl übergeben, der seinerseits das ermittelte Zeichen zurückgibt. Diese Zeichen werden dann in einer String Variable mit dem Namen 'Text' aneinandergehängt. Wenn die Schleife beendet und die Datei geschlossen ist, prüfe ich den Wert von 'Text'. Wenn der in 'Text' enthaltene String gleich 'TAG' ist, dann enthält die MP3 Datei ein 'ID3(v1) Tag'.

Lesen Sie die Hilfe

Das war über Jahre besonders in Internet Foren der Schlachtruf von vielen guten Programmierern. Um etwas richtig zu verstehen, müssen Sie es studieren, und um etwas zweckmäßig zu studieren, müssen Sie alles darüber lesen. Es steht außer Frage, dass Menschen die mehr über ein Thema gelesen haben es meist auch besser verstanden haben. Diese Weisheit gilt besonders für Programmiersprachen.

Den besten Rat den ich Ihnen zum erlernen von PureBasic geben kann: Lesen Sie die Hilfe von vorne bis hinten und lesen Sie jede Seite über jeden Befehl. Das hört sich zwar ziemlich langweilig an, aber glauben Sie mir, es wird Ihr Verständnis von PureBasic erheblich steigern und Sie bekommen einen guten Einblick in das, was mit dieser großartigen Programmiersprache alles möglich ist.

Manchmal ist das einzige was einen guten Programmierer von einem richtig Genialen trennt einfach nur die investierte Zeit für das Studium der Dokumentation.

Ich hoffe dieses Kapitel hat Ihnen genug Einblick gegeben wie Sie die Hilfe lesen müssen, und verstanden haben wie Sie die meisten der eingebauten Befehle unter Zuhilfenahme des Syntax Beispiels verwenden können.

8. Guter Programmierstil

Bis jetzt habe ich mich darauf konzentriert, Ihnen die wesentlichen Punkte der PureBasic Programmierung zu erklären. Die eingebauten Typen, Anweisungen und Ausdrücke, usw. Ich denke das jetzt der Zeitpunkt gekommen ist, Ihnen etwas darüber zu erzählen wie Sie Ihr Programm beim schreiben gestalten sollten.

In diesem Kapitel werde ich über etwas berichten, das alle Programmierer erkennen und anwenden sollten: Guter Programmierstil. Programmieren zu lernen kann viel Spaß bereiten, aber diese Freude verfliegt schnell wenn Sie sich mit unsauber programmiertem Code auseinandersetzen müssen (und versuchen ihn zu verstehen), besonders wenn es Ihr eigener Code ist. Sauber formatierter Code sorgt nicht nur dafür dass er professioneller aussieht, sondern er unterstützt ungemein beim lesen und verstehen des geschriebenen.

In den folgenden Abschnitten werde ich Ihnen Wege aufzeigen, wie Sie Ihren Code für eine maximale Lesbarkeit formatieren. Es ist nicht zwangsläufig der richtige oder falsche Weg, es ist einfach die Art wie ich üblicherweise vorgehe. Ich hoffe, dass Sie sich nach dem Lesen einen eigenen sauberen Programmierstil aneignen und diesen auch beibehalten.

Später in diesem Kapitel werde ich Ihnen einige Tipps und Tricks verraten wie Sie Fehler in Ihrem Code vermeiden können und gebe Ihnen Beispiele, wie Sie trotzdem auftretende Fehler behandeln können. Ich habe auch eine Anleitung für den PureBasic Debugger mit angefügt, mit dem Sie Probleme die in Ihrem Code auftreten sauber herunterbrechen und lokalisieren können.

Warum soll ich mich mit sauberem formatieren von Code aufhalten?

Diese Frage wird manchmal von Anfängern gestellt und sollte auf jeden Fall behandelt werden. Eine Antwort könnte sein: Wenn wir dieses Buch schreiben, ohne Überschriften, Zwischenüberschriften, Absätze oder Kapitel, und den kompletten Text in nur einen großen Block packen würden, wäre diese Buch dann genauso leicht zu lesen wie es momentan der Fall ist?

Die saubere Formatierung von Code hat absolut nichts mit dem Kompilierungsprozess zu tun und beeinflusst in keinster Weise die Ausführung des kompilierten Programms, sie dient ausschließlich der einfachen Lesbarkeit des Codes. Weshalb dann damit aufhalten? Ich garantiere Ihnen, dass Sie in Zukunft immer wieder bestimmte Quelltexte überarbeiten werden. Manchmal weil Sie ihn noch nicht fertiggestellt haben und manchmal weil Sie ihn in der Funktionalität erweitern möchten. Jedes Mal müssen Sie dann Ihren Code lesen und verstehen was er macht. Wenn Sie dann hässlichen, unsauberen Code vorliegen haben, garantiere ich Ihnen, dass Sie Schwierigkeiten haben werden diesen Code zu lesen und daran weiter zu arbeiten.

Wenn ein Fehler in Ihrem Programm auftritt und der Code nicht gut organisiert ist, haben Sie ebenfalls eine schwere Zeit vor sich, in der Sie den Fehler herunterbrechen müssen um die ursächliche Stelle in Ihrem Code zu finden. Ein Programm das Sie heute gut verstehen kann morgen schon völlig unklar sein. Wenn Sie dann schon eine Weile brauchen um sich wieder einzulesen, wie lange dauert es dann erst bei unsauberem Code?

Die Arbeit in Teams ist ein weiteres gutes Argument für das saubere formatieren des Quelltextes. Wenn Sie in einem Team an einem Projekt arbeiten und müssen Code von anderen Teilnehmern berichtigen oder erweitern, dann ist es essentiell dass alle Teammitglieder die gleichen Formatierungsstandards benutzen. Dadurch können alle Teammitglieder den Code von Anderen schnell und einfach verstehen, und effektiv am Projekt weiterarbeiten.

Die Wichtigkeit von Kommentaren

Das erste was ich erwähnen möchte wenn ich über guten Programmierstil spreche, ist die Verwendung von Kommentaren. Bis jetzt haben wir keine Kommentare in unseren Beispielen verwendet. Ich dachte ich warte bis zu diesem Kapitel um sie zweckmäßig zu beschreiben, da ich denke das Kommentare einer der wichtigsten Aspekte für das schreiben von gutem Code darstellen.

Um es einfach zu formulieren, Kommentare sind Textzeilen die Ihrem Code hinzugefügt werden und zum beschreiben oder dokumentieren der Funktion des Codes dienen. Sie werden niemals in das fertige Executable hinein kompiliert und haben auch sonst keine Auswirkungen auf das Programm. Sie können so viele Kommentare verwenden wie Sie wollen, da sie sich niemals negativ auf die Ausführungs-geschwindigkeit des Programmes auswirken.

Kommentare sind eine Möglichkeit, um detaillierter zu beschreiben was Sie schreiben, oder auch um einen Überblick zu verschiedenen Teilen Ihres Programmes zu geben, wie sie funktionieren und wie sie mit dem Rest des Programmes interagieren. Hier ein kleines Beispiel:

```
;Die Prozedur Pi() wird aufgerufen, um den Wert von PI zurückzugeben.
;Diese Berechnung ist nur auf sechs Dezimalstellen genau.

Procedure.f Pi()
  ProcedureReturn 4 * (4 * ATan(1 / 5) - ATan(1 / 239))
EndProcedure

KreisDurchmesser.i = 150

Debug "Ein Kreis mit einem Durchmesser von " + Str(KreisDurchmesser) + "mm"
Debug "hat einen Umfang von " + StrF(KreisDurchmesser * Pi()) + "mm."
```

In diesem Beispiel habe ich Kommentare zum beschreiben der Prozedur 'PI()' verwendet, und was noch wichtiger ist, ich habe ihre begrenzte Genauigkeit auf sechs Dezimalstellen erwähnt.

Das Erstellen von Kommentaren ist sehr einfach, da alle Kommentare mit einem Semikolon (';') beginnen, und das war es auch schon:

```
;Das ist ein Kommentar
```

Kommentare können wirklich überall in Ihrem Quelltext stehen, sogar in der gleichen Zeile mit einem anderen Befehl, wie hier:

```
Procedure.i ZaehleZusammen(a.i, b.i) ;Addiert 2 Zahlen und gibt das Ergebnis zurück
  ProcedureReturn a + b
EndProcedure
```

Sie müssen immer an das einleitende Semikolon denken, dieses definiert den Beginn eines Kommentars. Der Kommentar ist dann genauso lang wie die Zeile, mit dem Beginn der neuen Zeile endet der Kommentar. Wenn Sie den Kommentar in der nächsten Zeile fortsetzen möchten, dann müssen Sie diese Zeile mit einem Semikolon beginnen.

Um Code zweckmäßig zu kommentieren, sollten Sie die Kommentare signifikant gestalten und nicht auf die Verlockung hereinfallen einfach nur den Code mit den Kommentaren zu wiederholen. Kommentare sollten möglichst kurz sein, aber lang genug, dass sie alles nötige verständlich erklären. Die einfachste Regel beim schreiben von Kommentaren: Sie sollten sich einbilden, das Sie die etwas komplizierteren Stellen in Ihrem Programm einer fremden Person erklären. Wenn Sie in sechs Monaten Ihren Code wieder öffnen, werden Sie froh sein Kommentare zu finden die Ihnen helfen Ihre Arbeit zu verstehen.

Kommentare können für folgendes verwendet werden:

1). Hinzufügen von Lizenz- und Copyright Informationen.
2). Erklären warum ein bestimmter Schritt unternommen wurde.
3). Hinzufügen von Notizen, wo der Code noch verbessert werden kann.
4). Erklären der Funktion von komplizierten Prozeduren.
5). Erklären der Interna von Prozeduren durch zeichnen von Grafiken oder Formularen mittels ASCII-Art.

Meine Code Gestaltung

Das generelle Layout meiner Quelltexte und die Art wie ich bestimmte Sachen strukturiere, basiert auf dem Standard Code Format von vielen Programmiersprachen. Das ganze Buch wurde in diesem Stil geschrieben, der wie ich denke ein leicht zu lesendes (und lernendes) Format hat. Hier werde ich erklären warum ich das so getan habe und zeige Ihnen wie ich Code schreibe, um Ihnen einen guten Einstieg zum schreiben von sauberen und lesbaren Programmen zu geben. Ich hoffe Sie finden diesen Stil hilfreich und werden ihn weiter benutzen.

Variablen, Arrays, Listen und Prozeduren

Ich bezeichne diese Programmelemente üblicherweise mit klaren, akkuraten, aussprechbaren und beschreibenden Namen, basierend auf der 'CamelHump' (Kamelhöcker) Notation. Diese Notation ist einfach zu verstehen. Der erste Buchstabe wird groß geschrieben, dem folgen dann Kleinbuchstaben. Jedes weitere Wort im Namen beginnt dann wieder mit einem Großbuchstaben. Die Großbuchstaben dienen als Separatoren für die einzelnen Wörter, da Sie keine Leerzeichen in den Namen verwenden dürfen. Die

Großbuchstaben sorgen dafür, dass die Wörter aussehen als hätten sie Höcker, was den Namen 'CamelHump' Notation geprägt hat. Ich bevorzuge dieses Format für alle Variablen, Arrays und Listen, da ich denke, dass sie dadurch leichter lesbar sind.

Einige Beispiele:

```
AnzahlDerTage.i = 365
Dim MonateDesJahres.s(11)
NewList Tage.s()
```

Konstanten
Ich formatiere Konstanten im Standard C und Windows API Stil, also nur Großbuchstaben. Wenn ich einzelne Wörter auseinander halten möchte, separiere ich diese mit Unterstrichen.

Ein Beispiel:

```
GESCHWINDIGKEIT_DES_LICHTES = 299792458 ;Meter pro Sekunde
```

Strukturen
Für Strukturen verwende ich den gleichen Standard wie für die Konstanten, ich verwende nur Großbuchstaben in den Namen. Das entspricht auch wieder dem Standard C sowie dem Windows API Format. Wie bei den Konstanten verwende ich zum separieren von einzelnen Wörtern auch wieder Unterstriche. Beachten Sie, der folgende Strukturname hat nur Großbuchstaben.

Ein Beispiel:

```
Structure BENUTZER_SERVICE
  ServiceTyp.s
  ServiceCode.l
EndStructure
```

Einrückungen
Einrückungen sind eine gute Möglichkeit Quelltexte zu strukturieren. Durch sie wird es einfach, den Anfang und das Ende von Prozeduren, Schleifen und bedingten Anweisungen zu erkennen. Einrückungen werden sehr häufig verwendet, um Ihnen eine visuelle Hilfestellung zum lesen des Codes zu geben. Hier ein Beispiel für gute Einrückung:

```
; Gibt einen String zurück, der einen gerundeten Float Wert enthält.
; 'Zahl' = Zahl die gerundet und als String zurückgegeben werden soll.
; 'Dezimalstellen' = Anzahl der Dezimalstellen, auf die gerundet werden soll.

Procedure.s StrFRunden(Zahl.f, Dezimalstellen.i)

  Protected R.f
  Protected T.f

  If Dezimalstellen < 0
    Dezimalstellen = 0
  EndIf

  R.f = 0.5 * Pow(10, -1 * Dezimalstellen)
  T.f = Pow(10, Dezimalstellen)

  ProcedureReturn StrF(Int((Zahl + R) * T) / T, Dezimalstellen)

EndProcedure

Debug StrFRunden(3.1415927, 4)
```

In diesem Beispiel können Sie sehen, das ich den Code mit Tabulatoren eingerückt habe. Dadurch wird der Code zwischen den Start- und End- Schlüsselwörtern, wie zum Beispiel Prozeduren und 'If' Anweisungen, klar abgesetzt. Auf diese Art rücke ich den Code für Schleifen, Prozeduren, 'If' und 'Select' Anweisungen sowie Aufzählungen ein. Einrückungen sind bei verschachtelten Anweisungen besonders hilfreich, wie hier:

```
For x = 0 To 2
  For y = 0 To 2
    z.i = 0
    While z <= 10
```

```
      Debug x * y + z
      z + 1
    Wend
  Next y
Next x
```

Hier können Sie das Start- und End- Schlüsselwort jeder Schleife gut erkennen, und sie sehen welcher Code zur jeweiligen Schleife gehört. Für jeden Codeblock wird nur ein Tabulator als Einrückung verwendet, damit der Code nicht zu weit vom Start- und End- Schlüsselwort entfernt ist, was den Quelltext auch schwerer lesbar machen kann.

Wenn Sie in die Zeile 'For y = 0 To 2' schauen, können Sie einfach den Code weiter nach unten verfolgen und finden die Zeile 'Next y'. Das ist der Beginn und das Ende einer 'For' Schleife. Wenn wir auf diese Schleife blicken wissen wir, dass der gesamte Code, der nach rechts eingerückt ist, sich innerhalb der Schleife befindet. Das ganze wird mit wachsender Programmgröße erheblich nützlicher.

Mehrere Befehle in der selben Zeile

Ich schreibe manchmal mehrere Befehle in die selbe Zeile, um den Quelltext etwas kleiner zu machen. Das ist mit dem Doppelpunkt (':') Zeichen möglich. Der Doppelpunkt, wenn verwendet, teilt dem Compiler mit, dass eine neue Zeile begonnen hat und behandelt den nächsten Befehl als stünde er in einer separaten Zeile. Schauen Sie auf dieses Beispiel:

```
Dim Voegel.s(3)
Voegel(0)="Spatz" : Voegel(1)="Zaunkönig" : Voegel(2)="Kuckuck" : Voegel(3)="Eule"
```

Manche Programmierer runzeln die Stirn bei diese Art der Code Formatierung, da sie die Lesbarkeit beeinträchtigen kann. Ich denke, dass es manchmal hilfreich sein kann, wenn es nicht zu häufig verwendet wird.

Aufteilen von Quelltexten

Wenn ich ein Programm schreibe und es beginnt größer zu werden, dann tendiere ich dazu es in separate Quelltext Dateien aufzuteilen. Diese Dateien sind Standard PureBasic Dateien aber sie haben entweder die Endung '*.pb' oder '*.pbi'. '*.pb' Dateien sind die Standard PureBasic Quelltextdateien, während '*.pbi' Dateien PureBasic Include-Dateien sind. Diese zwei Arten von Dateien sind absolut identisch und werden beide korrekt von der IDE geöffnet. Die '*.pbi' Dateiendung ist reine Kosmetik, um schnell die Include Dateien von den Dateien des Hauptprogramms unterscheiden zu können, wenn sich diese im gleichen Ordner befinden.

In diesen separaten Dateien verwalte ich Teile des Hauptprogramms und dann binde ich diese alle mittels des 'IncludeFile' oder des 'XIncludeFile' Schlüsselwortes in den Hauptquelltext ein.

Wenn ich zum Beispiel sehr viele Prozeduren definiert habe, erstelle ich eine separate Quelltext Datei mit dem Namen 'Prozeduren.pbi', in die ich alle meine definierten Prozeduren hineinschiebe. Im Kopf meines Hauptquelltextes binde ich diesen Prozeduren Code dann folgendermaßen ein:

```
IncludeFile "Prozeduren.pbi"
```

Das hat den Effekt, das alle Code Zeilen aus der Datei 'Prozeduren.pbi' in den Hauptquelltext eingefügt sind, beginnend an dieser Stelle. Wo immer der 'IncludeFile' Befehl auftaucht, wird genau an dieser Stelle der Code aus der Include-Datei eingefügt. Dieses einbinden findet vor der Kompilierung statt, so das der Compiler nur eine große Quelltext Datei sieht.

Wenn Sie an vielen Stellen in Ihrem Hauptquelltext Include Dateien mit dem 'IncludeFile' Befehl einfügen und immer den selben Dateinamen verwenden, werden Sie doppelte Kopien von ihrem Code im Hauptquelltext erstellen. Wenn Sie das vermeiden wollen (führt bei der doppelten Definition von Prozedurnamen zu Fehlern), müssen Sie das 'XIncludeFile' Schlüsselwort verwenden. Dieses wird auf die gleiche Weise wie oben angewandt:

```
XIncludeFile "Prozeduren.pbi"
```

Die 'XIncludeFile' Zeile wird den Code aus 'Prozeduren.pbi' nur einfügen wenn diese Zeile zuvor noch nicht verwendet wurde. Ich verwende den 'IncludeFile' Befehl sehr selten, üblicherweise wenn ich nur eine Datei in das Programm einbinden muss. Ich bevorzuge die Benutzung von 'XIncludeFile', da es die Möglichkeit von auftretenden Fehlern verringert

Richtiges Beenden eines Programms

Um ein Programm richtig zu beenden, sollten Sie immer das 'End' Schlüsselwort verwenden. Dieses schließt alles in Ihrem Programm ordnungsgemäß und gibt allen Speicher frei, der von ihm verwendet wurde. Optional können Sie bei Bedarf auch einen Rückgabewert definieren. Die Syntax zur Verwendung ist einfach, schreiben Sie das Schlüsselwort einfach an die Stelle, an der Sie das Programm beenden möchten. In einem Programm können auch mehrere 'End' Anweisungen stehen, wenn Sie mehrere potentielle Programmende Situationen haben.

Ohne Rückgabewert:

```
End ; Beendet das Programm sofort und gibt allen verwendeten Speicher frei.
```

Mit Rückgabewert:

```
End 1 ; Beendet das Programm sofort, gibt den Speicher frei und meldet 1 zurück.
```

Goldene Regeln zum schreiben von gut lesbarem Code

Hier ist eine Liste mit goldenen Regeln die ich beim schreiben eines Programms beachte. Ich halte mich an diese Regeln, auch wenn ich nur kleine Programme schreibe. Wenn Sie dieser Liste folgen, werden Sie ebenfalls guten, sauberen und verständlichen Code schreiben. Wenn Sie sich bei Ihrer Arbeit an einen Standard halten, bekommt Ihr Code eine klare und bündige Struktur für jeden der ihn liest und er wirkt professioneller.

1). Geben Sie allen Variablen, Prozeduren, Arrays, usw. klare, akkurate, aussprechbare und beschreibende Namen.
2). Gruppieren Sie logisch verknüpfte Variablen oder Daten in Arrays oder Strukturen.
3). Prozeduren sollten eine Funktion verrichten und diese sollten sie gut verrichten.
4). Benutzen Sie Einrückungen, um die Code Struktur sichtbar zu machen.
5). Verwenden Sie Klammern in Ausdrücken, Missinterpretationen zu vermeiden.
6). Verwenden Sie Leerzeilen, um verschiedene Prozeduren und andere Code Blöcke zu separieren.
7). Versuchen Sie die Schlüsselwörter 'Goto' oder 'Gosub' zu vermeiden.
8). Verwenden Sie Kommentare, um anderen (oder Ihnen) Hilfestellung beim Verständnis des Codes zu geben.
9). Versuchen Sie nicht, schlechten Code mit Kommentaren zu dokumentieren, Schreiben Sie den Code in richtiger Form noch einmal.
10). Wenn Sie in einem Team arbeiten, vereinbaren Sie vor dem Start Formatierungsregeln und halten Sie sich daran

Minimieren von Fehlern und deren Behandlung

In diesem Kapitel werde ich über Methoden berichten, die Ihnen helfen werden Fehler in Ihren Programmen zu finden. Auch die erfahrensten und fähigsten Programmierer können Fehler machen oder vergessen gängige Sachen zu behandeln, oder übersehen sie vielleicht einfach. Hier präsentiere ich ein paar gute Vorgehensweisen zum arbeiten, gespickt mit wertvollen Tipps und Tricks die ihre Wachsamkeit fördern und die Chancen für auftretende Fehler minimieren.

Verwenden einer Fehlerbehandlungsprozedur

Wenn Sie PureBasic für große Projekte verwenden denken Sie vielleicht, dass Sie jede Menge Prüfungen auf einen Wahren Wert vornehmen. Das ist so, weil der Großteil der Befehle einen Wert zurückgibt. Dieser Wert ist bei Erfolg nahezu immer ein Integer Wert größer '0'. Manche Programmierer verwenden 'If' Anweisungen, um Befehle auf die Rückgabe von True zu überprüfen, bevor sie in ihrem Programm weiterarbeiten. Erinnern Sie sich an Kapitel 4, alle Werte ungleich '0' sind für PureBasic ein logisches True.

Einige Programmierer verwenden 'If' Anweisungen zum prüfen ob alles richtig verlaufen ist, bevor sie weiter Verfahren. Ich denke dieser Ansatz ist nur für kleinere Programme brauchbar, da er zu viel Verwirrung in größeren Projekten führen kann, wenn Sie den Überblick in vielen verschachtelten 'If' Anweisungen verlieren. Ich neige dazu eine Prozedur als Fehlerbehandlungsprozedur anstelle von 'If' Anweisungen zu verwenden. Das macht nicht nur Ihren Code leichter lesbar, sondern Sie können der Prozedur auch benutzerdefinierte Nachrichten übergeben, um den Benutzer darüber zu informieren wo genau der Fehler auftrat und welcher Befehl ihn verursacht hat. Um beide Seiten zu zeigen, hier ein Beispiel das 'If' Anweisungen verwendet um Befehle zu prüfen:

```
#TEXTDATEI = 1

If ReadFile(#TEXTDATEI, "TextDatei.txt")
  Debug ReadString(#TEXTDATEI)
  CloseFile(#TEXTDATEI)
Else
  MessageRequester("Fehler", "Konnte die Datei nicht öffnen: 'TextDatei.txt'.")
EndIf
```

Hier vergewissere ich mich, ob ich die Datei 'TextDatei.txt' lesen kann. Wenn dies funktioniert, dann lese ich einen String aus ihr. Während diese Methode für kleine Programme brauchbar ist, bevorzuge ich für größere Projekte diese Möglichkeit:

```
#TEXTDATEI = 1

Procedure FehlerFang(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

FehlerFang(ReadFile(#TEXTDATEI, "Datei.txt"), "Konnte 'Datei.txt' nicht öffnen.")
Debug ReadString(#TEXTDATEI)
CloseFile(#TEXTDATEI)
```

Hier habe ich eine Prozedur mit dem Namen 'FehlerFang()' verwendet, um die Rückgabewerte von Befehlen zu überwachen. Der Erste Parameter ('Ergebnis') nimmt den Rückgabewert des Befehls entgegen. Wenn ein Befehl auf diese Weise an die Prozedur übergeben wird, dann wird immer zuerst der Befehl ausgeführt, bevor die Prozedur aufgerufen wird. Das stellt sicher, dass alle Rückgabewerte korrekt übergeben werden. Der zweite Parameter ('Text.s') ist der String, den Sie anzeigen möchten wenn der Befehl '0' zurückgibt. Lassen Sie uns das noch etwas vertiefen.

Was passiert, wenn ich diesen Befehl an die 'FehlerFang()' Prozedur übergebe:

```
ReadFile(#TEXTDATEI, "Datei.txt")
```

Wenn die Datei 'Datei.txt' nicht auf ihrer Festplatte existiert oder es tritt ein anderer Fehler auf, dann gibt der 'ReadFile()' Befehl '0' zurück. Dieser Wert von '0' wird dann dem ersten Parameter der Fehlerbehandlungsprozedur zugewiesen. In der Prozedur wird überprüft ob dieser Wert gleich '0' ist. Wenn dies der Fall ist, wird die Fehlermeldung, die als zweiter Parameter übergeben wurde, in einem 'MessageRequester' angezeigt und das Programm endet. Diese Vorgehensweise ist praktisch, da Sie zu jedem Befehl in dem ein Fehler auftritt eine kurze, klare Fehlermeldung ausgeben können.

Wenn der Befehl, der an die Fehlerbehandlungsprozedur übergeben wurde, einen Wert größer '0' zurückliefert, dann war dieser erfolgreich und die Fehlerbehandlungsprozedur führt keine Aktion aus.

Die in obigem Beispiel vorgestellte Prozedur ist vielleicht etwas zu viel des Guten für so ein kleines Beispiel, aber diese Art der Verwendung ist nicht das, wofür sie hauptsächlich benutzt wird. Die Verwendung einer Fehlerbehandlungsprozedur ist erst in größeren Programmen wirklich sinnvoll, wo Sie viele Befehle nacheinander überprüfen müssen. Wenn die Prozedur definiert ist, können Sie damit so viele Befehle überprüfen wie Sie möchten und jeder kann, anders als mit einer 'If' Anweisung, in einer Zeile geprüft werden. Schauen Sie auf diese Beispiel, in dem ich mehrere Befehle prüfe:

```
FehlerFang(InitEngine3D(), "InitEngine3D() Initialisierungsfehler!")
FehlerFang(InitSprite(), "InitSprite() Initialisierungsfehler!")
FehlerFang(InitKeyboard(), "InitKeyboard() Initialisierungsfehler!")
FehlerFang(OpenScreen(800, 600, 32, "Spiel"), "Konnte den Screen nicht öffnen!")
```

Können Sie sich vorstellen, all diese Befehle mit 'If' Anweisungen zu prüfen? Es würde ein verschachtelter 'If' Alptraum werden, und das ist noch kein fertiges Programm. Die Verwendung einer Fehlerbehandlungsprozedur macht ihren Code hübscher, sauberer und lesbarer.

Wenn Sie so vorgehen, sind Fehler leichter anzuzeigen und unerwartete Probleme sind leichter zu handhaben. Das einzige Problem bei dieser Vorgehensweise ist der Name 'FehlerFang', der in jeder Zeile erscheint, was manche aufdringlich finden.

Verwenden Sie den 'EnableExplicit' Befehl

Für die einen ist dieser Befehl ein Geschenk des Himmels und für die anderen ist er eine Behinderung, was daran liegt, das er komplett optional verwendet werden kann. Dieser Befehl schaltet die explizite Variablen Deklaration in Ihrem ganzen Programm ein. Was bedeutet das? In Kapitel 2 habe ich Ihnen erklärt, das einer Variable ohne Suffix immer der Standardtyp zugewiesen wird, in der Regel Integer. Wenn das 'Define' Schlüsselwort verwendet wurde, um einen anderen Standardtyp festzulegen, dann wird jeder typenlosen (ohne Suffix) Variable dieser neue Standardtyp zugewiesen. Der 'EnableExplicit' Befehl beendet dieses Verhalten, nach seinem Aufruf müssen alle Variablen mit einem Geltungsbereich und einem festen Typ definiert werden. Lassen Sie mich demonstrieren, wo dieser Befehl wirklich nützlich ist.

Sagen wir zum Beispiel, ich habe eine Prozedur, der eine Variable von entscheidendem Wert übergeben werden muss und das Ergebnis der Prozedur wird gebraucht. Dies könnte folgendermaßen aussehen:

```
MengeProWoche.i = 1024

Procedure BerechneMengeProJahr(Wert.i)
  MengeProJahr.i = Wert * 52
  ProcedureReturn MengeProJahr
EndProcedure

Debug BerechneMengeProJahr(MengePerWoche)
```

Das sieht so weit in Ordnung aus, und wenn Sie schnell über das Beispiel lesen, sehen Sie keine Probleme. Wenn Sie das Programm starten, gibt die Prozedur 'BerechneMengeProJahr()' den Wert '0' zurück. Das ist offensichtlich nicht richtig, da wir der Meinung sind, die Variable 'MengeProWoche' mit einem Wert von '1024' übergeben zu haben. Wenn Sie etwas genauer auf den Prozeduraufruf schauen, sehen Sie, das ich tatsächlich eine Variable mit dem Namen 'MengePerWoche' übergeben habe. Beachten Sie, dass diese Variable falsch geschrieben ist, weshalb sie als neue Variable behandelt wird. Wenn Sie eine solche nicht zuvor definierte Variable übergeben, dann erstellt PureBasic automatisch eine neue und übergibt diese. Wenn einer Variablen bei der Erstellung kein Wert oder Typ übergeben wird, dann erhält diese immer den Standardtyp und den Wert '0'. Die neue Variable mit dem Namen 'MengePerWoche' in obigem Beispiel hat deshalb den Wert '0'.

Die automatische Variablendefinition, wie hier geschehen, wird von einigen Programmierern als fahrlässig und fehlerträchtig betrachtet. Dieses Standardverhalten kann mit dem 'EnableExplicit' Befehl abgestellt werden, wodurch der Compiler angewiesen wird strenger auf die Variablendefinition zu achten. Wenn wir den Befehl in obigem Beispiel verwendet hätten, hätten wir einige Informationsmeldungen erhalten, das einige Variablen nicht richtig definiert wurden. Das bedeutet, dass alle nach dem 'EnableExplicit' Schlüsselwort verwendeten Variablen mit einem der Variablendefinitions-Schlüsselwörter definiert werden müssen. Dies sind 'Define', 'Global', 'Protected', 'Static' und 'Shared', die ich in Kapitel 2 und 6 erklärt habe.

Wenn wir zu unserem obigen Beispiel zurückkehren, alle Variablen richtig definieren und das Programm neu starten, dann lenkt der Compiler unsere Aufmerksamkeit auf die letzte Zeile, indem er uns eine Nachricht präsentiert, dass die Variable 'MengePerWoche' nicht richtig definiert wurde. Das verwirrt etwas, da ich das bereits an einer früheren Stelle im Programm erledigt habe. Bei genauerer Betrachtung stelle ich dann fest, das die Variable falsch geschrieben wurde und berichtigt werden muss. So sieht das obige Beispiel aus, wenn wir den 'EnableExplicit' Befehle verwenden und alle Variablen richtig definiert haben.

```
EnableExplicit

Define MengeProWoche.i = 1024

Procedure BerechneMengeProJahr(Wert.i)
  Protected MengeProJahr.i = Wert * 52
  ProcedureReturn MengeProJahr
EndProcedure

Debug BerechneMengeProJahr(MengeProWoche)
```

Der hoffentliche Einsatz dieses Befehls bewahrt Sie vor Fehlern die durch falsch geschriebene Variablennamen entstehen, da der Compiler bei jeder nicht definierten Variable eine Meldung anzeigt, dass Sie den Geltungsbereich und die Definition klären müssen. Dies gibt Ihnen die Möglichkeit, diese Fehler zu korrigieren und Sie haben nicht das Risiko, durch diese Fehler wichtige Werte zu verlieren.

Sie können jederzeit zum PureBasic Standardverhalten zurückkehren indem Sie den 'DisableExplicit' Befehl verwenden.

Definieren von Variablen mit dem 'Define' Befehl

Der 'Define' Befehl kann auf zwei Arten verwendet werden. Die erste Möglichkeit ist das setzen des Standardtyps von typenlosen Variablen, wie ich es bereits in Kapitel 2 erklärt habe. In diesem Fall wird das 'Define' Schlüsselwort mit einem Suffix verwendet, das den neuen Standardtyp definiert, wie hier:

```
Define.s
MeinString = "Hallo"
```

Die zweite Möglichkeit ist das definieren von Variablen, nachdem ein 'EnableExplicit' Befehl verwendet wurde. Nach der Verwendung des 'EnableExplicit' Befehls müssen alle Variablen in Ihrem Programm klar definiert werden. In diesem Fall wird das 'Define' Schlüsselwort so verwendet:

```
EnableExplicit
Define MeineVariable.b = 10
```

Beachten Sie, das bei dieser Art der Verwendung an das 'Define' Schlüsselwort kein Suffix angehängt werden darf, da wir den Variablentyp mit dem Suffix der Variable definieren.

Einführung in den PureBasic Debugger

PureBasic stellt einen voll funktionsfähigen Debugger zur Verfügung, der Ihnen hilft Fehler und Bugs in Ihrem Programm zu finden. Dieser Debugger ist unbezahlbar, da er Ihnen die Möglichkeit gibt, den Programmfluss zu steuern und Ihnen jederzeit während des Programmlaufs einen Blick auf die momentanen Werte von Variablen, Arrays und Listen gewährt. Er stellt weiterhin fortgeschrittene Funktionen für Assembler Programmierer zur Verfügung, die es ermöglichen CPU Register zu prüfen und zu modifizieren, sowie das Einsehen von Werten, die mit bestimmten Speicheradressen verknüpft sind, ermöglichen. Der Ehemals enthaltene CPU Monitor, mit dem sich die Prozessorauslastung überprüfen lässt, wurde in Version 4.30 entfernt, da alle Betriebssysteme einen solchen zur Verfügung stellen. Unter Windows zum Beispiel ist dieser über den Taskmanager (Systemleistung) erreichbar.

Wenn Sie Ihr Programm starten und der Debugger trifft auf einen Fehler, dann wird das Programm angehalten und in der IDE die verursachende Zeile rot markiert. Weiterhin wird der Fehler im Fehlerbericht der IDE dokumentiert und in der Statusleiste der IDE angezeigt. Wenn ein Fehler auf diese Art abgefangen wurde, können Sie die Programm Kontrollfunktionen verwenden oder das laufende Programm beenden. Zum beenden des laufenden Programms verwenden Sie den 'Programm beenden' Befehl (Menü: Debugger → Programm beenden) oder die entsprechende Schaltfläche in der Werkzeugleiste. Wenn der Debugger abgeschaltet ist, werden keine Fehler abgefangen und können zu einem Programmabsturz führen.

Wenn Sie in der IDE ein neues Programm erstellen, ist der Debugger standardmäßig eingeschaltet. Das können Sie an der gedrückten Debugger Schaltfläche in der IDE Werkzeugleiste erkennen, siehe Abb. 23. Wenn diese Schaltfläche gedrückt dargestellt wird ist der Debugger eingeschaltet, in nicht gedrücktem Zustand ist der Debugger abgeschaltet. Diese Schaltfläche ist auch wieder eine Schnellzugriffsvariante für den 'Debugger verwenden' Menübefehl (Menü: Debugger → Debugger verwenden), der ebenfalls seinen Zustand wechselt. Der Debugger kann auch in den Compiler Optionen Ihres Programms eingeschaltet werden (Menü: Compiler → Compiler-Optionen... → Kompilieren/Starten → Debugger einschalten). All diese Möglichkeiten den Debugger einzuschalten sind miteinander verknüpft. Wenn eine geändert wird, übernehmen alle anderen den geänderten Status.

Das PureBasic Paket für Windows wird mit drei verschiedenen Varianten des Debuggers ausgeliefert. Diese können alle zum Debuggen Ihres Programms verwendet werden, aber nicht alle haben die gleiche Funktionalität. Der erste ist der eingebaute Debugger, der den größten Funktionsumfang bietet. Er wird standardmäßig verwendet, da er direkt in die IDE integriert ist. Da der IDE Debugger nur eine Kopie des Programms gleichzeitig Debuggen kann, haben Sie die Möglichkeit eine zweite Kopie des Programms mit dem ebenfalls installierten externen Debugger zu starten. Diese eigenständige Version hat nahezu den gleichen Funktionsumfang wie der integrierte Debugger, da er aber getrennt von der IDE läuft geht ein wenig Effizienz verloren, da der direkte Zugriff auf die IDE fehlt. Ein Vorteil des externen Debuggers soll nicht verschwiegen werden, er eignet sich, anders als der IDE Debugger, auch zum Remote Debuggen über eine Netzwerkverbindung. Der dritte Debugger läuft nur auf der Konsole. Der primäre Verwendungszweck ist der Betrieb auf nicht grafischen Umgebungen, wie zum Beispiel ein Text basiertes Linux System oder die Remote Entwicklung von Anwendungen mit Netzwerk Clients, die das SSH Protokoll verwenden. Die verfügbaren Debugger können in den IDE Einstellungen (Menü: Datei → Einstellungen → Debugger → Debugger-Typ auswählen) ausgewählt werden.

Auch wenn der Debugger ein großartiges Werkzeug ist um Probleme herunterzubrechen, so hat diese Funktionalität auch ihren Preis. Mit eingeschaltetem Debugger laufen Ihre Programme wesentlich langsamer als im abgeschalteten Zustand. Das sollte kein Problem sein, da die meisten fertigen Programme ohne Debugger kompiliert werden, um ein Maximum an Geschwindigkeit und Kompaktheit zu erreichen. Diesen Umstand müssen Sie im Hinterkopf behalten, wenn Sie zeitkritische Anwendungen entwickeln oder bestimmte Codeabschnitte zeitlich abstimmen müssen, usw.

Wenn Sie den Debugger zum Debuggen Ihres Codes benötigen, bestimmte Abschnitte aber nicht mehr auf diesen angewiesen sind, können Sie die Befehle 'DisableDebugger' und 'EnableDebugger' verwenden. Die Verwendung ist selbsterklärend, der 'DisableDebugger' Befehl schaltet den Debugger ab dieser Zeile aus und der 'EnableDebugger' reaktiviert ihn wieder. Wenn Sie den Debugger ausschalten, werden Sie bemerken, das damit auch alle 'Debug' Befehle deaktiviert werden. Das ist so, weil die 'Debug' Befehle ein Teil des Debuggers sind und mit ausgeschaltetem Debugger nicht kompiliert werden.

Verwenden des Debuggers

Die Debugger Funktionen können jederzeit während des Programmlaufs verwendet werden und können über das Debugger Menü oder über die damit verknüpften Schaltflächen in der Werkzeugleiste aufgerufen werden. Auf den Fehlerbericht (Menü: Debugger → Fehlerbericht) kann ebenfalls jederzeit zugegriffen werden. Während Sie den Debugger verwenden sind alle von Ihrem Programm benötigten Quelltext Dateien schreibgeschützt, bis Sie das Programm beendet haben. Das stellt sicher, das der momentan verwendete Code in nicht verändert wird und bietet somit eine gewisse Versionskontrolle für Ihren Code.

Die Debugger Werkzeugleisten Schaltflächen

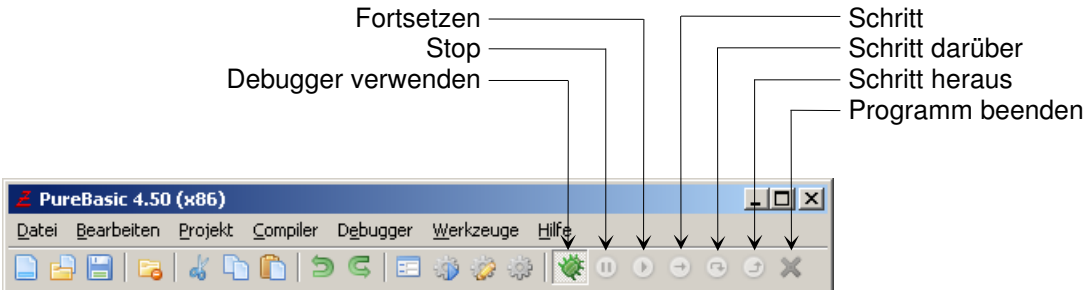


Abb. 23

Um Ihnen einen Überblick zum Debugger zu verschaffen, beginne ich mit der Erklärung der verfügbaren Programm Kontrollfunktionen. Diese Funktionen bieten die Möglichkeit, ihr Programm jederzeit anzuhalten und aktuelle Werte in Variablen, Arrays oder Listen einzusehen. Er erlaubt außerdem Ihr Programm schrittweise (Zeile für Zeile) abzuarbeiten, damit Sie genau verfolgen können, wie Ihr Programm arbeitet. Der Status eines jeden laufenden Programms wird in der Statusleiste der IDE und im Fehlerbericht angezeigt. Die Debugger Programm Kontrollfunktions-Schaltflächen in der Werkzeugleiste sind in Abb. 23 zu sehen. Dies sind gespiegelte Befehle aus dem Debugger Menü.

Sie können Ihr Programm jederzeit anhalten um die Debugger Steuerelemente zu verwenden, indem Sie das 'CallDebugger' Schlüsselwort in Ihrem Code verwenden. Benutzen Sie den 'Stop' Befehl vom Debugger Menü (Menü: Debugger → Stop) oder drücken Sie die 'Stop' Schaltfläche in der Werkzeugleiste, während ihr Programm läuft. Sie können auch Haltepunkte verwenden um die Programmausführung anzuhalten. Um einen Haltepunkt in der IDE zu verwenden, setzen Sie den Cursor in die Zeile, in der Sie das Programm anhalten möchten und die Kontrolle an den Debugger übergeben wollen. Dann wählen Sie den 'Haltepunkt' Menübefehl (Menü: Debugger → Haltepunkt) um einen Haltepunkt hinzuzufügen. Sie werden feststellen, das ein kleines Rechteck in der IDE Spalte mit den Zeilennummern erscheint. Das ist ein visueller Hinweis, um zu zeigen wo Haltepunkte definiert sind. Wenn das Programm das nächste mal gestartet wird, wird es in der Zeile anhalten in der der Haltepunkt definiert ist und die Debugger Programm Kontrollfunktionen sind verfügbar. Sie haben dann die Möglichkeit jegliche aktuelle Datenwerte zu betrachten oder sich Schrittweise durch ihr Programm zu arbeiten, um den Code näher zu analysieren. Hier ist eine kurze Beschreibung der Programm Kontrollfunktionen:

'Stop'
Hält das Programm an und zeigt die aktuelle Zeile an.

'Fortsetzen'
Setzt die Programmausführung fort, bis ein neuer Haltebefehl auftaucht.

'Schritt'
Führt eine Zeile im Programm aus und hält das Programm vor der nächsten Zeile wieder an.

'Schritt darüber'
Führt eine Zeile im Programm aus und hält das Programm vor der nächsten Zeile wieder an. Erfolgt in der ausgeführten Zeile ein Prozeduraufruf, wird diese im unterschied zu 'Schritt', komplett abgearbeitet und das Programm hält in der Zeile nach dem Prozeduraufruf.

'Schritt heraus'
Führt den verbleibenden Code innerhalb der aktuellen Prozedur aus und hält nach Rückkehr aus dieser erneut an. Wenn sich die aktuelle Zeile nicht innerhalb einer Prozedur befindet, wird ein normaler 'Schritt' ausgeführt.

'Programm beenden'
Beendet das Programm sofort und schließt alle damit verknüpften Debugger Fenster.

Hier ein kleines Beispiel um die Programmausführung mit dem 'CallDebugger' Befehl anzuhalten:

```
CallDebugger
For x.i = 1 To 10
  Debug x
Next x
End
```

Wenn das obige Programm mit dem 'CallDebugger' Befehl angehalten wurde, können Sie sich einfach Schrittweise durch das Programm arbeiten, indem Sie die 'Schritt' Schaltfläche in der Werkzeugleiste anklicken. Das Programm arbeitet dann bei jedem Klick eine Zeile ab. Klicken Sie zehnmal und Sie werden sehen, das der Wert von 'x' im Debug Ausgabefenster angezeigt wird und dass er mit jedem Schleifendurchlauf um Eins erhöht wird. Wenn ein Programm auf diese Weise angehalten wurde, können Sie die Werte jeder/s beliebigen Variable, Array oder Liste mit der Variablenliste betrachten (Menü: Debugger → Variablenliste). Dieser kleine Überblick sollte genügen um die Grundlagen des Debuggers zu verstehen. Für fortgeschrittenere Informationen über weitere Leistungsmerkmale des Debuggers, schauen Sie in die PureBasic Hilfe.

Die 'OnError' Bibliothek

Die eingebaute 'OnError' Bibliothek bietet Ihnen die Möglichkeit, Fehler in Ihrem fertigen Executable abzufangen wenn der Debugger nicht verfügbar ist. In der Entwicklungsphase verwenden Sie den Debugger um Fehler in Ihrem Programm abzufangen. Wenn Sie aber Ihr fertiges Programm kompilieren, schalten Sie den Debugger normalerweise ab, um das kleinst- und schnellstmögliche Executable zu erstellen. Die Geschwindigkeit des Programms steigt mit abgeschaltetem Debugger etwa um den Faktor sechs. Mit der 'OnError' Bibliothek ist es einfach, fortgeschrittene Fehlerbehandlung in Ihr Programm zu integrieren, und Ihnen steht trotzdem die volle Geschwindigkeit von PureBasic zur Verfügung.

Es gibt eine Menge leistungsfähige Befehle in dieser Bibliothek, aber diese alle zu besprechen würde den Rahmen des Buches sprengen, deshalb werde ich mich auf die gängigsten und leicht verständlichen beschränken. Zuerst zeige ich Ihnen wie Sie einen Fehler effektiv abfangen und beschreiben, ohne dass Sie auf den Debugger zurückgreifen müssen. Schauen Sie auf folgendes Beispiel:

```
;Setzen der Fehlerbehandlungsroutine.
OnErrorGoto(?FehlerBehandlung)

;Auslösen eines klassischen 'Division durch Null' Fehlers.
Null.i         = 0
TestVariable.i = 100 / Null

;Behandele jegliche Systemfehler die im Programm auftreten.
FehlerBehandlung:
Text.s = "ID Nummer des Fehlers:" + #TAB$ + Str(ErrorCode()) + #CRLF$
Text.s + "Fehler Beschreibung:"   + #TAB$ + ErrorMessage()   + #CRLF$
Text.s + "Aufgetreten in Zeile:"  + #TAB$ + Str(ErrorLine()) + #CRLF$
Text.s + "Aufgetreten in Datei:"  + #TAB$ + ErrorFile()      + #CRLF$
MessageRequester("FEHLER", Text)
End
```

Hier habe ich den 'OnErrorGoto()' Befehl verwendet, um zu spezifizieren, wo im Fehlerfall hingesprungen werden soll. Der Parameter des Befehls ist ein Sprungmarke die angesprungen werden soll. Wenn Sie

etwas genauer auf die übergebene Sprungmarke schauen stellen Sie fest, dass ich vor den Namen der Sprungmarke ein Fragezeichen gesetzt habe. Dieses wird benötigt, da der 'OnErrorGoto()' Befehl einen Zeiger des Sprungziels anstelle des Namens benötigt. Ein wie hier vorangestelltes Fragezeichen gibt die Speicheradresse der Sprungmarke zurück. Ein Zeiger ist eine Variable, die eine Speicheradresse von einem irgendwo gespeicherten Objekt enthält, schauen Sie in Kapitel 13 (Zeiger) für eine komplette Erklärung. Wie beim 'Goto' und 'Gosub' Befehl lassen wir auch hier den Doppelpunkt hinter dem Sprungmarkennamen aus, wenn wir diesen an den 'OnErrorGoto()' Befehl übergeben.

Nachdem das Sprungziel mit dem 'OnErrorGoto()' Befehl definiert wurde, dann helfen uns andere Befehle aus der 'OnError' Bibliothek weiter, zu verstehen was und wo etwas falsch gelaufen ist. In obigem Beispiel habe ich folgende Befehle verwendet:

'ErrorCode()'
Dieser Befehl gibt die einmalige Nummer des zuletzt aufgetretenen Fehlers zurück.

'ErrorMessage()'
Dieser Befehl gibt einen String zurück, der eine Beschreibung des aufgetretenen Fehlers enthält.

'ErrorLine()'
Dieser Befehl gibt die Zeilennummer Ihres Quelltextes zurück, in der der Fehler aufgetreten ist, entweder im Hauptquelltext oder in einer Include Datei. Damit dieser Befehl richtig funktioniert müssen Sie die 'OnError-Unterstützung einschalten' Compiler Option anwählen, bevor Sie Ihr Programm kompilieren.

'ErrorFile()'
Dieser Befehl gibt einen String zurück, der Sie darüber informiert in welcher Quelltext Datei der Fehler aufgetreten ist. Das ist sehr nützlich, wenn Sie viele Include Dateien in Ihrem Projekt verwenden. Damit dieser Befehl richtig funktioniert, müssen Sie die 'OnError-Unterstützung einschalten' Compiler Option anwählen, bevor Sie Ihr Programm kompilieren.

Die letzten beiden hier aufgeführten Befehle erfordern das Einschalten der 'OnError-Unterstützung einschalten' Compiler Option, bevor Sie Ihr Programm kompilieren. Diese finden Sie in den Compiler Optionen innerhalb der IDE, (Menü: Compiler → Compiler-Optionen... → Compiler-Optionen → OnError-Unterstützung einschalten).

Um die Fehlerbehandlung mittels der 'OnError' Befehle zu demonstrieren, habe ich einen reproduzierbaren Fehler in das Programm eingebaut. Dieser Fehler ist ein einfacher 'Division durch Null' Fehler und wird zum Beispiel so programmiert:

```
;Auslösen eines klassischen 'Division durch Null' Fehlers.
Null.i         = 0
TestVariable.i = 100 / Null
```

Wenn der Debugger eingeschaltet ist, fängt er diesen Fehler einfach ab, ist er aber abgeschaltet, wird dieser Fehler nicht erkannt. Wir können die 'OnError' Bibliothek verwenden um diesen Fehler ohne die Hilfe des Debuggers abzufangen. Um dieses Beispiel zweckmäßig zu starten, müssen Sie den Debugger abschalten und die 'OnError-Unterstützung einschalten' Compiler Option aktivieren. Wenn Sie nun das Programm kompilieren, sollte es den 'Division durch Null' Fehler abfangen und Ihnen eine detaillierte Rückmeldung geben, alles ohne die Hilfe des Debuggers. Abb. 24 zeigt dieses Beispiel in Aktion und listet die gesammelten Informationen der anderen 'OnError' Befehle auf.

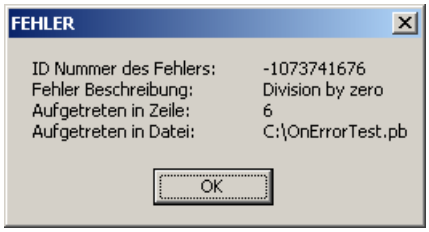


Abb. 24

Im letzten Beispiel habe ich den 'OnErrorGoto()' Befehl verwendet um eine Sprungmarke zu spezifizieren, die im Fehlerfall angesprungen wird. Aufgrund der Struktur dieses Befehles ist es aber nicht ratsam aus der Fehlerbehandlungsroutine auf die Daten in Ihrem Programm zuzugreifen, da die Adressen von z.B. Variablen unter Umständen ihre Gültigkeit verloren haben können. Die beste Vorgehensweise ist, einfach

Informationen über den Fehler zu erfassen, diese anzuzeigen und dann das Programm zu beenden. Möchten Sie dem Benutzer noch die Möglichkeit geben, zum Beispiel ungesicherte Daten zu speichern, sollten Sie für den Aufruf der Fehlerbehandlungsprozedur den 'OnErrorCall()' Befehl verwenden. Mit diesem ist es innerhalb der Fehlerbehandlungsprozedur möglich noch auf Daten im Programm zuzugreifen und diese gegebenenfalls zu sichern. Dies muss allerdings alles innerhalb der Fehlerbehandlungsprozedur erfolgen, da beim beenden der Prozedur auch das Programm endet. Hier ein Beispiel:

```
;Behandele jegliche Systemfehler die im Programm auftreten.
Procedure FehlerBehandlung()
  Text.s = "ID Nummer des Fehlers:" + #TAB$ + Str(ErrorCode()) + #CRLF$
  Text.s + "Fehler Beschreibung:"   + #TAB$ + ErrorMessage()   + #CRLF$
  Text.s + "Aufgetreten in Zeile:"  + #TAB$ + Str(ErrorLine()) + #CRLF$
  Text.s + "Aufgetreten in Datei:"  + #TAB$ + ErrorFile()      + #CRLF$ + #CRLF$
  Text.s + "Möchten Sie Ihre Daten noch sichern?"
  RueckgabeWert.i = MessageRequester("FEHLER", Text, #PB_MessageRequester_YesNo)
  If RueckgabeWert = #PB_MessageRequester_Yes
    Text.s = "Hier könnten noch Befehle zum sichern von Daten stehen." + #CRLF$
    Text.s + "Mit dem Ende dieser Prozedur endet auch das Programm."
    MessageRequester("Information", Text)
  EndIf
EndProcedure

;Setzen der Fehlerbehandlungsroutine.
OnErrorCall(@FehlerBehandlung())

;Auslösen eines klassischen 'Division durch Null' Fehlers.
Null.l         = 0
TestVariable.l = 100 / Null
End
```

Hier demonstriere ich, wie Sie nach einem Fehler noch Aufräumarbeiten in Ihrem Programm durchführen könnten. Hier kann der Benutzer im Bedarfsfall mit großer Wahrscheinlichkeit noch seine Daten sichern, bevor das Programm endet.

'OnErrorGoto' und 'OnErrorCall'

Diese beiden Befehle sind sich sehr ähnlich, indem Sie Ihnen die Möglichkeit bieten im Fehlerfall an eine Stelle im Programm zu springen, an der Sie die Fehler behandeln und beschreiben können. Der Hauptunterschied liegt in der Weiterarbeit mit dem Programm.

Der 'OnErrorGoto' Befehl gibt Ihnen die Möglichkeit eine Sprungmarke oder Fehlerbehandlungsprozedur zu definieren, die im Fehlerfall aufgerufen wird und den Fehler auswertet. Danach sollten Sie Ihr Programm verlassen, da ein Zugriff auf Daten aufgrund von evtl. ungültigen Variablenadressen nicht mehr ratsam ist.

Der 'OnErrorCall' Befehl gibt Ihnen die Möglichkeit eine Fehlerbehandlungsprozedur zu definieren, die im Fehlerfall aufgerufen wird und den Fehler auswertet. Innerhalb der Prozedur haben Sie noch die Möglichkeit Aufräumarbeiten in Ihrem Programm durchzuführen (z.B. Daten zu sichern). Mit dem Ende der Fehlerbehandlungsprozedur endet auch das Programm.

Sagen wir zum Beispiel, das Sie einen Texteditor geschrieben haben und Torsten verwendet ihn um einen Brief zu schreiben. Nach einer Stunde Schreibarbeit tritt ein unbegründeter Fehler auf und das Programm zeigt eine Fehlermeldung. Torsten möchte nicht die Arbeit der letzten Stunde an einen Fehler verlieren und damit auch seinen Brief. In diesem Fall kann Torsten darüber informiert werden, was im Programm passiert ist, ihm aber noch die Möglichkeit geben werden seinen Brief zu speichern bevor das Programm endet. In der Fehlermeldung könnte auch eine E-Mail Adresse stehen, so dass Torsten Sie kontaktieren kann um den Sachverhalt zu schildern, was Ihnen wiederum die Möglichkeit gibt diesen Fehler zu korrigieren.

Erzeugen und Auswerten von benutzerdefinierten Fehlern

Bis hierher habe ich Ihnen Wege gezeigt, Systemfehler die in Ihrem Programm auftreten abzufangen und zu dokumentieren. Auf diese Weise ist es möglich, jeden auftretenden Systemfehler abzufangen. Vielleicht wollen Sie manchmal auch eigene Fehler zum Zweck der Individualisierung erstellen. Wenn dies der Fall ist, können Sie den 'RaiseError()' Befehl verwenden um eigene Fehler zu erzeugen und diese zu melden.

Der 'RaiseError()' Befehl benötigt einen Parameter um richtig zu arbeiten. Dieser Parameter ist ein Integer Wert der den Fehler identifiziert. Ich verwende üblicherweise Konstanten um eigene Fehlernummern zu erstellen, auf die ich mich dann mit dem Namen beziehen kann.

```
#FEHLER_DATEILESEN     = 1
#FEHLER_DATEISCHREIBEN = 2
#DATEI_TEXT            = 1

;Setzen der Fehlerbehandlungsroutine.
OnErrorGoto(?FehlerBehandlung)

If ReadFile(#DATEI_TEXT, "Report.txt") = #False
  ;Wenn das Lesen aus der Datei fehlschlägt, dann melde einen Fehler
  RaiseError(#FEHLER_DATEILESEN)
EndIf
End

;Behandele jegliche Systemfehler die im Programm auftreten.
FehlerBehandlung:
Text.s + "ID Nummer des Fehlers:" + #TAB$ + Str(ErrorCode()) + #CRLF$
Select ErrorCode()
  Case #FEHLER_DATEILESEN
    Beschreibung.s = "Die Datei konnte nicht gelesen werden."
  Case #FEHLER_DATEISCHREIBEN
    Beschreibung.s = "Die Datei konnte nicht geschrieben werden."
EndSelect
Text.s + "Fehler Beschreibung:"   + #TAB$ + Beschreibung     + #CRLF$
Text.s + "Aufgetreten in Zeile:"  + #TAB$ + Str(ErrorLine()) + #CRLF$
Text.s + "Aufgetreten in Datei:"  + #TAB$ + ErrorFile()      + #CRLF$ + #CRLF$
MessageRequester("FEHLER", Text)
End
```

Wenn ich in obigem Beispiel die Datei 'Report.txt' nicht unter Verwendung des 'ReadFile()' Befehls lesen kann, erzeuge ich einen benutzerdefinierten Fehler mit dem 'RaiseError()' Befehl. Dieser Befehl wird durch die Konstante '#FEHLER_DATEILESEN' identifiziert, die den Wert '1' hat. Wenn der Fehler auf diese Art ausgelöst wurde, wird die Fehlerbehandlungsprozedur aufgerufen. In dieser Prozedur können Sie mit dem 'ErrorCode()' Befehl prüfen, welche Fehlernummer erzeugt wurde. Abhängig von dem Ergebnis dieses Befehls können Sie dann eine entsprechend zugeschnittene Fehlerbeschreibung nach Ihren Wünschen erzeugen. Hier verwende ich den 'ErrorCode()' Befehl mit einer 'Select' Anweisung, die verschiedene beschreibende Strings abhängig vom Fehlerwert zurückgibt. Die Meldung im Message Requester zeigt dann diese Beschreibung an.

Wie Sie in diesen Beispielen erkennen können, bietet die 'OnError' Bibliothek fortgeschrittene Möglichkeiten, Fehler zu prüfen, verwendet dabei aber eine einfache Syntax.

II. Grafische Benutzeroberflächen

In diesem Teil des Buches spreche ich über grafische Benutzeroberflächen und wie sie mit PureBasic erstellt werden. Nahezu alle modernen Betriebssysteme verfügen über eine eingebaute grafische Benutzeroberfläche, die dem Benutzer ein flüssiges arbeiten mit Programmen die diese nutzen erlaubt. Das Betriebssystem stellt den Anwendungen die grafische Benutzeroberfläche über ein 'Application Programming Interface' (API) zur Verfügung. Durch dieses kann eine Anwendung dem Betriebssystem mitteilen wie die Benutzeroberfläche des Programms gestaltet werden soll. Das hört sich ziemlich kompliziert an, aber es ist relativ einfach und elegant mit den PureBasic Bibliotheken 'Window', 'Menu' und 'Gadget' zu handhaben.

PureBasic erstellt die grafischen Oberflächen für Ihr Programm, indem es auf das 'Application Programming Interface' des Systems zugreift für das Sie ihr Programm erstellen. Mit anderen Worten, wenn Sie eine Oberfläche für ein Programm erstellen und dieses auf einem bestimmten System kompilieren, dann hat das Programm das richtige Aussehen für das Betriebssystem auf dem es erstellt wurde. Das ist sehr wichtig für die professionelle Anwendungsentwicklung.

Dieser Abschnitt beginnt mit der Erklärung und Demonstration von Programmen die eine Konsole als Benutzeroberfläche verwenden, was unbestreitbar die einfachste Form von Benutzeroberflächen ist. Später erkläre ich dann wie Sie Programme mit betriebssystemspezifischen Grafischen Benutzeroberflächen erstellen und wie Sie Menüs und Grafiken hinzufügen. Im letzten Abschnitt gebe ich Ihnen einen Überblick zum 'PureBasic Visual Form Designer'. Mit diesem Programm können Sie ihre Oberfläche visuell gestalten, gerade so als würden Sie ein Programm malen.

Nach der Lektüre dieses Buchabschnittes haben Sie verstanden wie Sie grafische Benutzeroberflächen für Ihre Programme erstellen und wie Sie Benutzereingaben verarbeiten müssen.

9. Erstellen von Benutzeroberflächen

In diesem Kapitel erkläre ich Ihnen wie Sie grafische Oberflächen für Ihre Programme erstellen. PureBasic macht diese Aufgabe sehr einfach, da es die komplexe Schnittstelle zur Anwendungsentwicklung in einfach zu erlernenden Befehlen abbildet. Ich werde Ihnen vollständig erklären, wie Sie grafische Oberflächen komplett mit Menüs und teilweise Grafiken erstellen. Ich werde außerdem besprechen wie Sie Ereignisse auf Ihrer Oberfläche behandeln müssen, wenn der Benutzer zum Beispiel eine Schaltfläche anklickt oder einen Menüpunkt auswählt. Ich hoffe, dass Sie nach dem Lesen dieses Kapitels gerüstet sind, Benutzeroberflächen für alle Programme die Sie entwickeln wollen, zu erstellen.

Konsolen Programme

Wir können nicht rennen bevor wir laufen gelernt haben, deshalb werde ich Ihnen eine Einführung in das geben, was als Konsolenanwendung bekannt ist. Konsolenprogramme sind, wie der Name schon sagt, Programme die eine Konsole als Benutzeroberfläche haben. Eine Konsole ist eine textbasierte Oberfläche, die Eingaben annimmt und Ausgaben mit einfachen Zeichen darstellen kann. Auf einigen Betriebssystemen kann man mit der Konsole einfache Grafiken anzeigen, indem man einen Teil des ASCII Zeichensatzes durch grafische Symbole ersetzt.

Konsolen Oberflächen werden üblicherweise in Programmen verwendet die keine vollwertige Benutzeroberfläche benötigen. Diese Art Programme sind häufig Kommandozeilen Werkzeuge, die von anderen Konsolen oder Dingen wie CGI Programmen, die im Hintergrund auf Webservern laufen, gestartet werden. Kurz, eine Konsole wird benutzt um einfache Informationen von Programmen auszugeben und einfache Texteingaben vom Benutzer entgegenzunehmen. Befehle, die eine Konsole erstellen und mit ihr arbeiten, sind in der 'Console' Bibliothek zusammengefasst (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Console). Diese Bibliothek bietet dem PureBasic Programmierer verschiedene Befehle um Text anzuzeigen, Benutzereingaben entgegen zu nehmen, die Konsole zu löschen und die Farben zu ändern. Hier ein Beispiel wie in PureBasic ein Konsolenprogramm erstellt wird:

```
If OpenConsole()
  Print("Das ist ein Konsolen Testprogramm, druecken Sie Enter zum beenden...")
  Input()
  CloseConsole()
EndIf
End
```

In diesem Beispiel habe ich den 'OpenConsole()' und den 'CloseConsole()' Befehl zum öffnen und schließen des aktuellen Konsolenfensters verwendet, was selbsterklärend ist. Der zweite Befehl ist der 'Print()' Befehl, der einen String als Parameter benötigt, der dann im Konsolenfenster angezeigt wird. Dieser Befehl ist weitestgehend mit dem 'PrintN()' Befehl aus der 'Console' Bibliothek identisch. Der 'PrintN()' Befehl gibt ebenfalls eine Textzeile aus, er fügt am Ende aber noch ein Zeilenende Zeichen ein und springt dann in die nächste Zeile nachdem er den Text geschrieben hat. Es besteht eine große Ähnlichkeit mit den Befehlen 'WriteString()' und 'WriteStringN()' die ich Ihnen in Kapitel 7 erklärt habe (Arbeiten mit Dateien).

Den letzten Befehl, den ich in obigem Beispiel verwendet habe, ist 'Input()'. Dieser Befehl hält das Programm an, bis die Enter Taste auf der Tastatur gedrückt wird. Dieser Befehl gibt alle Zeichen (als String) zurück, die in die Konsole eingegeben wurden, bevor abschließend die Enter Taste gedrückt wird. Da ich in meinem Programm keine Rückgabewerte verarbeite, wird der Befehl an dieser Stelle nur dazu verwendet die Konsole offen zu halten, damit der Benutzer den von mir ausgegebenen Text lesen kann. Wenn wir diesen Befehl weglassen, ist die Konsole genauso schnell wieder geschlossen wie sie geöffnet wurde. Die Verwendung des 'Input()' Befehls auf diese Art, ist eine einfache Möglichkeit die Konsole offen zu halten bis wir den ausgegebenen Text gelesen haben. Wir sollten aber den Benutzer darüber Informieren, dass er die Enter Taste drücken muss um das Programm fortzuführen.

Lesen von Benutzereingaben

Vielleicht möchten Sie manchmal die Eingaben des Benutzers lesen, was eine einfache Zahl oder ein Textstring sein kann. Obwohl es schön ist alle Benutzereingaben einzusammeln, dürfen Sie nicht vergessen das der 'Input()' Befehl nur Strings zurückgibt. Das nächste Beispiel demonstriert das und führt Sie gleich in ein paar neue Konsolenbefehle ein:

```
If OpenConsole()

  EnableGraphicalConsole(#True)

  Repeat

    ConsoleColor(10, 0)
    PrintN("EINMALEINS GENERATOR")
    PrintN("")
    ConsoleColor(7, 0)
    PrintN("Bitte geben Sie eine Zahl ein und druecken dann Enter...")
    PrintN("")

    Zahl.q = Val(Input())
    If Zahl = 0
      ClearConsole()
      Continue
    Else
      Break
    EndIf

  ForEver
  PrintN("")

  For x.i = 1 To 10
    PrintN(Str(x) + " x " + Str(Zahl) + " = " + Str(x * Zahl))
  Next x
  PrintN("")

  Print("Druecken Sie Enter zum beenden...")
  Input()
  CloseConsole()
EndIf
End
```

Dieses Beispiel sieht sehr kompliziert aus, aber das ist es nicht wirklich wenn Sie das Programm Zeile für Zeile durchlesen. Der erste neue Befehl den ich hier verwendet habe ist 'EnableGraphicalConsole()'. Dieser schaltet die grafischen Fähigkeiten der Konsole in Abhängigkeit des übergebenen Parameters ('#True' oder '#False') ein oder aus. Da wir später den 'ClearConsole()' Befehl verwenden, der nur in der grafischen Konsole funktioniert, setzen wir den Parameter für 'EnableGraphicalConsole()' auf True.

Unterschiede zwischen grafischem und nicht grafischem Konsolen Modus

Mit dem 'EnableGraphicalConsole()' Befehl können Sie zwischen dem Text Modus und dem grafischen Modus der aktuellen Konsole umschalten. Hier die Unterschiede der einzelnen Modi:

Text Modus (Standard):
ASCII Kontrollzeichen funktionieren korrekt (ASCII Bereich '0' bis '31').
Die Umleitung von Ausgabeströmen mittels 'Pipes' (Kommunikationskanäle) funktioniert korrekt (wichtig für CGI Programme).
Lange Strings werden in die nächste Zeile umgebrochen, wenn sie das Ende des Konsolenfensters erreichen.
Sie können Daten aus der Konsole lesen und in diese schreiben, die nicht notwendigerweise Text basiert sind.

Grafischer Modus (wechseln mit 'EnableGraphicalConsole(#True)'):
Textzeichen außerhalb des ASCII Bereichs '33' bis '126' werden als kleine, einfache Grafiken angezeigt.
Lange Strings werden abgeschnitten wenn sie das Ende des Konsolenfensters erreichen.
'ClearConsole()' löscht vollständig alle Ausgaben in der Konsole.
'ConsoleLocate()' platziert den Cursor im Konsolenfenster, was eine Textausgabe an jeder beliebigen Stelle in der Konsole ermöglicht.

Diese Liste enthält ein paar fortgeschrittene Punkte die Ihnen noch nicht bekannt sind, aber ich habe sie in diese Liste aufgenommen, damit Sie diese in Zukunft als eine Referenz verwenden können wenn Sie mit diesen Dingen etwas vertrauter sind.

Ich habe auch an einigen Stellen die Textfarbe mit dem 'ConsoleColor()' Befehl geändert. Der erste Parameter ist die Textfarbe und der zweite Parameter ist die Hintergrundfarbe. Die Parameter sind Zahlenwerte von '0' bis '15', die verschiedene Farben repräsentieren. Schauen Sie in die Hilfe um zu erfahren welche Farbe mit welcher Zahl verknüpft ist (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Console → ConsoleColor).

Weiterhin habe ich in diesem Beispiel eine Schleife verwendet um die Benutzereingaben zu verwalten und zu vergleichen. Diese wird als eine Möglichkeit verwendet das Programm immer wieder neu zu starten, wenn der Benutzer einen ungültigen Wert eingibt. Ich möchte, dass der Benutzer eine Zahl eingibt, deshalb verwende ich den 'Val()' Befehl um den vom 'Input()' Befehl zurückgegebenen String in einen Quadwert zu konvertieren. Dieser zurückgegebene Wert wird dann geprüft ob er gleich '0' ist. In diesem Fall hat der Benutzer einen nicht numerischen String oder die Zahl '0' eingegeben. Wenn dies der Fall ist entferne ich mit dem 'ClearConsole()' Befehl alle Ausgaben auf der Konsole und zwinge die Schleife zur Fortsetzung am Anfang, wo ich den Original Text neu ausgebe und den Benutzer zu einer Eingabe auffordere. Wenn die Benutzereingabe ein Zahl größer '0' ist, breche ich die Schleife ab und fahre mit dem restlichen Programm fort, welches die Einmaleins Tabelle für diese Zahl ausgibt.

Lesen von Benutzereingaben in Echtzeit

Das letzte Beispiel hat auf schöne Weise die Verwendung des 'Input()' Befehls demonstriert, der sehr hilfreich ist wenn Sie Strings oder ähnliches eingeben wollen. Allerdings hat dieser Befehl die Einschränkung, dass zur Übergabe der Daten an das Programm die Enter Taste gedrückt werden muss. Was ist wenn Sie eine Benutzereingabe in Echtzeit erfassen möchten, um sofort festzustellen wenn eine Taste gedrückt wurde, um zum Beispiel eine Aktion zu starten. Dieses Verhalten kann mit den Befehlen 'Inkey()' und 'RawKey()' erzielt werden. Schauen Sie auf das nächste Beispiel in dem beide Befehle verwendet werden:

```
Procedure ZeigeTitel()
  ConsoleColor(10, 0)
  PrintN("TASTEN CODE FINDER")
  PrintN("")
  ConsoleColor(7, 0)
EndProcedure

Procedure ZeigeBeendenText()
  PrintN("")
  ConsoleColor(8, 0)
  PrintN("Druecken Sie eine andere Taste oder druecken Sie Escape zum beenden")
```

```
  ConsoleColor(7, 0)
EndProcedure

If OpenConsole()

  EnableGraphicalConsole(#True)
  ZeigeTitel()
  PrintN("Druecken Sie eine Taste...")
  Repeat
    TastenDruck.s = Inkey()
    TastenCode.l  = RawKey()

    If TastenDruck <> ""

      ClearConsole()
      ZeigeTitel()
      PrintN("Gedrueckte Taste: " + TastenDruck)
      PrintN("Tasten Code     : " + Str(TastenCode))
      ZeigeBeendenText()

    ElseIf TastenCode

      ClearConsole()
      ZeigeTitel()
      PrintN("Gedrueckte Taste: " + "kein-ASCII")
      PrintN("Tasten Code     : " + Str(TastenCode))
      ZeigeBeendenText()

    Else

      Delay(1)

    EndIf

  Until TastenDruck = #ESC$
  CloseConsole()
EndIf
End
```

Dieses Beispiel ähnelt dem vorherigen dahin gehend, dass wir eine Konsole öffnen, diese in den grafischen Modus umschalten und einigen Text auf ihr anzeigen. Der Hauptunterschied besteht darin, dass die 'Repeat' Schleife kontinuierlich zwei Befehle aufruft, die ermitteln ob und wenn ja, welche Taste gedrückt wurde. Diese beiden Befehle sind 'Inkey()' und 'RawKey()'.

Der erste Befehl, 'Inkey()' gibt einen Ein-Zeichen String der zum Zeitpunkt des Aufrufs gedrückten Taste zurück. Wenn ich die Taste 'D' auf meiner Tastatur drücke, während dieser Befehl aufgerufen wird, dann gibt 'Inkey()' den String 'd' zurück. Auch wenn dieser Befehl einfach zu verstehen ist, müssen Sie trotzdem darauf achten, dass die Tasten auf einer Standard Tastatur üblicherweise mit Großbuchstaben bedruckt sind. Wenn eine nicht ASCII Taste gedrückt wird während der Befehl aufgerufen wird, dann gibt der 'Inkey()' Befehl einen leeren String zurück, bis eine ASCII Taste gedrückt wird.

Der zweite Befehl zum sammeln von Informationen welche Taste gedrückt wurde ist ein Begleiter des 'Inkey()' Befehls mit Namen 'RawKey()'. Wie der Name des Befehls schon nahelegt, gibt dieser Befehl den rohen numerischen Tastencode der zum Zeitpunkt des Befehlsaufrufs gedrückten Taste zurück. In obigem Beispiel können Sie sehen, wie beide Befehle während des Schleifendurchlaufs Informationen über die gedrückte Taste sammeln. Das ist eine gute Möglichkeit Benutzereingaben in Echtzeit zu erfassen und beim Druck einer bestimmten Taste dementsprechend zu reagieren.

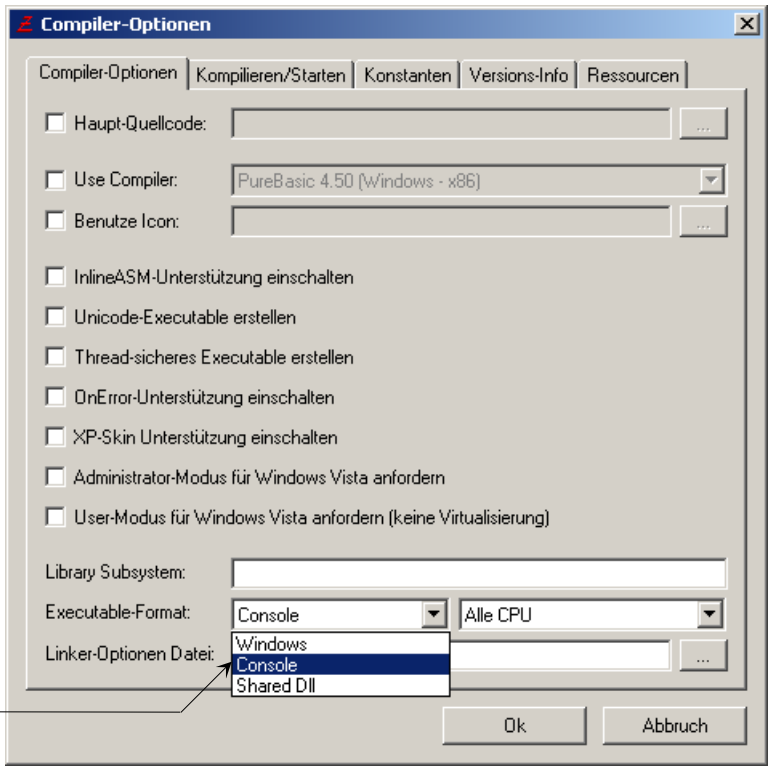
Verzögern von Aktivitäten

Sie haben wahrscheinlich bemerkt, das ich in obigem Beispiel auch einen 'Delay()' Befehl verwendet habe, um das Programm für '1' Millisekunde zu pausieren. Auch wenn dies etwas fremdartig und unnötig aussieht, hat der 'Delay()' Befehl an dieser Stelle einen bestimmten Grund. Da Sie Ihre Programme in einem Multitasking Betriebssystem verwenden, ist es guter Programmierstil nicht zu viel CPU Zeit zu beanspruchen, wenn dies nicht erforderlich ist. Mit der Verwendung des 'Delay()' Befehls geben Sie anderen Programmen die Möglichkeit CPU Zeit zu nutzen während Ihr Programm pausiert. Schon eine Verzögerung von '1' Millisekunde ermöglicht es dem Betriebssystem die CPU lange genug freizugeben, dass andere Programme die Möglichkeit haben diese zu nutzen. Dieser Befehl ist besonders nützlich wenn Sie Schleifen

verwenden um die gleiche Aktion immer wieder auszuführen. Wenn Sie den 'Delay()' Befehl in einer solchen Schleife nicht verwenden, wird Ihr Programm alle CPU Ressourcen belegen bis das Programm beendet wird, weshalb andere Programme keine Rechenzeit mehr abbekommen. Das kann dazu führen, dass die anderen Programme 'einfrieren' und nicht mehr wie gewohnt reagieren.

Kompilieren eines echten Konsolen Programms

Wenn Sie die Konsolen Beispiele in der IDE starten, dann sehen die erstellten Programme wie richtige Konsolenprogramme aus und verhalten sich auch so, aber es sind keine Konsolenprogramme im eigentlichen Sinn, bis Sie sie im 'Console' Format kompiliert haben.



Der 'Compiler-Optionen' Dialog, wie er unter Microsoft Windows erscheint Abb. 25

Abb. 25 zeigt wo Sie die Option zum einstellen des Executable Format in der 'Compiler-Optionen' Dialogbox finden (Menü: Compiler → Compiler-Optionen...). Wenn Sie die oben gezeigte Option angewählt haben, klicken Sie auf 'OK' und erstellen Ihr fertiges Programm mit dem 'Executable erstellen..' (Menü: Compiler → Executable erstellen...) Befehl. Der PureBasic Compiler wird nun ein reines Konsolenprogramm erstellen.

Der Hauptunterschied zwischen den Executable Formaten besteht darin, dass die IDE standardmäßig das betriebssystemeigene Executable Format verwendet. Diese Programme, ob per Doppelklick- oder aus der Konsole gestartet, öffnen immer ein neues Konsolenfenster bevor sie starten. Wenn Sie das 'Console' Executable Format verwenden, können Sie explizit festlegen, dass dieses Programm als reines Konsolenprogramm kompiliert wird. Das bedeutet, dass das Programm ganz normal startet wenn Sie es per Doppelklick ausführen, aber wenn Sie es von der Kommandozeile starten wird kein neues Konsolenfenster geöffnet, sondern es arbeitet korrekt als ein Programm innerhalb der geöffneten Kommandozeile.

Und das ist alles, was zum Programmieren von Kommandozeilen Oberflächen nötig ist. Sie bieten sich nicht für die Verwendung in komplizierten Oberflächen an (obwohl in der Vergangenheit sogar Spiele erstellt wurden, die den grafischen Konsolen Modus verwendet haben). Konsolen werden heutzutage nur noch für die einfache Ausgabe von Kommandozeilen Programmen verwendet.

Erstellung von Betriebssystem üblichen Benutzeroberflächen

In diesem Abschnitt werde ich Ihnen zeigen, wie Sie Benutzeroberflächen erstellen, die aussehen und sich bedienen lassen wie die anderen Programme die Sie auf Ihrem Betriebssystem verwenden. Diese werden 'Betriebssystem übliche' Benutzeroberflächen genannt, da PureBasic das darunterliegende 'API' des Betriebssystems verwendet um die einzelnen Elemente der Oberfläche zu zeichnen. Schauen Sie als Beispiel für eine solche Oberfläche auf die PureBasic IDE. Diese wurde von Grund auf komplett in PureBasic geschrieben, welches seinerseits auf das API des Betriebssystems zurückgreift.

All diese Erwähnungen des Betriebssystem API entmutigen Sie vielleicht und Sie denken das klingt alles sehr kompliziert, aber keine Panik. Die gute Nachricht ist, dass Sie überhaupt nicht wissen müssen was 'unter der Haube' passiert, da PureBasic alles automatisch verwaltet und alles komplizierte von Ihnen fernhält. Schon mit den in PureBasic eingebauten Bibliotheken können Sie mächtige Anwendungen Programmieren, die mit den Anwendungen anderer Programmiersprachen konkurrieren können. Im Durchschnitt sind kompilierte PureBasic Programme kleiner und schneller als vergleichbare Programme, die mit Industrie Standard Sprachen erstellt wurden. Dies wurde viele Male durch Wettbewerbe im PureBasic Online Forum bestätigt.

Nun lassen Sie uns weitergehen und unsere erste Betriebssystem übliche Benutzeroberfläche erstellen.

Hallo Welt

Wie in der Programmierwelt üblich, ist das erste Beispiel für eine Oberfläche in einer Programmiersprache ein 'Hallo Welt' Programm. Öffnen Sie eine neue Datei in der IDE und geben folgendes ein:

```
#FENSTER_HAUPT = 1
#FLAGS         = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Hallo Welt", #FLAGS)
  Repeat
    Ereignis.i = WaitWindowEvent()
  Until Ereignis = #PB_Event_CloseWindow
EndIf
End
```

Wenn Sie diese Beispiel starten, werden Sie ein Fenster sehen, das ähnlich wie Abb. 26 aussieht. Um dieses Fenster zu schließen klicken Sie den 'Schließen' Knopf oben rechts an. Auch wenn das Beispiel nur aus einem blanken Fenster und einem Schließknopf in der Titelleiste besteht, ist das die Basis für alle Benutzeroberflächen in PureBasic.

(Microsoft Windows Beispiel) Abb. 26

In obigem Beispiel habe ich den 'OpenWindow()' Befehl verwendet um das eigentliche Fenster zu öffnen und um zu definieren, welche Eigenschaften es haben soll. Wenn Sie die Hilfe öffnen, den 'OpenWindow()' Befehl nachschlagen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Window → OpenWindow) und sich das Syntax Beispiel anschauen, können Sie sehen dass ich die Flags '#PB_Window_SystemMenu' und '#PB_Window_ScreenCentered' verwendet habe. Mit diesen habe ich definiert, dass das Fenster ein komplettes Funktionsmenü mit Schließknopf erhält und beim öffnen zentriert auf dem Bildschirm dargestellt wird. Wenn Sie wie hier mehrere Flags verwenden, müssen Sie diese mit einem Bitweise 'OR' Operator verbinden, wie obiger Code zeigt. Dieser Operator kombiniert die Werte aller verwendeten Flags auf binärer Ebene (für eine ausführliche Erklärung zu diesem Operator schauen Sie in Kapitel 3).

Nach dem 'OpenWindow()' Befehl habe ich eine 'Repeat' Schleife verwendet, die als Hauptschleife dient und alle eventuell im Fenster auftretenden Ereignisse verarbeitet.

Warum sind die Positionsparameter des Fensters auf '0' gesetzt?

Wenn Sie ein Fenster mit dem 'OpenWindow()' Befehl öffnen, geben der zweite und der dritte Parameter die Koordinaten auf dem Bildschirm an, wo das neue Fenster gezeichnet werden soll. Jedes Mal wenn Sie die Flags '#PB_Window_ScreenCentered' oder '#PB_Window_WindowCentered' im gleichen Befehl verwenden, werden diese Parameter ignoriert. Das ist so, weil PureBasic die Positionsparameter übergeht, um das Fenster auf dem Bildschirm oder auf einem anderen Fenster zu zentrieren. Wen dies der Fall ist, können Sie bedenkenlos '0' für beide Positionsparameter verwenden oder Sie benutzen die Konstante '#PB_Ignore' um explizit in Ihrem Quelltext kenntlich zu machen, das diese beiden Parameter Ignoriert werden weil Sie eines der beiden Flags übergeben haben.

Die Hauptschleife

Die Hauptschleife ist das, was das Programm 'am leben' erhält. Sie erlaubt dem Programm kontinuierlich zu laufen, während die Oberfläche neu gezeichnet wird (wenn das Fenster auf dem Desktop bewegt wird) und behandelt alle Ereignisse die im Fenster auftreten. Dankenswerter Weise behandelt PureBasic alle Neuzeichnen-Operationen automatisch, so dass wir uns nur um die Hauptschleife kümmern müssen, um das Programm am laufen zu halten und andere Ereignisse zu verarbeiten.

In meinem kleinen 'Hallo Welt Programm habe ich eine 'Repeat' Schleife als Hauptschleife verwendet, was mir die Möglichkeit gibt diese Schleife endlos laufen zu lassen bis eine bestimmtes Ereignis auftritt, in diesem Fall bis zum Auftreten eines Ereignisses vom Typ '#PB_Event_CloseWindow'. Dieses Ereignis wird durch klicken auf den Schließknopf erzeugt. Wenn dieses Ereignis empfangen wurde, wird die Schleife unterbrochen und das Programm endet.

Verstehen von Ereignissen

Jedes Programm, das Sie in einer beliebigen Programmiersprache erstellen, geht annähernd gleich beim behandeln von Ereignissen vor. Das schließt das Erkennen eines Ereignisses in der Hauptschleife und anschließender Ausführung eines bestimmten Code Abschnitts (in Abhängigkeit, welches Ereignis auftrat) mit ein. Welche Art von Ereignissen gibt es? Ein Ereignis kann durch viele Aktionen in Ihrem Programm ausgelöst werden. Beispiele für ausgelöste Ereignisse sind das klicken auf ein Gadget, das drücken einer Taste auf der Tastatur, die Auswahl eines Menü Elements, das bewegen des Programmfensters oder das schließen des Programms. Der Trick ist zu Wissen welches Ereignis ausgelöst wurde, damit Sie zweckmäßig darauf reagieren können. Um dies zu erfahren verwenden wir den 'WaitWindowEvent()' oder den 'WindowEvent()' Befehl.

Was ist ein Gadget?

In PureBasic ist ein Gadget ein Objekt auf einer grafischen Benutzeroberfläche, das eine gewisse Art von Interaktivität für Ihr Programm zur Verfügung stellt. In Microsoft Windows werden diese Objekte 'Controls' (Steuerelemente) genannt und in einigen Linux Distributionen werden sie 'Widgets' genannt. Gadgets sind Schaltflächen, Checkboxen, Schieber, Eingabefelder, Reiter, Rahmen, Fortschrittsbalken, usw. All diese unterschiedlichen interaktiven Komponenten nennt PureBasic Gadgets.

Diese beiden Befehle sind weitestgehend identisch, da Sie beide beim auftreten eines Ereignisses einen Ereignis Identifikator in Form eines Wertes vom Typ Integer zurückgeben. Der Unterschied zwischen beiden Befehlen besteht darin, dass der 'WaitWindowEvent()' Befehl Ihr Programm anhält bis ein Ereignis auftritt, dann den Ereignis Identifikator zurückgibt und das Programm normal ausführt bis er wieder aufgerufen wird. (Das ist sehr praktisch für die Verwendung in grafischen Benutzeroberflächen, da es dem Programm ermöglicht so wenig CPU Zeit wie möglich zu beanspruchen. Diese wird nur verwendet wenn ein Ereignis auftritt.) Der 'WindowEvent()' Befehl hingegen hält das Programm nicht an. Er prüft nur ob ein Ereignis aufgetreten ist das er zurückmelden kann. Wenn dies der Fall ist, gibt der Befehl den Ereignis Identifikator zurück. Dieser Befehl wird sehr selten in Benutzeroberflächen benutzt, da er diese ziemlich CPU 'hungrig' machen kann, diese also viel mehr CPU Leistung beanspruchen als nötig ist. Der 'WindowEvent()' Befehl wird oft verwendet wenn Sie Ihre Hauptschleife um jeden Preis am laufen halten müssen, wenn Sie zum Beispiel eine dynamische Grafik in Ihrem Programm anzeigen, die kontinuierlich neugezeichnet werden

muss, usw. Wenn dies der Fall ist, müssen Sie den 'WindowEvent()' Befehl verwenden. Der gute Programmierstil gibt Ihnen aber vor, dass Sie den 'Delay(1)' Trick anwenden sollten (wurde im Abschnitt Konsolen Programme erklärt) um zu verhindern, dass das Programm zu viele CPU Ressourcen verbraucht. Wenn kein Ereignis auftritt verwenden Sie 'Delay(1)' damit auch andere Programme die Chance haben die CPU zu nutzen.

Im 'Hallo Welt' Programm habe ich einen 'WaitWindowEvent()' Befehl innerhalb der 'Repeat' Schleife verwendet, um das Programm anzuhalten bis ein Ereignis auftritt. Wenn ein Ereignis auftrat gibt der 'WaitWindowEvent()' Befehl den Ereignis Identifikator zurück, der in einer Integer Variable mit dem Namen 'Ereignis' gespeichert wird. Die Schleife ist dann so programmiert, dass sie endet wenn 'Ereignis' den Wert '#PB_Event_CloseWindow' hat. Das ist eine eingebaute Konstante, die den Identifikator für ein 'Close Window' Ereignis von der Oberfläche enthält. Wenn beide gleich sind muss ein 'Close Window' Ereignis aufgetreten sein und die Schleife endet, und damit auch das Programm.

Ereignisse passieren die ganze Zeit

Auch wenn in diesem kleinen Beispiel keine anderen Ereignisse behandelt wurden, bedeutet das nicht, das diese nicht aufgetreten sind. Wenn wir das selbe 'Hallo Welt' Programm verwenden und fügen diesem eine Zeile mit einem 'Debug' Befehl hinzu, wird es die Identifikatoren aller Ereignisse während der Laufzeit im Debug Ausgabefenster anzeigen.

Starten Sie den folgenden Code, dann verschieben Sie das Fenster und klicken an verschiedenen Stellen hinein. Sie werden sehen, dass viele Ereignisse auftreten.

```
#FENSTER_HAUPT = 1
#FLAGS         = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Hallo Welt", #FLAGS)
  Repeat
    Ereignis.i = WaitWindowEvent()
    Debug Ereignis
  Until Ereignis = #PB_Event_CloseWindow
EndIf
End
```

Das sind die ganzen Identifikatoren die vom 'WaitWindowEvent()' Befehl für jedes Ereignis zurückgegeben wurden, auch wenn sie in Ihrem Programm nicht behandelt wurden. Das ist vollkommen normal und wird Ihr Programm nicht im geringsten verlangsamen.

Hinzufügen von 'Gadgets'

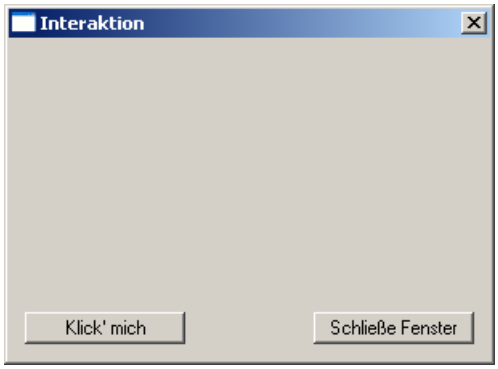
Bis jetzt haben wir nur ein einfaches Fenster erstellt das den Text 'Hallo Welt' in der Titelleiste anzeigt. Auch wenn das eine schöne Einführung in Benutzeroberflächen ist, so weist dieses Beispiel, abgesehen vom Schließknopf, keinerlei Interaktion mit dem Benutzer auf. Im nächsten Abschnitt werde ich Ihnen zeigen, wie Sie Gadgets in Ihr Programmfenster einbauen und damit die Funktionalität und die Interaktivität erhöhen.

Im nächsten Beispiel habe ich zwei Schaltflächen hinzugefügt und zeige Ihnen, wie Sie die Ereignisse von diesen erfassen.

```
Enumeration
  #FENSTER_HAUPT
  #KNOPF_INTERAKTION
  #KNOPF_SCHLIESSEN
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Interaktion", #FLAGS)

  ButtonGadget(#KNOPF_INTERAKTION, 10, 170, 100, 20, "Klick' mich")
  ButtonGadget(#KNOPF_SCHLIESSEN, 190, 170, 100, 20, "Schließe Fenster")
  Repeat

    Ereignis.i = WaitWindowEvent()
    Select Ereignis
      Case #PB_Event_Gadget
        Select EventGadget()
```

```
          Case #KNOPF_INTERAKTION
            Debug "Die Schaltfläche wurde gedrückt."
          Case #KNOPF_SCHLIESSEN
            Ende = #True
        EndSelect
    EndSelect

  Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

EndIf
End
```

Wenn Sie diesen Code überblicken, sollte er für Sie gut verständlich sein, deshalb werde ich mich auf einige Schlüsselfunktionen konzentrieren, die neu für Sie sind.



(Microsoft Windows Beispiel) Abb. 27

In diesem Beispiel habe ich mit dem 'ButtonGadget()' Befehl zwei Schaltflächen in das Fenster eingefügt. Wenn Sie die Hilfe öffnen und zur 'ButtonGadget()' Seite springen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Gadget → ButtonGadget), sehen Sie im Syntax Beispiel die erforderlichen Parameter für diesen Befehl.

Mit den Positionsparametern des 'ButtonGadget()' Befehls habe ich die Schaltflächen am unteren Rand des Fensters platziert.

Ein weiteres Schlüsselelement ist die 'Select' Anweisung in der Hauptschleife. Mit dieser prüfe ich die 'Ereignis' Variable, die den von 'WaitWindowEvent()' zurückgegebenen Wert enthält. Bei der Prüfung gehe ich folgendermaßen vor:

```
...
Select Ereignis
  Case #PB_Event_Gadget
    ...
EndSelect
...
```

Sie werden feststellen, das die erste 'Case' Anweisung auf einen Wert '#PB_Event_Gadget' überprüft. Das ist ein spezieller globaler Ereignis Wert, der zum prüfen verwendet wird, ob das Ereignis von einem Gadget ausgelöst wurde. Mit anderen Worten, wenn 'Ereignis' gleich '#PB_Event_Gadget', dann enthält 'Ereignis' einen Ereignis Identifikator von einem Gadget.

Wenn wir wissen, dass das Ereignis von einem Gadget stammt, müssen wir noch unterscheiden von welchem Gadget das Ereignis ausgelöst wurde. Dies ermitteln wir mit dem 'EventGadget()' Befehl, welcher die PB Nummer des Ursprungs Gadget zurückgibt. Den Rückgabewert von 'EventGadget()' prüfe ich in diesem Beispiel mit einer weiteren 'Select' Anweisung.

```
...
Select EventGadget()
  Case #KNOPF_INTERAKTION
    Debug "Die Schaltfläche wurde gedrückt."
  Case #KNOPF_SCHLIESSEN
    Ende = #True
EndSelect
...
```

Hier habe ich zwei 'Case' Anweisungen, eine zum Prüfen von '#KNOPF_INTERAKTION' und eine zum prüfen von '#KNOPF_SCHLIESSEN'. Wenn der Rückgabewert von 'EventGadget()' gleich '#KNOPF_INTERAKTION' ist, dann wurde die 'Klick' mich' Schaltfläche gedrückt. Das wissen wir, weil der 'EventGadget()' Befehl die PB Nummer dieses Gadgets zurückgegeben hat. Dann habe ich in der Zeile nach der 'Case' Anweisung etwas Code eingefügt, der beim Auftreten dieses Ereignisses abgearbeitet werden soll.

Die selbe Logik steckt hinter der nächsten 'Case' Anweisung. Wenn der Rückgabewert gleich '#KNOPF_SCHLIESSEN' ist, dann wurde die 'Schließe Fenster' Schaltfläche gedrückt. Wenn dies der Fall ist, weise ich der 'Ende' Variable den Wert True zu. Damit erreiche ich, dass die Hauptschleife endet, und damit auch das Programm.

Hier sehen sie auch, warum 'Enumeration' Blöcke so unglaublich hilfreich sind. Nach der Definition von numerischen Konstanten für alle Gadgets zu Beginn des Quelltextes, kann ich dann auf jeden einzelnen Wert über seinen Namen anstelle einer Zahl zugreifen. Das macht den Quelltext auf jeden Fall klarer und lesbarer.

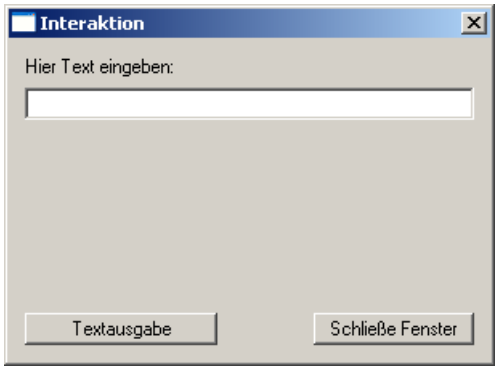
Entgegennehmen von Texteingaben

Lassen Sie uns das letzte Beispiel ein wenig erweitern und ein String Gadget hinzufügen, so dass wir ein wenig Text eingeben können den das Programm verwenden kann. String Gadgets sind sehr praktisch, da sie Ihrem Programm auf einfache Weise ermöglichen Texteingaben entgegenzunehmen. Das nächste Beispiel zeigt, wie Sie den Wert eines String Gadgets entgegennehmen und in Ihrem Programm verwenden. Lassen Sie uns Eines erstellen, das die volle Fensterbreite hat. Abb. 28 zeigt wie das kompilierte Programm aussieht, und hier ist der Quelltext:

```
Enumeration
  #FENSTER_HAUPT
  #TEXT_EINGABE
  #STRING_EINGABE
  #KNOPF_INTERAKTION
  #KNOPF_SCHLIESSEN
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Interaktion", #FLAGS)

  TextGadget(#TEXT_EINGABE, 10, 10, 280, 20, "Hier Text eingeben:")
  StringGadget(#STRING_EINGABE, 10, 30, 280, 20, "")
  ButtonGadget(#KNOPF_INTERAKTION, 10, 170, 120, 20, "Textausgabe")
  ButtonGadget(#KNOPF_SCHLIESSEN, 190, 170, 100, 20, "Schließe Fenster")
  SetActiveGadget(#STRING_EINGABE)
  Repeat

    Ereignis.i = WaitWindowEvent()
    Select Ereignis
      Case #PB_Event_Gadget
        Select EventGadget()
          Case #KNOPF_INTERAKTION
            Debug GetGadgetText(#STRING_EINGABE)
          Case #KNOPF_SCHLIESSEN
            Ende = #True
        EndSelect
    EndSelect

  Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

EndIf
End
```

In dieser aktualisierten Version habe ich dem 'Enumeration' Block noch ein paar Konstanten hinzugefügt, damit ich auf einfache Weise noch ein paar Gadgets hinzufügen kann. Das erste neu hinzugefügte Gadget ist ein Text Gadget, da einen nicht editierbaren Text auf dem Fenster anzeigt, in diesem Beispiel habe ich es zur Anzeige des Textes 'Hier Text eingeben:' verwendet. Das zweite neue Gadget ist ein String Gadget, das ein Eingabefeld auf dem Fenster erstellt, in welches Sie ihren eigenen Text eingeben können.

Wenn Sie die PureBasic Hilfe öffnen und zur 'StringGadget()' Seite wechseln (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Gadget → StringGadget), können Sie die Parameter sehen, die dieses Gadget benötigt. Ich habe die Parameter für Position und Größe so gewählt, dass das String Gadget am oberen Fensterrand angezeigt wird, unterhalb des Text Gadgets. Als 'Inhalt$' Parameter habe ich einen leeren String übergeben, der durch zwei direkt aufeinander folgende Anführungszeichen dargestellt wird. Es ist einfach ein String der keinen Inhalt hat. Ich habe einen Leerstring gewählt, da der 'Inhalt$' Parameter den Startstring des String Gadgets repräsentiert und ich wollte das Programm mit einem leeren Gadget starten.



(Microsoft Windows Beispiel) Abb. 28

Wenn dieses Programm läuft werden Sie feststellen das der Text Cursor im String Gadget blinkt. Das ist kein Standardverhalten, normalerweise müssen Sie auf das Gadget klicken um ihm den Fokus für den Cursor zu geben, damit dieser dort erscheint. In meinem Programm habe ich beim Start dem Gadget mit dem Befehl 'SetActiveGadget()' den Fokus zugewiesen. Dieser Befehl benötigt die PB Nummer des zu aktivierenden Gadgets als Parameter. Ich habe das in diesem Beispiel getan, um dem Benutzer zu helfen und ihm einen Mausklick zu ersparen.

Um die Interaktivität des Programms zu testen, geben Sie einen Text in das String Gadget ein und drücken dann die 'Textausgabe' Schaltfläche. Sie werden dann sehen, das der Text im Debug Ausgabefenster angezeigt wird. Das ist so, weil ich die '#KNOPF_INTERAKTION' 'Case' Anweisung verändert habe. Statt einen fertigen Text auszugeben, lese ich nun den Inhalt des String Gadgets und gebe den gelesenen Wert im Debug Ausgabefenster aus.

Der 'GetGadgetText()' Befehl benötigt als Parameter die PB Nummer des Gadgets, aus dem Sie den Text lesen wollen. Hier habe ich das String Gadget ausgewählt. Dieser Befehl kann mit vielen PureBasic Gadgets verwendet werden, aber in Verbindung mit einem String Gadget gibt er einfach den Text als String zurück, den ich dann unmittelbar mit dem 'Debug' Befehl ausgebe.

Anzeigen von Text

Lassen Sie uns das Programm noch ein wenig erweitern und den ausgelesenen Text nicht im Debug Ausgabefenster, sondern in einem Gadget direkt in unserem Programm anzeigen. Im nächsten Beispiel habe ich ein ListView-Gadget hinzugefügt, um die eingegebenen Texte in einer Liste nicht editierbarer Strings anzuzeigen.

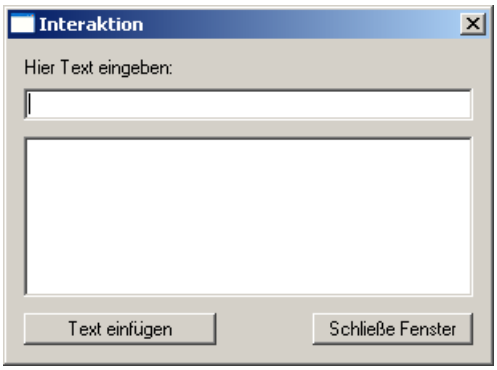
Ein ListView-Gadget ist eine gute Möglichkeit eine Liste von Strings anzuzeigen, da jeder String seine eigene Zeile in dem Gadget hat.

PureBasic stellt eine Menge Befehle zum arbeiten mit ListView-Gadgets bereit. Diese reichen vom hinzufügen und löschen bis zum zählen und sortieren der enthaltenen Elemente. Im folgenden kleinen Beispiel habe ich nur den 'AddGadgetItem()' Befehl verwendet, um neue Strings hinzuzufügen. Abb. 29 zeigt das neue Programm im Betrieb, hier ist der Code:

```
Enumeration
  #FENSTER_HAUPT
  #TEXT_EINGABE
  #STRING_EINGABE
  #LISTE_EINGABE
  #KNOPF_INTERAKTION
  #KNOPF_SCHLIESSEN
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
```

```
If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Interaktion", #FLAGS)

  TextGadget(#TEXT_EINGABE, 10, 10, 280, 20, "Hier Text eingeben:")
  StringGadget(#STRING_EINGABE, 10, 30, 280, 20, "")
  ListViewGadget(#LISTE_EINGABE, 10, 60, 280, 100)
  ButtonGadget(#KNOPF_INTERAKTION, 10, 170, 120, 20, "Text einfügen")
  ButtonGadget(#KNOPF_SCHLIESSEN, 190, 170, 100, 20, "Schließe Fenster")
  SetActiveGadget(#STRING_EINGABE)
  Repeat

    Ereignis.i = WaitWindowEvent()
    Select Ereignis
      Case #PB_Event_Gadget
        Select EventGadget()
          Case #KNOPF_INTERAKTION
            AddGadgetItem(#LISTE_EINGABE, -1, GetGadgetText(#STRING_EINGABE))
            SetGadgetText(#STRING_EINGABE, "")
            SetActiveGadget(#STRING_EINGABE)
          Case #KNOPF_SCHLIESSEN
            Ende = #True
        EndSelect
    EndSelect

  Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

EndIf
End
```

Dieses Beispiel ist dem letzten sehr ähnlich, tatsächlich ist der einzige Unterschied das hinzugefügte ListView-Gadget. Wenn Sie in der PureBasic Hilfe unter 'ListViewGadget()' nachschauen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Gadget → ListViewGadget), erhalten Sie einen guten Überblick über dieses Gadget und was Sie damit alles bewerkstelligen können. Das Syntax Beispiel zeigt Ihnen, welche Parameter benötigt werden. Ich habe Position und Größe so definiert, dass das ListView-Gadget in der Mitte des Hauptfensters dargestellt wird, über den Schaltflächen und unter dem String Gadget.



(Microsoft Windows Beispiel) Abb. 29

Wenn Sie dieses Programm gestartet haben, geben Sie einen Text in das String Gadget ein und drücken die 'Text einfügen' Schaltfläche. Dadurch wird der Text aus dem String Gadget ausgelesen und dem ListView-Gadget hinzugefügt. Wenn das erledigt ist, wird das String Gadget geleert und das Programm wartet auf eine neue Eingabe.

Das alles passiert, weil ich erneut die Funktionalität der '#KNOPF_INTERAKTION' 'Case' Anweisung verändert habe, die beim drücken der 'Text einfügen' Schaltfläche aufgerufen wird. Hier sind die drei neuen Zeilen die ich der '#KNOPF_INTERAKTION' 'Case' Anweisung hinzugefügt habe:

```
...
Case #KNOPF_INTERAKTION
  AddGadgetItem(#LISTE_EINGABE, -1, GetGadgetText(#STRING_EINGABE))
  SetGadgetText(#STRING_EINGABE, "")
  SetActiveGadget(#STRING_EINGABE)
...
```

Die erste Zeile nach der 'Case' Anweisung fügt dem ListView-Gadget ein Element hinzu. Wenn Sie die Seite zu 'AddGadgetItem()' in der Hilfe öffnen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Gadget → AddGadgetItem), dann sehen Sie die von diesem Befehl benötigten Parameter. Der erste Parameter ist die PB Nummer des Gadgets dem Sie ein Element hinzufügen wollen, in unserem Fall ist '#LISTE_EINGABE' die Konstante die diese PB Nummer enthält. Der zweite Parameter gibt die Position in der Liste an, an der Sie den neuen Eintrag einfügen wollen. Denken Sie daran, das die Indizes bei '0' beginnen. In obigem Beispiel möchte ich erreichen, das der Text immer am Ende der Liste eingefügt wird, egal welcher Index gerade aktuell ist, weshalb ich diesen Parameter mit '-1' belege. Das weist den Befehl an, den Eintrag am Listenende einzufügen, egal wie groß die Liste ist.

Der dritte Parameter ist der eigentliche String den Sie dem Gadget hinzufügen wollen. Ich habe an dieser Stelle den 'GetGadgetText()' Befehl verwendet, um den zurückgegebenen String aus dem '#STRING_EINGABE' String Gadget als Parameter zu übergeben. 'AddGadgetItem()' hat mehr Parameter als ich verwendet habe, aber die weiteren sind optional und deshalb nicht zwingend nötig, weshalb ich sie nicht benutzt habe.

Die zweite Zeile nach der 'Case' Anweisung verwendet den 'SetGadgetText()' Befehl mit einem leeren String, um den Wert des Gadgets auf nichts zu setzen. Nachdem ich den Wert in das ListView-Gadget übertragen habe, möchte ich das String Gadget leeren, damit es für eine neue Eingabe bereit ist.

Die letzte Zeile verwendet noch einmal den 'SetActiveGadget()' Befehl, um dem '#STRING_EINGABE' String Gadget erneut den Fokus zu geben. Dadurch wird der Text Cursor erneut im String Gadget positioniert und erspart dem Benutzer wieder einen Mausklick.

Was haben wir bis jetzt gelernt?

Ich hoffe, das die letzten paar Beispiele eine gute Einführung in die Programmierung von Grafischen Benutzeroberflächen waren und eine gute Basis für das weitere Lernen bilden. In diesem Buch kann ich Ihnen nicht zu allen erdenklichen Benutzeroberflächen ein Beispiel geben. Ich kann Ihnen lediglich die Grundlegenden Bausteine erklären, auf die Sie dann aufbauen können.

Der Anfang beim erstellen von Benutzeroberflächen in PureBasic ist immer der selbe. Zuerst müssen Sie ein Fenster mit dem 'OpenWindow()' Befehl öffnen. Als zweiten Schritt müssen Sie bei Bedarf ein paar Gadgets einfügen und als dritten Schritt müssen Sie eine Hauptschleife erstellen, um das Programm am laufen zu halten und auftretende Ereignisse zu behandeln. Alle Ereignisse die auftreten werden vom 'WaitWindowEvent()' oder vom 'WindowEvent()' Befehl innerhalb der Hauptschleife zurückgegeben.

Lesen Sie den Gadget Abschnitt in der Hilfe

Wenn Sie das bisher geschriebene verstanden haben, dann sind Sie auf einem guten Weg, zu begreifen wie Benutzeroberflächen in PureBasic realisiert werden. Wenn Sie die Grundlagen verstanden haben besteht der Rest der Arbeit nur noch daraus, zu lernen welche Gadgets es gibt und wie Sie benutzt werden. Dieses Wissen können Sie durch das Lesen der Hilfe im Abschnitt Gadget (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Gadget) erlangen. Ich empfehle Ihnen diesen Abschnitt von vorne bis hinten durchzulesen um zu verstehen welche Gadgets für Ihren Anwendungsfall geeignet sind und welche Befehle mit ihnen funktionieren.

Nehmen wir zum Beispiel den 'SetGadgetText()' Befehl, dieser kann mit vielen verschiedenen Gadgets verwendet werden. Wenn Sie auf die Hilfeseite des 'SetGadgetText()' Befehls schauen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Gadget → SetGadgetText), sehen Sie alle Gadgets die diesen Befehl unterstützen. Zusätzlich erhalten Sie noch spezielle Informationen zu den bestimmten Gadgets.

Die Auflistung aller Gadgets und Befehle die mit ihnen arbeiten würde den Rahmen dieses Buches sprengen. Ich denke, dass es wohl besser ist wenn Sie sich in der Hilfe selbst mit den Gadgets vertraut machen.

Wenn Sie einen schnellen Überblick benötigen welche Gadgets in PureBasic verfügbar sind, dann schauen Sie im Anhang B (Hilfreiche Tabellen) nach. Dort finden Sie eine vollständige Liste aller Gadgets und eine kurze Beschreibung zu Jedem.

Hinzufügen eines Menüs

Das Hinzufügen eines Menüs zu einer Benutzeroberfläche ist in PureBasic sehr einfach und vergleichbar dem Hinzufügen von Gadgets. Wenn Menüelemente angewählt werden, lösen sie wie ein Gadget ein Ereignis aus. Diese Ereignisse werden wie die Ereignisse von Gadgets in der Hauptschleife behandelt.

Alle Menüelemente werden wie Gadgets mit ihrer eigenen PB Nummer definiert und über diese Nummer werden Sie beim behandeln der Ereignisse auseinander gehalten. Hier ein kleines Beispiel:

```
Enumeration
  #FENSTER_HAUPT
  #MENU_HAUPT
  #MENU_ENDE
  #MENU_UEBER
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Menü Beispiel", #FLAGS)
  If CreateMenu(#MENU_HAUPT, WindowID(#FENSTER_HAUPT))

    MenuTitle("Datei")
    MenuItem(#MENU_ENDE, "Beenden")
    MenuTitle("Hilfe")
    MenuItem(#MENU_UEBER, "Über...")

    Repeat
      Ereignis.i = WaitWindowEvent()
      Select Ereignis
        Case #PB_Event_Menu
          Select EventMenu()
            Case #MENU_ENDE
              Ende = #True
            Case #MENU_UEBER
              MessageRequester("Über", "Hier beschreiben Sie Ihr Programm.")
          EndSelect

      EndSelect

    Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

  EndIf
EndIf
End
```

Diese kleine Beispiel zeigt ein übliches Menü, das einem Fenster hinzugefügt wurde. Sie sehen zwei Menütitel, die jeweils ein Menüelement enthalten. Wenn Sie in obiges Beispiel schauen, können Sie sehen wie ein Derartiges Menü erstellt wird.

Zuerst müssen wir ein Fenster erzeugen, da ein Menü einem Fenster zugeordnet wird. Nachdem wir das Fenster erstellt haben verwenden wir den 'CreateMenu()' Befehl um ein Menü Gerüst zu erstellen. Dieser Befehl benötigt zwei Parameter, der Erste ist die PB Nummer dieses Menü Objektes und der Zweite ist der Betriebssystem Identifikator des Fensters, dem das Menü zugeordnet ist. Hier verwende ich den 'WindowID()' Befehl, um den Betriebssystem Identifikator des Hauptfensters zu ermitteln.

Nachdem das Menügerüst erfolgreich erstellt wurde können Sie es mit den eigentlichen Menütiteln und Menüelementen Befüllen. Hierzu beginne ich mit dem 'MenuTitle()' Befehl. Dieser Befehl erstellt einen Menütitel, der am oberen Rand des Fensters angezeigt wird (oder im Falle von Mac OS X am oberen Rand des Bildschirms). Dieser Befehl benötigt einen String Parameter, der den Namen des Menütitels enthält.

Wenn ein Menütitel definiert wurde, werden von nun an alle erstellten Menüelemente unterhalb dieses Titels erscheinen. Menüelemente werden mit dem 'MenuItem()' Befehl erstellt. Dieser Befehl benötigt zwei Parameter, der Erste ist die PB Nummer mit der das Element verknüpft ist, und der Zweite ist der String, mit dem das Element im Menü dargestellt werden soll.

In meinem Beispiel habe ich unterhalb des 'Datei' Menütitels ein Element mit dem Namen 'Beenden' definiert. Dann habe ich erneut den 'MenuTitle()' Befehl verwendet, um einen weiteren Menütitel mit dem Namen 'Hilfe' zu erstellen. Wenn ein weiterer Menütitel erstellt wird, werden von nun an alle neu hinzugefügten Elemente unterhalb dieses Titels angezeigt, so wie das neu erstellte 'Über...' Menüelement.

Erkennen von Menüereignissen

Wenn das Menü mit allen Menütiteln und Elementen erstellt wurde, können wir die von ihnen Ausgelösten Ereignisse in der Hauptschleife erfassen. Genau wie bei der Auswertung der Gadgets verwenden wir einen

globalen Ereigniswert mit dem Namen '#PB_Event_Menu'.

Wenn das vom 'WaitWindowEvent()' zurückgemeldete Ereignis gleich '#PB_Event_Menu' ist, dann handelt es sich um ein Ereignis von einem Menüelement. Ich habe diesen Befehl in einer 'Select' Anweisung verwendet, wie hier:

```
...
Select Ereignis
  Case #PB_Event_Menu
    Select EventMenu()
      Case #MENU_ENDE
        Ende = #True
      Case #MENU_UEBER
        MessageRequester("Über", "Hier beschreiben Sie Ihr Programm.")
    EndSelect
EndSelect
...
```

Um genau zu ermitteln welches Menüelement der Auslöser war, müssen wir den 'EventMenu()' Befehl verwenden. Dieser Befehl gibt die PB Nummer des auslösenden Menüelements zurück. Ich habe diesen Befehl in einer weiteren 'Select' Anweisung verwendet, um mehrere Ereignisse auswerten zu können. Auf diese Weise kann ich elegant die Ereignisse von jedem beliebigen Menüelement auswerten. Wenn in dieser 'Select' Anweisung der vom 'EventMenu()' Befehl zurückgegebene Wert gleich '#MENU_ENDE' ist, dann wurde das 'Beenden' Element angewählt. In diesem Fall weise ich der Variable 'Ende' den Wert True zu, womit die Hauptschleife endet und damit auch das Programm. Wenn der vom 'EventMenu()' Befehl zurückgegebene Wert gleich '#MENU_UEBER' ist, dann wurde das 'Über...' Element angewählt und ich zeige ein Nachrichtenfenster mit Informationen über mein Programm an.

Untermenü Elemente

Bis jetzt habe ich Ihnen gezeigt wie Sie Standard Menüs erstellen können, Sie können mit PureBasic auch sogenannte Untermenüs erstellen. Es handelt sich um Menüs, die wie ein Ast aus dem Hauptmenü herausragen. Damit lässt sich ein baumartig strukturiertes Menü wie das 'Start' Menü von Windows erstellen. Alle in diesem Untermenü angeordneten Elemente erzeugen Ereignisse auf die gleiche Weise wie alle anderen Menüelemente und können auf die gleiche Weise in der Hauptschleife behandelt werden. Hier ein Beispiel:

```
Enumeration
  #FENSTER_HAUPT
  #MENU_HAUPT
  #MENU_UNTER
  #MENU_ENDE
  #MENU_UEBER
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Menü Beispiel", #FLAGS)
  If CreateMenu(#MENU_HAUPT, WindowID(#FENSTER_HAUPT))

    MenuTitle("Datei")
    OpenSubMenu("Hauptmenü Element")
    MenuItem(#MENU_UNTER, "Untermenü Element")
    CloseSubMenu()
    MenuBar()
    MenuItem(#MENU_ENDE, "Beenden")
    MenuTitle("Hilfe")
    MenuItem(#MENU_UEBER, "Über...")

    Repeat
      Ereignis.i = WaitWindowEvent()
      Select Ereignis
        Case #PB_Event_Menu
          Select EventMenu()
            Case #MENU_UNTER
              Debug "Untermenü Element angewählt."
            Case #MENU_ENDE
              Ende = #True
```

```
            Case #MENU_UEBER
              MessageRequester("Über", "Hier beschreiben Sie Ihr Programm.")
          EndSelect

      EndSelect

    Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

  EndIf
EndIf
End
```

Ein Untermenü wird durch das Verwenden des 'OpenSubMenu()' Befehls nach einem 'MenuTitle()' Befehl erstellt. Das Untermenü gehört zum Menütitel wie jedes andere Menüelement. Der 'OpenSubMenu()' Befehl benötigt einen Parameter. Hierbei handelt es sich um einen String, der vor dem Untermenü Pfeil angezeigt wird. Wenn der 'OpenSubMenu()' Befehl verwendet wurde, werden alle nachfolgenden Menüelemente, die durch den 'MenuItem()' Befehl hinzugefügt werden, dem Untermenü zugeordnet. Zum abschließen des Untermenüs müssen Sie den 'CloseSubMenu()' Befehl verwenden, der die Menüerstellung wieder an das Hauptmenü übergibt. Danach können Sie dem Hauptmenü weitere Elemente hinzufügen.

Die dem Untermenü hinzugefügten Elemente erzeugen Ereignisse, die auf die gleiche Weise wie Ereignisse von jedem anderen Menüelement in der Hauptschleife ausgewertet werden können. Wenn Sie die 'Select' Anweisung des 'EventMenu()' Befehls betrachten, sehen Sie, das ich einen neuen zu behandelnden Wert eingebaut habe: '#MENU_UNTER'. Wenn der zurückgegebene Wert von 'EventMenu()' gleich dem Wert von '#MENU_UNTER' ist, dann wurde das Element im Untermenü angeklickt. In diesem Fall gebe ich einen Text im Debug Ausgabefenster aus.

Separieren von Menüelementen

Wenn Sie einen genaueren Blick auf das obige Beispiel werfen sehen Sie, dass ich dort einen weiteren neuen Befehl eingefügt habe. Dieser Befehl zeichnet einen schönen grafischen Separator in das Menü, was sehr nützlich ist um einzelne Menüelemente zu separieren. Dieser neue Befehl ist der 'MenuBar()' Befehl. Er benötigt keine Parameter und kann an jeder Stelle verwendet werden, an der Sie einen Menüseparator benötigen. Üblicherweise platziert man einen Separator vor dem 'Beenden' Menüelement, um dieses von den anderen Elementen im 'Datei' Menü abzutrennen, und genau das habe ich in obigem Beispiel getan.

Kombinieren von Gadgets und Menüs im gleichen Programm

Das Einbinden von Menüs und Gadgets im gleichen Programm ist sehr einfach. Zuerst müssen Sie ein Fenster öffnen und ein Menü erstellen. Danach fügen Sie Ihre Gadgets hinzu, so einfach ist das. Hier ein Code Beispiel:

```
Enumeration
  #FENSTER_HAUPT
  #MENU_HAUPT
  #MENU_ENDE
  #MENU_UEBER
  #TEXT_EINGABE
  #STRING_EINGABE
  #LISTE_EINGABE
  #KNOPF_INTERAKTION
  #KNOPF_SCHLIESSEN
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 222, "Interaktion", #FLAGS)
  If CreateMenu(#MENU_HAUPT, WindowID(#FENSTER_HAUPT))
    MenuTitle("Datei")
    MenuItem(#MENU_ENDE, "Beenden")
    MenuTitle("Hilfe")
    MenuItem(#MENU_UEBER, "Über...")
    TextGadget(#TEXT_EINGABE, 10, 10, 280, 20, "Bitte Text eingeben:")
    StringGadget(#STRING_EINGABE, 10, 30, 280, 20, "")
    ListViewGadget(#LISTE_EINGABE, 10, 60, 280, 100)
    ButtonGadget(#KNOPF_INTERAKTION, 10, 170, 120, 20, "Text Eingabe")
    ButtonGadget(#KNOPF_SCHLIESSEN, 190, 170, 100, 20, "Schließe Fenster")
    SetActiveGadget(#STRING_EINGABE)
    Repeat
```

```
        Ereignis.i = WaitWindowEvent()
        Select Ereignis
          Case #PB_Event_Menu
            Select EventMenu()
              Case #MENU_ENDE
                Ende = #True
              Case #MENU_UEBER
                MessageRequester("Über", "Das ist Ihre Programmbeschreibung.")
            EndSelect
          Case #PB_Event_Gadget
            Select EventGadget()
              Case #KNOPF_INTERAKTION
                AddGadgetItem(#LISTE_EINGABE, -1, GetGadgetText(#STRING_EINGABE))
                SetGadgetText(#STRING_EINGABE, "")
                SetActiveGadget(#STRING_EINGABE)
              Case #KNOPF_SCHLIESSEN
                Ende = #True
            EndSelect
        EndSelect
      Until Ereignis = #PB_Event_CloseWindow Or Ende = #True
    EndIf
  EndIf
  End
```

Um die Ereignisse von Menüelementen oder von Gadgets auszuwerten, können Sie die verschiedenen globalen Konstanten verwenden. In obigem Beispiel habe ich beide in der selben 'Select' Anweisung verwendet, die die 'Ereignis' Variable folgendermaßen prüfte:

```
...
Select Ereignis
  Case #PB_Event_Menu
    Select EventMenu()
      ...
      ;hier werden die Menüereignisse ausgewertet
      ...
    EndSelect
  Case #PB_Event_Gadget
    Select EventGadget()
      ...
      ;hier werden die Gadgetereignisse ausgewertet
      ...
    EndSelect
EndSelect
...
```

Wenn ein Ereignis von einem Menüelement ausgelöst wird, dann wird es von der '#PB_Event_Menu' 'Case' Anweisung ausgewertet. Das exakte Menüelement wird dann vom 'EventMenu()' Befehl ermittelt. Wenn ein Ereignis von einem Gadget ausgelöst wird, dann wird es von der '#PB_Event_Gadget' 'Case' Anweisung ausgewertet. Das exakte Gadget wird dann vom 'EventGadget()' Befehl ermittelt. Das ermöglicht Ihnen die Behandlung von allen Menü- und Gadget Ereignissen in einer verschachtelten 'Select' Anweisung, wie es in obigem Beispiel gezeigt wird.

Hinzufügen von Icons zu Menüelementen

Sie haben sicher in Abb. 30 die Icons in der Menüleiste bemerkt. PureBasic stellt einen einfachen Befehl zur Verfügung, mit dem Sie selbst solche Icons in Menüs einfügen können. Wenn Sie anstelle des 'CreateMenu()' Befehls den Befehl 'CreateImageMenu()' verwenden, können Sie an jedes Menüelement eine 'ImageID' als Parameter anhängen. Das mit dieser 'ImageID' verknüpfte Bild wird dann im Menü angezeigt. Schauen Sie in die PureBasic Hilfe, um mehr über diesen Befehl zu erfahren.

Menü Tastenkürzel

Die meisten heutigen Programme stellen eine Form von Tastenkürzeln zur Verfügung um Menüelemente anzuwählen. Diese Tastenkürzel sind in der Regel Tastenkombinationen, die Sie drücken können um ein Menüereignis auszulösen, ohne dass Sie das Menü physikalisch öffnen müssen. PureBasic kann diese

Tastenkürzel sehr einfach in Ihr Programm einbauen. Um das zu demonstrieren, müssen wir ein Fenster mit einem Menü erstellen. In diesem Menü müssen wir unsere Menüelemente definieren, aber dieses mal definieren wir diese mit Tastenkürzeln.

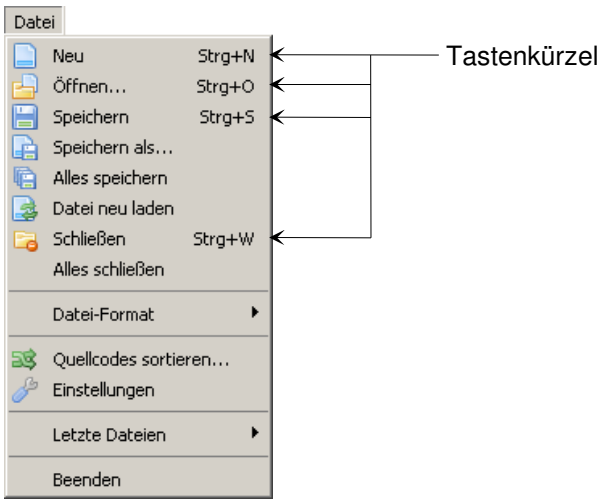


Abb. 30

Ein Tastenkürzel wird durch den String hinter dem Menüelement beschrieben. Dieser String zeigt, welche Tastenkombination Sie drücken müssen, um das entsprechende Menüereignis auszulösen. Abb. 30 zeigt die Tastenkürzel die im Datei Menü der PureBasic IDE definiert sind. Wenn Sie auf die Abbildung schauen können Sie sehen, dass Sie die Tastenkombination 'Strg + N' drücken können (gleichzeitiges drücken der Tasten 'Strg' und 'N'), um ein neues Dokument in der PureBasic IDE zu erstellen.

Wie Sie ebenfalls in Abb. 30 sehen können, sind die Tastenkürzel rechts von den Menüelementen angeordnet. Dadurch heben sich die Tastenkürzel vom Text der Menüelemente ab, was Verwechslungen vermeidet.

Im nächsten Beispielcode werde ich das vorherige Menübeispiel um Tastenkürzel und Tastenkombinationen erweitern. Starten Sie den Code und probieren Sie weitere Kombinationen aus.

```
Enumeration
  #FENSTER_HAUPT
  #MENU_HAUPT
  #M_ENDE
  #M_UEBER
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Menü Beispiel", #FLAGS)
  If CreateMenu(#MENU_HAUPT, WindowID(#FENSTER_HAUPT))

    MenuTitle("Datei")
    MenuItem(#M_ENDE, "Beenden" + #TAB$ + "Strg+Q")
    MenuTitle("Hilfe")
    MenuItem(#M_UEBER, "Über..." + #TAB$ + "Strg+A")

    AddKeyboardShortcut(#FENSTER_HAUPT,#PB_Shortcut_Control|#PB_Shortcut_Q,#M_ENDE)
    AddKeyboardShortcut(#FENSTER_HAUPT,#PB_Shortcut_Control|#PB_Shortcut_A,#M_UEBER)

    Repeat
      Ereignis.i = WaitWindowEvent()
      Select Ereignis
        Case #PB_Event_Menu
          Select EventMenu()
            Case #M_ENDE
              Ende = #True
            Case #M_UEBER
              MessageRequester("Über", "Hier beschreiben Sie Ihr Programm.")
          EndSelect
```

```
        EndSelect

      Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

    EndIf
  EndIf
  End
```

Wenn Sie dieses Beispiel starten werden Sie sehen, dass neben den Menüelementen nun Tastenkürzel vorhanden sind. Diese werden unter Verwendung eines TAB Zeichens im Menüstring definiert, wie hier:

```
...
MenuItem(#M_ENDE, "Beenden" + #TAB$ + "Strg+Q")
...
```

Dieser Befehl benötigt zwei Parameter, aber der zweite String Parameter setzt sich aus drei Teilen zusammen. Der erste Teil ist der Menüelement String, welcher in diesem Fall "Beenden" ist. Dann hängen wir ein TAB Zeichen an den String an. Hierzu verwenden wir die eingebaute '#TAB$' String Konstante. Diese Konstante hat den ASCII Wert '9' in String Form. Dann hängen wir einen weiteren String an. Hierbei handelt es sich um das Tastenkürzel, in diesem Fall 'Strg+Q'.

An dieser Stelle wird ein TAB Zeichen anstelle eines Leerzeichens verwendet, damit die Tastenkürzel immer rechts vom Menü sind, unabhängig davon wie lang der String des Menüelementes ist.

Nachdem wir die Tastenkombination neben dem Menü eingegeben haben, ist der visuelle Teil der Arbeit erledigt. Dieser Teil dient lediglich dazu, den Benutzer über das Vorhandensein eines Tastenkürzels zu informieren. Um die Funktionalität für das Tastenkürzel einzubauen müssen wir den 'AddKeyboardShortcut()' Befehl verwenden. Ich habe diesen Befehl in meinem Code folgendermaßen verwendet:

```
...
AddKeyboardShortcut(#FENSTER_HAUPT, #PB_Shortcut_Control|#PB_Shortcut_Q, #M_ENDE)
AddKeyboardShortcut(#FENSTER_HAUPT, #PB_Shortcut_Control|#PB_Shortcut_A, #M_UEBER)
...
```

Dieser Befehl benötigt drei Parameter. Der Erste ist die PB Nummer des Fensters in dem das Tastenkürzel verankert wird, in diesem Fall '#FENSTER_HAUPT'. Der zweite Parameter ist ein Wert, der die eigentliche Tastenkombination repräsentiert. Dieser Wert wird aus eingebauten Konstanten gebildet, die verschiedene Tastenkombinationen auf der Tastatur repräsentieren. In der ersten Zeile in obigem Codeausschnitt habe ich die Konstanten '#PB_Shortcut_Control' und #PB_Shortcut_Q' miteinander verknüpft um die Tastenkombination 'Strg+Q' zu erhalten. Diese beiden Konstanten werden mit dem bitweise 'ODER' Operator ('|') verknüpft. Alle verfügbaren Tasten Konstanten, die Sie zur Erstellung von Tastenkombinationen verwenden können, sind in der PureBasic Hilfe aufgelistet (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Window → AddKeyboardShortcut). Der Dritte Parameter ist die PB Nummer des Menüelements, dem Sie die Tastenkombination zuweisen möchten. In der ersten Zeile des obigen Codeausschnitts ist das die Konstante '#M_ENDE' , die ich zuvor auch zum Erstellen des Menüelements 'Beenden' verwendet habe. Das bedeutet, das beim drücken der Tastenkombination auf der Tastatur das gleiche Ereignis ausgelöst wird, welches beim anwählen des Menüelements 'Beenden' erzeugt wird.

In der nächsten Zeile des Codeausschnitts habe ich diesen Befehl einfach mit anderen Parametern wiederholt, um ein Tastenkürzel für das 'Über...' Menüelement zu erzeugen.

Die Ereignisse die von diesen Tastenkürzeln erzeugt werden, werden auf die gleiche Weise behandelt wie die Ereignisse die von den entsprechenden Menüelementen erzeugt werden und es gibt absolut keinen Unterschied in der Hauptschleife, wie Sie in obigem Beispiel sehen können.

Tastenkürzel ohne ein Menü

Mit den gleichen Befehlen können Sie auch Tastenkürzel definieren, die unabhängig von einem Menü sind. Sie müssen nur den 'AddKeyboardShortcut()' Befehl verwenden und ein Tastenkürzel erstellen, das nicht mit einem Menüelement sondern mit dem erstellten Fenster verknüpft wird. An Stelle der PB Nummer des Menüelementes übergeben Sie als dritten Parameter einfach eine einmalige Zahl. In der Hauptschleife prüfen Sie dann in den Menüereignissen auf diesen Wert, auch wenn kein Menü vorhanden ist. Hier ein Beispiel:

```
#FENSTER = 1
#KUERZEL = 2

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER, 0, 0, 300, 200, "Hallo Welt", #FLAGS)
  AddKeyboardShortcut(#FENSTER, #PB_Shortcut_Control | #PB_Shortcut_Z, #KUERZEL)
  Repeat
    Ereignis.i = WaitWindowEvent()
    Select Ereignis
      Case #PB_Event_Menu
        Select EventMenu()
          Case #KUERZEL
            Debug "Das Tastenkürzel wurde gedrückt"
        EndSelect
    EndSelect
  Until Ereignis = #PB_Event_CloseWindow
EndIf
End
```

Wenn Sie obiges Beispiel starten und 'Strg+Z' drücken, dann sehen Sie den Text 'Das Tastenkürzel wurde gedrückt' im Debug Ausgabefenster. Dieses Beispiel funktioniert, weil der 'AddKeyboardShortcut()' Befehl ein Menüereignis erzeugt das in der Hauptschleife korrekt behandelt wird, auch wenn dem Fenster physikalisch kein Menü zugeordnet ist. Das ist ein praktischer Trick den Sie immer dann anwenden sollten, wenn Sie Ihrem Programm ein Tastenkürzel, aber kein Menü hinzufügen wollen.

Einbinden von Grafiken in Ihr Programm

Manchmal möchten Sie in Ihre Programme sicher Bilder einbinden und PureBasic macht dieses Vorhaben sehr einfach. Im folgenden Abschnitt werde ich Ihnen einen von zwei Wegen zeigen, wie Sie ihrer Oberfläche Bilder hinzufügen können. Die erste Möglichkeit ist das Bild von einer externen Quelle einzulesen, zum Beispiel von einer Bilddatei im Programmverzeichnis. Die zweite Möglichkeit ist das direkte Einbinden der Grafik in Ihr Programm, wodurch Sie nur eine ausführbare Datei erhalten. Beide Möglichkeiten führen zum gleichen Ergebnis, die Wahl der Methode ist davon abhängig wie Sie die Bilddatei selbst weiter behandeln wollen. Beide Methoden benötigen zum Anzeigen des Bildes ein Image Gadget das auf Ihrer Oberfläche platziert werden muss. Dieses wird auf die gleiche Weise wie jedes andere Gadget in Ihrem Fenster platziert und es folgt den gleichen Regeln

Laden von Bildern in Ihr Programm

Die Methode des Ladens von Bildern in Ihre Oberfläche ist von externen Bilddateien abhängig. Anders als bei den eingebetteten Bildern muss bei dieser Methode zur Programmdatei auch die Bilddatei geliefert werden. Hier ein Beispiel:

```
Enumeration
  #FENSTER_HAUPT
  #BILD_DATEI
  #BILD_ANZEIGE
  #KNOPF_ENDE
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Bild Beispiel", #FLAGS)
  If LoadImage(#BILD_DATEI, "bild.bmp")

    ImageGadget(#BILD_ANZEIGE, 10, 10, 280, 150, ImageID(#BILD_DATEI))
    ButtonGadget(#KNOPF_ENDE, 100, 170, 100, 20, "Fenster schließen")
    Repeat

      Ereignis.i = WaitWindowEvent()
      Select Ereignis
        Case #PB_Event_Gadget
          Select EventGadget()
            Case #KNOPF_ENDE
              Ende = #True
          EndSelect
      EndSelect
```

```
      Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

    EndIf
  EndIf
  End
```

Damit dieses Beispiel zweckmäßig funktioniert benötigen Sie ein Bild mit der Größe von '280 x 150' Pixel im Bitmap Format. Diese Bilddatei muss sich mit dem Quelltext im gleichen Verzeichnis befinden. Wenn sich alle Elemente am richtigen Ort befinden und das Programm gestartet wird, sieht es ähnlich Abb. 31 aus.

In diesem Beispiel habe ich den 'LoadImage()' Befehl verwendet, um eine Bilddatei von der Festplatte in den Speicher zu laden, damit ich es direkt im Programm anzeigen kann, wie hier:

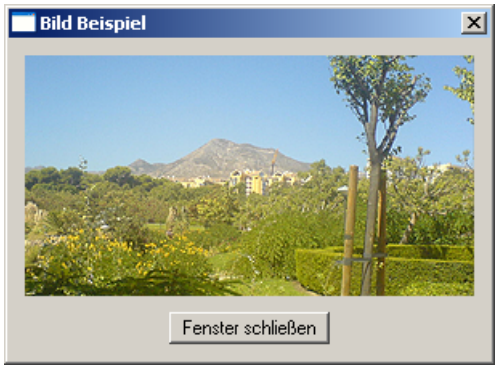
```
...
LoadImage(#BILD_DATEI, "bild.bmp")
...
```

Wenn Sie den 'LoadImage()' Befehl in der PureBasic Hilfedatei betrachten (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Image → LoadImage) können Sie sehen, dass der Befehl drei Parameter benötigt. Der erste Parameter ist die PB Nummer die dem geladenen Bild zugewiesen wird, der Zweite ist die Stelle auf der Festplatte, an der die Bilddatei hinterlegt ist. Wenn im zweiten Parameter keine Pfadinformationen angegeben werden, bezieht sich der angegebene Name auf den relativen Pfad des Programms. Den dritten, optionalen Parameter, habe ich in diesem Beispiel nicht verwendet. Wie üblich prüfe ich den Rückgabewert des Befehls in einer 'If' Anweisung, um mich zu vergewissern dass das Bild korrekt geladen wurde, bevor ich mit der Programmausführung fortfahre.

Wenn das Bild ordnungsgemäß geladen wurde, können Sie es in einem Image Gadget in Ihrem Programm anzeigen. Schauen Sie auf das Image Gadget Beispiel, dort habe ich den 'ImageGadget()' Befehl folgendermaßen verwendet:

```
...
ImageGadget(#BILD_ANZEIGE, 10, 10, 280, 150, ImageID(#BILD_DATEI))
...
```

Der 'ImageGadget()' Befehl ist einfach zu verstehen. Der erste Parameter ist die PB Nummer die dem Gadget zugewiesen wird, nicht das geladene Bild. Die nächsten vier Parameter definieren die Größe und die Position des Gadgets, wie bei den anderen Gadgets. über den fünften Parameter definieren wir, welches Bild im Gadget angezeigt werden soll. An dieser Stelle muss der OS Indentifikator des zuvor geladenen Bildes übergeben werden. Da wir bereits ein passendes Bild geladen haben, können wir den 'ImageID()' Befehl verwenden, um den OS Identifikator zu ermitteln. In obigem Code habe ich den 'ImageID()' Befehl verwendet, um den OS Identifikator des '#BILD_DATEI' Bildes zu ermitteln. Dieses Bild habe ich zuvor mit dem 'LoadImage()' Befehl geladen.



(Microsoft Windows Beispiel) Abb. 31

Diese Möglichkeit Bilder zu laden und sie in einem Image Gadget anzuzeigen ist eine einfache Möglichkeit dieses zu erledigen, aber Sie dürfen bei der Weitergabe Ihres fertigen Programms nicht vergessen alle verwendeten Bilder mit dem fertigen Executable auszuliefern, da Ihr Programm immer nach den externen Bildern sucht, auch wenn es kompiliert ist. Wenn Sie Ihre Bilder komplett in das Executable einbinden wollen, lesen Sie den nächsten Abschnitt.

Einbinden von Bildern in Ihr Programm

Die vorherige Methode Bilder zu laden ist gut, aber manchmal möchten Sie vielleicht ein komplett eigenständiges Executable erstellen, in das alle verwendeten Bilder komplett eingebettet sind, so das Sie nicht vom Vorhandensein anderer Dateien abhängig sind. Diese Methode verwendet eine 'DATA' Sektion, mit deren Hilfe Sie Dateien in binärer Form in das Executable einbinden können. Hier ein Beispiel:

```
Enumeration
  #FENSTER_HAUPT
  #BILD_SPEICHER
  #BILD_ANZEIGE
  #KNOPF_ENDE
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Bild Beispiel", #FLAGS)

  If CatchImage(#BILD_SPEICHER, ?Bild)
    ImageGadget(#BILD_ANZEIGE, 10, 10, 280, 150, ImageID(#BILD_SPEICHER))
    ButtonGadget(#KNOPF_ENDE, 100, 170, 100, 20, "Fenster schließen")

    Repeat

      Ereignis.i = WaitWindowEvent()
      Select Ereignis
        Case #PB_Event_Gadget
          Select EventGadget()
            Case #KNOPF_ENDE
              Ende = #True
          EndSelect
      EndSelect

    Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

  EndIf
EndIf
End

DataSection
  Bild:
  IncludeBinary "bild.bmp"
EndDataSection
```

Dieses Beispiel ist dem vorhergehenden sehr ähnlich, außer dass diesmal das Bild in eine Data Sektion eingebunden wurde, während wir das Programm kompiliert haben. Wenn dieses Programm gestartet wird, wird das Bild aus der Data Sektion gelesen und nicht von der Festplatte. Wenn Sie in den Quelltext schauen, finden Sie die Data Sektion am Ende des Programmcodes:

```
...
DataSection
  Bild:
  IncludeBinary "bild.bmp"
EndDataSection
```

Die Befehle 'DataSection' und 'EndDataSection' definieren den Beginn und das Ende des Data Bereiches. Innerhalb dieses Bereiches haben Sie die Möglichkeit mit dem 'IncludeBinary' Befehl Daten in diesen Bereich einzufügen. Dieser Befehl verwendet keine Klammern, er benötigt lediglich einen String als Parameter, der die einzubindende Datei definiert.

In diesem Beispiel habe ich die Datei 'bild.bmp' eingebunden. Sie werden auch bemerkt haben, das ich vor dem 'IncludeBinary' Befehl eine Sprungmarke definiert habe, sehr ähnlich der Definition von Unterroutinen. Dadurch erhält Ihr Programm die Möglichkeit, die Startadresse des Bildes im Speicher zu ermitteln. Wenn ein Bild auf diese Weise in das Programm eingebunden wurde, können wir es auf einfache Weise in unserem Hauptprogramm verwenden.

Im Hauptprogramm benötigen wir nicht länger den 'LoadImage()' Befehl um das Bild in den Speicher zu laden. Wenn das Bild in die Data Sektion eingebettet wurde, wird es bereits geladen wenn das Programm

startet. Deshalb verwenden wir den 'CatchImage()' Befehl um das Bild aus der Data Sektion zu laden, wie hier:

```
...
CatchImage(#BILD_SPEICHER, ?Bild)
...
```

Wenn Sie den 'CatchImage()' Befehl in der PureBasic Hilfe betrachten (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Image → CatchImage) sehen Sie, dass dieser Befehl drei Parameter benötigt. Der Erste ist die PB Nummer die mit diesem Bild verknüpft wird. Der Zweite Parameter ist die Speicheradresse, an der das Bild zu finden ist. Erinnern Sie sich, dass ich eine Sprungmarke in der Data Sektion verwendet habe, weshalb ich von dieser die Speicheradresse ermitteln muss. Das erreiche ich mit einem Fragezeichen vor dem Sprungmarkennamen, den ich als Parameter verwende. Dieses Fragezeichen ist eine spezielle Ein-Zeichen Funktion, die die Speicheradresse einer beliebigen Sprungmarke zurückgibt. Dieser Vorgang wird in Kapitel 13 (Zeiger) etwas ausführlicher erklärt. Den dritten Parameter habe ich in diesem Beispiel nicht verwendet, da er optional ist.

Das Anzeigen eines Bildes mit dem 'CatchImage()' Befehl funktioniert genau so wie das Anzeigen mit dem 'LoadImage()' Befehl. Auch hier verwenden wir wieder ein Image Gadget und übergeben ihm die Parameter auf die gleiche Weise wie zuvor:

```
...
ImageGadget(#BILD_ANZEIGE, 10, 10, 280, 150, ImageID(#BILD_SPEICHER))
...
```

Wenn Sie dieses Programm starten, sollte es wieder wie Abb. 31 aussehen. Das ist so, weil es sich um das gleiche Programm handelt, mit dem Unterschied, dass sie die externe 'bild.bmp' Datei nicht mehr benötigt wird.

Welche Bildformate können verwendet werden?
In den letzten paar Beispielen habe ich nur Bilder im Bitmap Format (*.bmp) geladen und ins Programm eingebunden, aber Sie sind nicht auf dieses Format beschränkt.

Standardmäßig kann PureBasic Dateien im Bitmap (*.bmp) und Icon (*.ico) Format laden und anzeigen, doch manchmal können diese Formate beschränkend sein. Wenn Sie andere Formate verwenden möchten, können Sie optionale Decoder Befehle verwenden, um den Befehlen zum laden von Bildern weitere Funktionalität zu geben. Diese anderen Decoder sind sehr einfach zu verwenden, Sie müssen nur den Decoder Befehl am Anfang Ihres Quelltextes verwenden. Von nun an bieten alle Befehle zum laden von Bildern volle Unterstützung für die vom Decoder unterstützten Bildformate. In der PureBasic Hilfe erhalten Sie mehr Informationen über die entsprechenden Decoder (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → ImagePlugin).

Hier sind die weiteren Decoder Befehle:

'UseJPEG2000ImageDecoder()'
Dieser Dekoder fügt Unterstützung für das JPEG 2000 (*-jpg/*-jpeg) Datei Format hinzu.

'UseJPEGImageDecoder()'
Dieser Dekoder fügt Unterstützung für das JPEG (*-jpg/*-jpeg) Datei Format hinzu.

'UsePNGImageDecoder()'
Dieser Dekoder fügt Unterstützung für das PNG (*.png) Datei Format hinzu.

'UseTIFFImageDecoder()'
Dieser Dekoder fügt Unterstützung für das TIFF (*.tif/*.tiff) Datei Format hinzu.

'UseTGAImageDecoder()'
Dieser Dekoder fügt Unterstützung für das TARGA (*.tga) Datei Format hinzu.

Wie bereits zuvor erwähnt, ist die Verwendung dieser Decoder Befehle extrem einfach. Lassen Sie uns dem letzten Beispiel die Unterstützung für JPEG Bilder hinzufügen, so dass wir JPEG Bilder an Stelle von Bitmap Bildern einfügen können:

```
UseJPEGImageDecoder()

Enumeration
  #FENSTER_HAUPT
  #BILD_SPEICHER
  #BILD_ANZEIGE
  #KNOPF_ENDE
EndEnumeration

Global Ende.i = #False
#FLAGS        = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 300, 200, "Bild Beispiel", #FLAGS)

  If CatchImage(#BILD_SPEICHER, ?Bild)
    ImageGadget(#BILD_ANZEIGE, 10, 10, 280, 150, ImageID(#BILD_SPEICHER))
    ButtonGadget(#KNOPF_ENDE, 100, 170, 100, 20, "Fenster schließen")

    Repeat

      Ereignis.i = WaitWindowEvent()
      Select Ereignis
        Case #PB_Event_Gadget
          Select EventGadget()
            Case #KNOPF_ENDE
              Ende = #True
          EndSelect
      EndSelect

    Until Ereignis = #PB_Event_CloseWindow Or Ende = #True

  EndIf
EndIf
End

DataSection
  Bild:
  IncludeBinary "bild.jpg"
EndDataSection
```

In diesem Beispiel wurden zur Unterstützung von JPEG Bildern nur zwei kleine Details gegenüber dem vorhergehenden Beispiel geändert. Am Anfang des Programms habe ich den 'UseJPEGImageDecoder()' Befehl eingefügt und habe das Bild in der Data Sektion durch ein Bild im JPEG Format ersetzt. Dieses Programm bindet nun ein JPEG Bild an Stelle eines Bitmap Bildes ein.

Alle anderen Decoder werden auf die gleiche Weise verwendet. Sie müssen nur den Decoder Befehl zu Beginn des Programms aufrufen, um Unterstützung für die weiteren Dateiformate zur Verfügung zu stellen.

Ein erster Blick auf den neuen 'Visual Designer'

Zusammen mit PureBasic erhalten Sie einen mächtigen 'Visual Designer', der es Ihnen ermöglicht, Oberflächen visuell und Dynamisch zu erstellen. Das bedeutet, das Sie buchstäblich Ihre Gadgets auf ein Fenster zeichnen können, um Ihre Oberfläche zu erstellen. Der darunterliegende PureBasic Code wird automatisch und in Echtzeit für Sie generiert.

Die Verwendung eines solchen Werkzeugs kann bei der Entwicklung von Oberflächen eine Menge Zeit sparen, da das aussehen Ihrer Oberfläche durch einfaches Verschieben von Objekten anpassen können und das resultierende Ergebnis sofort sehen. Sie haben weiterhin die Möglichkeit den Code wieder in den Visual Designer zu laden, wenn Sie etwas hinzufügen möchten oder die Funktionalität Ihres Programms erweitern wollen. Es gibt allerdings eine Beschränkung hinsichtlich des Ladens von bestehendem Code. Code der nicht vom Visual Designer erstellt wurde, wird möglicherweise nicht korrekt geöffnet wenn Sie ihn zum bearbeiten laden. Der Visual Designer erstellt den Code auf eine ganz bestimmte Weise, und wenn der handgeschriebene Code von diesem Format abweicht, hat der Designer Probleme diesen korrekt einzulesen. Der Visual Designer wurde wie alle anderen mitgelieferten Werkzeuge in PureBasic geschrieben und wird in der PureBasic Gemeinde aussagekräftig als 'VD' bezeichnet.

Warum hast Du das nicht früher erwähnt?

Ich dachte, dass es sehr wichtig für Sie ist, dass Sie zuerst lernen wie die Dinge in PureBasic funktionieren. Durch das erworbene Wissen hinsichtlich Gadgets, Menüs und Ereignissen haben Sie nun eine gute Basis, um den vom Visual Designer erstellten Code besser zu verstehen. Deshalb habe ich mich dazu entschlossen, den 'VD' erst am Ende dieses Kapitels zu erwähnen.

Das Hauptfenster

Wenn Sie den Visual Designer starten, wird er so ähnlich wie in Abb. 32 aussehen. Das Hauptfenster ist ein 'Multiple Document Interface' (MDI), das alle Werkzeuge enthält die Sie zum erstellen von Oberflächen benötigen. Der Visual Designer besteht aus einer Menüleiste und einer Werkzeugleiste darunter. Auf der linken Seite ist eine große Werkzeugtafel, auf der sich eine Gadget Palette und eine Eigenschaften Palette befindet. In der Mitte befindet sich das erstellte Fenster und darunter findet sich der dazugehörige erstellte Code.

Die Verwendung des Visual Designers

Wenn Sie auf Abb. 32 schauen, sehen Sie die sich in Bearbeitung befindende Oberfläche, die durch einen Pfeil auf der rechten Seite markiert wird. Dieses Fenster ist einfach zu bestücken, da es mit einer kleinen Punkten bestehenden Fläche überzogen ist. Diese Punkte sind ein Raster Gitter, das es ermöglicht die Gadgets darauf einrasten zu lassen. Dadurch wird das Ausrichten von Gadgets sehr schnell und einfach. Diese Punkte erscheinen nicht im fertigen Programm.

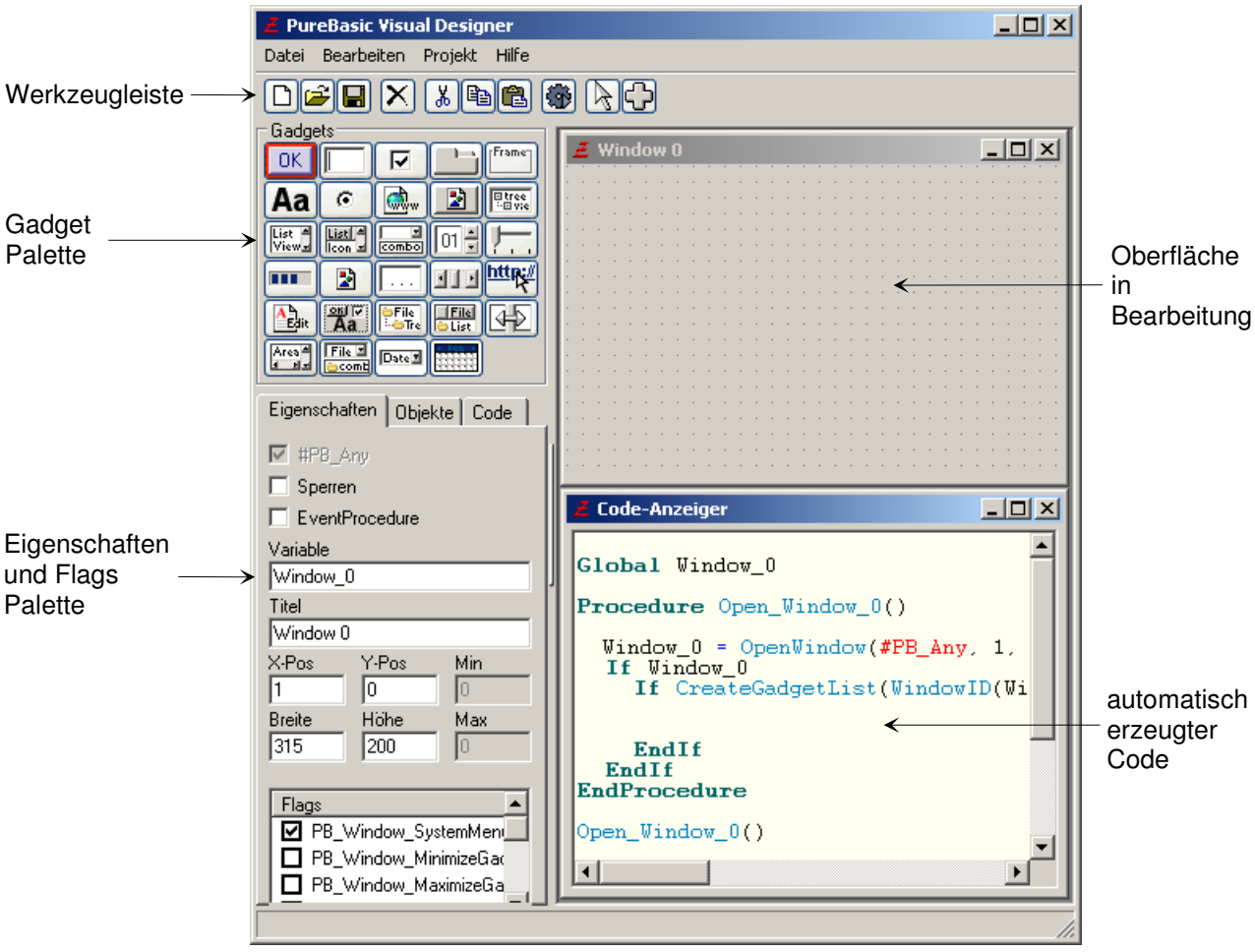


Abb. 32

Wenn Sie das Fenster auf die gewünschte Größe gebracht haben, können Sie damit beginnen Ihre Gadgets zu platzieren. Dazu wählen Sie in der Gadget Palette ein Werkzeug aus und zeichnen mit diesem Werkzeug auf dem Fenster. Damit zeichnen Sie ein dynamisches, in der Größe änderbares Gadget von der Art des ausgewählten Werkzeugs auf das Fenster. Nachdem Sie das Gadget gezeichnet haben sehen Sie, dass es mit blauen Punkten umrahmt ist. Diese Punkte dienen als Angriffspunkte, die es Ihnen ermöglichen das Gadget nach dem Zeichnen in der Größe zu ändern. Während Sie Gadgets zu Ihrer Oberfläche hinzufügen

können Sie im Code Viewer beobachten wie der automatisch generierte Code zu Ihrem Projekt hinzugefügt wird.

Nachdem Sie alle Gadgets platziert haben und die Gestaltung Ihrer Oberfläche abgeschlossen haben, können Sie Ihr Projekt mit einem Klick auf das Disketten Symbol in der Werkzeugleiste speichern. Dadurch wird der automatisch generierte Code in einer Standard PureBasic Quelltext Datei gespeichert, die Sie mit der IDE öffnen, bearbeiten und kompilieren können.

Vor-und Nachteile des Visual Designers

Obwohl der Visual Designer ein großartiges Werkzeug zum schnellen und einfachen erstellen von Benutzeroberflächen ist, reagiert er doch empfindlich auf die Veränderung des generierten Codes. Sie müssen die Art und Weise des generierten Codes akzeptieren und bei Veränderungen in der IDE müssen Sie sich streng an die vorgegebenen Regeln halten. Wenn Sie etwas erfahrener in PureBasic sind, ist das kein echtes Problem, auf Einsteiger kann der generierte Code aber sehr kompliziert und fortgeschritten wirken.

Der entscheidende Vorteil bei der Verwendung des Visual Designers ist die Geschwindigkeit. Sie können in Sekunden eine nützliche Benutzeroberfläche zusammenklicken und dann durch einfaches bewegen und in der Größe ändern die Gadgets am richtigen Ort platzieren. Das sollten Sie nicht unterschätzen. Die Zeit die sie beim erstellen von Benutzeroberflächen mit dem VD einsparen ist enorm, auch weil Sie die Zeit zum kompilieren sparen, um zu sehen wie die Oberfläche aussieht.

Einige Menschen mögen den Visual Designer und einige nicht, so einfach ist das. Einige behaupten, dass es das beste ist den Code selbst zu erstellen, damit Sie genau wissen was 'unter der Haube' passiert, während andere argumentieren, dass Sie in der heutigen Zeit unter Konkurrenzdruck eine schnelle Lösung zum erstellen von Oberflächen brauchen. Ich bin sicher, dass diese Diskussion weitergehen wird, aber letztendlich ist es Ihre Entscheidung wie Sie bei Ihren Projekten vorgehen. Die Werkzeuge sind vorhanden, Sie entscheiden ob Sie sie verwenden.

Erfahren Sie mehr über den Visual Designer

Der Visual Designer wird komplett mit einer Hilfedatei ausgeliefert, die das Arbeiten mit dem VD in einem ähnlichen Stil wie die IDE Hilfe erklärt. Um die Hilfe aufzurufen drücken Sie im Visual Designer die 'F1' Taste und die Hilfe wird geladen. Ich empfehle Ihnen sich diese Hilfe gründlich durchzulesen.

Die neue Version des Visual Designers (siehe Abb. 32) wird momentan entwickelt und kann von der Visual Designer Website heruntergeladen werden. Einen entsprechenden Internet Link finden Sie im Anhang A (Nützliche Internet Links).

Die Entwicklung des Visual Designer ruht nun schon einige Zeit und es gibt momentan keine verlässlichen Zusagen für eine Weiterentwicklung. Da sich in der Zwischenzeit auch einige Befehle in PureBasic geändert haben produziert der Visual Designer teilweise veralteten Code, weshalb eine Weiterverwendung nur noch eingeschränkt zu empfehlen ist. Das ist im Moment eine etwas unbefriedigende Situation, möglicherweise bringen neuere Versionen des PureBasic Paketes hier Besserung.

(Anmerkung des Übersetzers)

III. Grafik und Sound

Programme die Grafik und Sound verwenden sind die Voraussetzung für die Welt der Computerspiele und Interaktiven Anwendungen. Wenn Sie so geniale Sachen Programmieren möchten, brauchen Sie eine Programmiersprache die Sie bei der Umsetzung Ihrer Ideen unterstützt. PureBasic stellt einfache, mächtige und funktionsreiche Befehle zur Verfügung, mit deren Hilfe Sie qualitativ hochwertige Spiele, Demos, Bildschirmschoner und andere interaktive Anwendungen erstellen können. In diesem Abschnitt werde ich Ihnen die Grundlagen über die Verwendung von Grafik und Sound innerhalb von PureBasic Programmen näher bringen.

Zuerst werde ich Ihnen die 2D Zeichenbefehle erklären, die es Ihnen ermöglichen einfache Figuren, Linien und Text zu zeichnen. Anschließend werde ich Ihnen demonstrieren, wie Sie diese Bilder speichern können. Dann werde ich Ihnen zeigen, wie Sie einen Vollbild 'Screen' öffnen, in den Sie Grafiken zeichnen können. Weiterhin werde ich Ihnen die 'Sprites' erklären, und wie Sie diese netten grafischen Effekte erstellen und verwenden.

Danach werden wir zur 3D Grafik übergehen und ich werde Ihnen die mit PureBasic mitgelieferte 3D Engine sowie die Grundlagen der 3D Grafik in PureBasic erklären. Die 3D Engine nennt sich OGRE und ist eine qualitativ hochwertige Engine von einem Dritthersteller, mit deren Hilfe es möglich ist verblüffende Grafikeffekte zu erzeugen.

Das letzte Kapitel in diesem Abschnitt handelt von der Sound Programmierung. Es behandelt die Punkte laden und abspielen von Sounddaten mit dem Schwerpunkt Wave Dateien, Tracker Module, MP3's und CD Audio.

Die Kapitel in diesem Abschnitt sind keine kompletten Leitfäden, wie sie einen bestimmten grafischen Effekt erzielen oder wie Sie Spiele schreiben, das überlasse ich anderen Texten. Diese Kapitel dienen der Einführung in die Befehle der Bibliotheken die dieses ermöglichen. Diese Informationen bilden eine Basis, die es Ihnen ermöglicht eigene Experimente zu starten und damit Spaß zu haben. Ich hoffe, dass dieser Abschnitt Ihnen einen guten Überblick darüber gibt, was im grafischen Bereich mit PureBasic möglich ist und Sie inspiriert eigene Demos oder Spiele zu erstellen.

10. 2D-Grafik

In diesem Kapitel erkläre ich Ihnen, wie Sie mit PureBasic 2D Grafiken zeichnen. Der Begriff '2D Grafiken' umfasst ein weites Gebiet und viele Zeichenoperationen fallen unter diesen beschreibenden Schirm. Wenn ich in diesem Buch von 2D Grafiken spreche, meine ich einfach zweidimensionale Grafiken, die auf eine Grafische Benutzeroberfläche, auf 'Screens', auf Bilder im Speicher oder auch auf dem Drucker ausgegeben werden können. Diese Grafiken können von unterschiedlicher Komplexität sein. Sie reicht vom einfachen Pixel bis zum vollfarbigen Bild. Alle 2D Grafiken sind zweidimensional, was Sie aus dem Namen herleiten können, anders als 3D Grafiken, die 3D Modelle anzeigen und eine dritte 'Tiefen' Dimension haben.

Mit 2D Grafiken lassen sich einige eindrucksvolle Effekte erzeugen. Viele Spiele, Bildschirmschoner und Demos wurden mit PureBasic erstellt. Ich hoffe, dass es Ihnen möglich ist die Beispiele zu kopieren und von diesen zu lernen, damit Sie eigene Interessante Grafiken für Ihre Programme erstellen können.

2D Zeichenbefehle

PureBasic enthält Befehle, die es Ihnen erlauben einfache Formen und Farben in Ihr Programm zu zeichnen. All diese einfachen 2D Zeichenbefehle sind in der '2D Drawing' Bibliothek enthalten (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → 2DDrawing). Diese Befehle sind nützlich, um einfache Formen und Text in verschiedenen Farben zu zeichnen. Weiterhin gibt es auch ein paar Befehle, die es Ihnen ermöglichen auf ein zuvor erstelltes oder geladenes Bild zu zeichnen.

Worauf kann ich zeichnen?

Die integrierten 2D Zeichenbefehle können für verschiedene Ausgabekanäle verwendet werden, wie ein grafisches Benutzeroberflächen Fenster, einen PureBasic Screen, ein Sprite, ein im Speicher erstelltes Bild, eine Textur eines 3D Modells oder einfach der Drucker. Diese sechs Methoden decken weitestgehend alles ab, worauf Sie zeichnen können, und jede Methode ist so einfach wie die anderen. Die Verwendung dieser Ausgabemethoden ist in PureBasic eine einfache Sache, da Sie nur einmal definieren müssen, welchen Ausgabekanal Sie verwenden möchten, und dann mit den Zeichenbefehlen direkt auf diesen Ausgabekanal

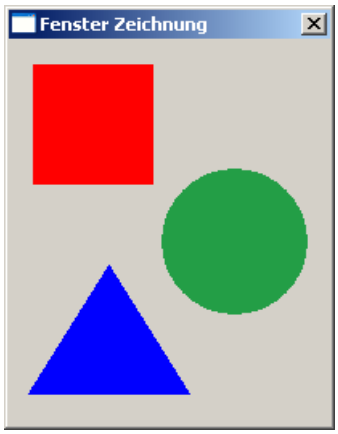
zeichnen können. Um die Sache weiter zu vereinfachen, verwenden alle Ausgabemethoden die gleiche Befehls Syntax

Zeichnen auf einem Fenster

Im ersten Beispiel zeige ich Ihnen, wie Sie 2D Grafiken direkt auf ein Fenster zeichnen. Sie werden das wahrscheinlich niemals in einer reellen Anwendung tun, aber es ist eine einfache Übung für Sie, damit Sie die Form der Syntax lernen. Schauen Sie auf diesen Code:

```
#FENSTER_HAUPT = 1

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER_HAUPT, 0, 0, 200, 240, "Fenster Zeichnung", #FLAGS)

  If StartDrawing(WindowOutput(#FENSTER_HAUPT))
    Box(15, 15, 75, 75, RGB(255, 0, 0))
    Circle(140, 125, 45, RGB(35, 158, 70))
    ;die nächsten 2D Zeichenbefehle zeichnen ein Dreieck
    LineXY(62, 140, 112, 220, RGB(0, 0, 255))
    LineXY(112, 220, 12, 220, RGB(0, 0, 255))
    LineXY(12, 220, 62, 140, RGB(0, 0, 255))
    FillArea(62, 180, RGB(0, 0, 255), RGB(0, 0, 255))
    StopDrawing()
  EndIf

  Repeat
    Ereignis.i = WaitWindowEvent()
  Until Ereignis = #PB_Event_CloseWindow
EndIf
End
```

Wenn Sie obiges Beispiel starten, wird ein Fenster ähnlich Abb. 33 geöffnet, und ein rotes Quadrat, ein grüner Kreis und ein blaues Dreieck werden auf dieses gezeichnet. Diese Formen werden direkt auf das Fenster gezeichnet, es werden keine Gadgets verwendet. Damit Sie obigen Code verstehen, muss ich zuerst die Syntax der Zeichenbefehle erklären.



(Microsoft Windows Beispiel) Abb. 33

Zuerst sehen Sie, dass ich einen neuen Befehl mit Namen 'StartDrawing()' verwendet habe. Dieser Befehl weist PureBasic an, dass ich mit den 2D Grafik Befehlen zeichnen möchte und worauf ich zeichnen möchte. Wenn Sie auf die Hilfeseite des 'StartDrawing()' Befehls schauen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → 2DDrawing → StartDrawing), können Sie sehen, dass dieser Befehl fünf weitere Befehle als Parameter akzeptiert, die die Ausgabemethode spezifizieren. Das sind:

'WindowOutput'

Verwenden Sie diesen Befehl um direkt auf ein PureBasic Benutzeroberflächen Fenster zu zeichnen. Dieser Befehl benötigt eine PB Nummer als Parameter, um das Fenster auf das Sie zeichnen wollen zu spezifizieren.

'ScreenOutput'

Verwenden Sie diesen Befehl, um direkt auf einen PureBasic Screen zu zeichnen.

'SpriteOutput'
Verwenden Sie diesen Befehl, um auf ein PureBasic Sprite zu zeichnen. Dieser Befehl benötigt die PB Nummer des Sprite, auf das Sie die Zeichenoperationen anwenden wollen.

'ImageOutput()'
Verwenden Sie diesen Befehl, um auf ein geladenes oder im Speicher neu erstelltes Bild zu zeichnen. Diese Methode benötigt als Parameter die PB Nummer des Bildes auf das Sie zeichnen möchten.

'PrinterOutput()'
Verwenden Sie diesen Befehl um die Zeichenbefehle direkt auf den Drucker umzuleiten.

In meinem Beispiel habe ich angegeben, dass ich direkt auf das Fenster zeichnen möchte, wie hier:

```
...
If StartDrawing(WindowOutput(#FENSTER_HAUPT))
...
```

Wenn ich den Ausgabekanal wie hier definiert habe, kann ich jeden beliebigen 2D Zeichenbefehl verwenden, um auf diesen Ausgabekanal zu zeichnen. Mit anderen Worten, alles was ich mit den 2D Zeichenbefehlen zeichne wird nun auf das Fenster gezeichnet, welches ich dem 'WindowOutput()' Befehl als Parameter übergeben habe. Wenn Sie den 'StartDrawing()' Befehl verwenden, müssen Sie immer auch den 'StopDrawing()' Befehl verwenden, um die Zeichenoperation abzuschließen. Dieser Befehl signalisiert dem PureBasic Compiler, dass ich keine 2D Zeichenbefehle mehr verwenden möchte und von nun an wieder normal mit meinem Programm fortfahren möchte. Dieser Befehl ist sehr wichtig, da eventuell Fehler in Ihrem Programm auftreten können, wenn Sie dem Compiler nicht mitteilen dass Sie die Zeichenoperation beendet haben.

Innerhalb des Zeichenblocks habe ich vier 2D Zeichenbefehle verwendet um die eigentlichen Formen auf das Fenster zu zeichnen. Das sind 'Box()', 'Circle()', 'LineXY()' und 'FillArea()'. All diese Befehle werden detailliert in der Hilfe erklärt und sind über Links auf der '2D Drawing' Hilfeseite erreichbar (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → 2DDrawing). Nahezu alle 2D Zeichenbefehle haben ähnliche Parameter. Es gibt üblicherweise Parameter für die 'x' und 'y' Position und manchmal verschiedene Größenparameter. Der letzte Parameter ist üblicherweise die Farbe, die diese Form erhalten soll. Wenn wir zum Beispiel den 'Box()' Befehl betrachten, sehen wir das die ersten beiden Parameter die Position der Form auf dem Ausgabekanal definieren. Der dritte und der vierte Parameter geben die Breite und die Höhe der Form an und der fünfte Parameter definiert die Farbe.

Arbeiten mit Farben
Farben werden in PureBasic über einen einfachen 24 Bit Farbwert spezifiziert. Dieser Wert ist eine große Zahl, die verschiedene Kombinationen von Rot, Grün und Blau repräsentiert. Um den 24 Bit Wert einer Farbe zu erhalten, müssen wir den 'RGB()' Befehl verwenden. Dieser Befehl benötigt drei Parameter, die die individuellen Werte von Rot, Grün und Blau spezifizieren. Jeder dieser Werte hat einen Wertebereich von '0' bis '255'. Auf diese Weise ist es möglich, den 'RGB()' Befehl bis zu 16,7 Millionen Farben als individuelle 24 Bit Werte zurückgeben zu lassen. Im Fenster Zeichen Beispiel, das Sie in Abb. 33 sehen, habe ich den Befehl folgendermaßen verwendet:

```
...
Box(15, 15, 75, 75, RGB(255, 0, 0))
...
```

Hier habe ich den 'RGB()' Befehl innerhalb des 'Box()' Befehls als Parameter verwendet. Wenn Sie etwas genauer auf den Befehl schauen erkennen Sie, dass ich den Maximalwert für den Rot Parameter übergeben habe und die Minimalwerte für die Grün und Blau Parameter. Das bedeutet, dass der 'RGB()' Befehl einen 24 Bit Wert zurückgibt, der ein volles Rot ohne Grün und Blau Anteil enthält. Das gleiche gilt für das Dreieck:

```
...
LineXY(62, 140, 112, 220, RGB(0, 0, 255))
LineXY(112, 220, 12, 220, RGB(0, 0, 255))
LineXY(12, 220, 62, 140, RGB(0, 0, 255))
FillArea(62, 180, RGB(0, 0, 255), RGB(0, 0, 255))
...
```

Diese vier Zeilen sorgen dafür, dass das blaue Dreieck auf das Fenster in Abb. 33 gezeichnet wird. Die ersten Drei sind 'LineXY()' Befehle, die die drei Seiten des Dreiecks zeichnen. Der letzte Befehl ist ein 'FillArea()' Befehl, der einen Punkt in der Mitte des Dreiecks aufnimmt und es von dort mit der zuvor

verwendeten Linienfarbe auffüllt. Sie können sehen, das alle vier Befehle die gleiche Farbe verwenden. Der erste und der zweite Parameter des 'RGB()' Befehls, die Rot und Grün repräsentieren, sind beide '0', während der dritte Parameter, der Blau repräsentiert, den Wert '255' hat. Das bedeutet, das der 'RGB()' Befehl eine 24 Bit Wert zurückgibt, der ein volles Blau ohne Rot- und Grün Anteile zurückgibt. Der grüne Kreis in Abb. 33 weicht ein wenig von den anderen Elementen ab, da er alle drei Farben verwendet, um den gewünschten Farbton zu erzielen, wie hier:

```
...
Circle(140, 125, 45, RGB(35, 158, 70))
...
```

Hier habe ich eine Mischung aus Rot , Grün und Blau verwendet um dem Kreis in einem netten Grasgrün zu färben, anstelle eines vollen, leuchtenden Grün. Farben auf diese Weise in PureBasic zu verwenden macht die Sache einfach und erleichtert dem Anfänger das Verstehen. Unter Verwendung des 'RGB()' Befehls können Sie nahezu jede Farbe erzeugen, die das Menschliche Auge auseinanderhalten kann.

Während der 'RGB()' Befehl die Rot, Grün und Blau Werte zu einem 24 Bit Wert kombinieren kann, kommt vielleicht eines Tages die Zeit, in der Sie einen 24 Bit Wert in seine Einzel Farbbestandteile zerlegen wollen. Das können Sie mit den Befehlen 'Red()', 'Green()', und 'Blue()' erledigen. Schauen Sie auf dieses Beispiel:

```
FarbWert.i = RGB(35, 158, 70)

Debug "Der 24 Bit Farbwert Setzt sich aus folgenden Teilen zusammen: "
Debug "Rot: "  + Str(Red(FarbWert))
Debug "Grün: " + Str(Green(FarbWert))
Debug "Blau: " + Str(Blue(FarbWert))
```

Jeder der Befehle 'Red()', 'Green()', und 'Blue()' benötigt einen 24 Bit Wert als Parameter und gibt dann den korrespondierenden Farbwert zurück. In der ersten Zeile des obigen Beispiels generiere ich einen 24 Bit Farbwert, der unter anderem einen Rot Anteil von 35 enthält. Wenn ich zu einem späteren Zeitpunkt den Rotwert aus dem 24 Bit Wert extrahieren möchte, kann ich diesen 24 Bit Wert an den Befehl 'Red()' übergeben, der dann den Rot Anteil als Wert zurückgibt, in diesem Fall '35'. Die anderen Farbbefehle arbeiten auf die gleiche Weise.

Warum sollten Sie nicht direkt auf Fenster zeichnen?

Das Zeichnen auf Fenster, auch wenn es möglich ist, ist nicht notwendigerweise der richtige Weg um Grafiken anzuzeigen, die mit den 2D Zeichenbefehlen erstellt wurden. Das liegt hauptsächlich an der Methode, wie verschiedene Betriebssysteme die Aktualisierung der Fensteranzeige handhaben. Wenn Sie zum Beispiel unter Microsoft Windows direkt auf ein Fenster zeichnen und danach ein anderes Fenster über das zuvor erstellte bewegen, dann wird die gezeichnete Grafik durch diesen Vorgang weggewischt. Ursache ist die interne Ereignisverarbeitung von Microsoft Windows. Diese bestimmt automatisch ob ein Fenster neugezeichnet wird oder nicht, und manchmal entscheidet sie sich dafür es nicht neu zu zeichnen. Sie können das Betriebssystem zwingen, nach einem derartigen Ereignis das Fenster neu zu zeichnen, allerdings sind hierfür tiefergehende Kenntnisse des jeweiligen Betriebssystem API erforderlich. Eine etwas elegantere Methode um Grafiken anzuzeigen ist das Erstellen eines Bildes im Speicher, auf das Sie dann mit den Zeichenbefehlen zeichnen können. Anschließend zeigen Sie dieses Bild in einem Image Gadget auf Ihrem Fenster an. Durch die Verwendung eines Image Gadget kümmert sich das Fenster automatisch darum ob das Bild neu gezeichnet werden muss. Dadurch ist gewährleistet, dass das Bild immer sichtbar ist wenn das Fenster zu sehen ist.

Zeichnen auf ein Bild

Das Arbeiten mit neuen Bildern in PureBasic ist einfach, da es eine komplette Bibliothek mit Befehlen zum erstellen und manipulieren von Bildern enthält. Im nächsten Beispiel versuche ich das letzte Programm zu emulieren, allerdings mit einem kleinen Unterschied. Ich werde ein neues Bild mit dem 'CreateImage()' Befehl erstellen, anschließend die Formen darauf zeichnen und dann das fertige Bild in einem Image Gadget anzeigen. Das Fenster in diesem Beispiel braucht sich nicht um irgendwelche neu zeichnen Operationen zu kümmern, und sollte sich so verhalten, wie wir es von einem normalen Fenster erwarten. In der PureBasic Hilfedatei können Sie mehr Informationen über die Bild Befehle erhalten, diese finden Sie unter der 'Image' Bibliothek (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Image). Hier unser Beispiel:

```
Enumeration
  #FENSTER_HAUPT
  #BILD_GADGET
  #BILD_HAUPT
EndEnumeration
```

```
If CreateImage(#BILD_HAUPT, 180, 220)
  If StartDrawing(ImageOutput(#BILD_HAUPT))
    ;Da jedes neue Bild einen schwarzen Hintergrund hat, zeichnen wir einen weißen:
    Box(0, 0, 180, 220, RGB(255, 255, 255))
    ;Nun zeichnen wir die Formen:
    Box(5, 5, 75, 75, RGB(255, 0, 0))
    Circle(130, 115, 45, RGB(35, 158, 70))
    LineXY(52, 130, 102, 210, RGB(0, 0, 255))
    LineXY(102, 210, 2, 210, RGB(0, 0, 255))
    LineXY(2, 210, 52, 130, RGB(0, 0, 255))
    FillArea(52, 170, RGB(0, 0, 255), RGB(0, 0, 255))
    StopDrawing()
  EndIf
EndIf

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered

If OpenWindow(#FENSTER_HAUPT, 0, 0, 200, 240, "Zeichnen auf ein neues Bild", #FLAGS)
  ImageGadget(#BILD_GADGET, 10, 10, 180, 220, ImageID(#BILD_HAUPT))
  Repeat
    Ereignis.i = WaitWindowEvent()
  Until Ereignis = #PB_Event_CloseWindow
EndIf
End
```

Der Großteil des Codes sollte selbsterklärend sein, da der einzige Unterschied zum letzten Beispiel darin besteht, dass wir nun mit dem 'CreateImage()' Befehl ein neues Bild im Speicher erstellen und auf dieses anstelle eines Fensters zeichnen. Der 'CreateImage()' Befehl benötigt vier Parameter. Der Erste ist die PB Nummer die mit dem Bild verknüpft ist. Der Zweite und der dritte Parameter definieren die Breite und die Höhe (in Pixel) des neuen Bildes und der vierte Parameter gibt die Farbtiefe des Bildes an. Dieser Parameter ist optional. Wenn er wie in meinem Beispiel nicht verwendet wird, ist die Farbtiefe 24 Bit.

Was ist die Farbtiefe eines Bildes?

Jeder Pixel auf einem Computer Monitor oder in einem digitalen Bild wird durch eine binäre Zahl beschrieben, genau so wie Sie es bei einem Computer erwarten. Je höher die Anzahl der Bits (oder binären Ziffern) die zum beschreiben eines einzelnen Pixels verwendet werden, desto größer ist der Farbbereich der mit diesem Pixel dargestellt werden kann. Die Anzahl der Bits beschreibt immer einen einzelnen Pixel in einem beliebigen digitalen Bild oder auf dem Computer Monitor. Dieser Wert wird als Farbtiefe bezeichnet. Gängige Farbtiefen sind 1 Bit, 8 Bit, 16 Bit, 24 Bit und 32 Bit.

1 Bit Bilder können nur die Farben schwarz und weiß (2 Farben) darstellen, da jedes Bit das einen Pixel beschreibt den Wert '1' oder '0' haben kann.

32 Bit Pixel sind auf der anderen Seite imstande mehr Farben darzustellen, als das menschliche Auge unterscheiden kann, weshalb dieses Format üblicherweise in Bildern für digitale Filme, digitale Fotos, realistische Computerspiele, usw. verwendet wird. Moderne Computer verwenden heutzutage nur noch 32 Bit pro Pixel.

Nachdem das neue Bild erstellt wurde teile ich dem Programm mit, dass ich auf das Bild zeichnen möchte. Hierzu verwende ich folgende Zeile:

```
...
StartDrawing(ImageOutput(#BILD_HAUPT))
...
```

Hier verwende ich den 'StartDrawing()' Befehl mit der 'ImageOutput()' Methode. Damit definiere ich, dass alle folgenden Zeichenbefehle auf dem Bild mit der Nummer '#BILD_HAUPT' ausgeführt werden.

Diese Methode zum zeichnen ist nicht auf neu erstellte Bilder beschränkt. Sie können auch ein Bild laden und auf diese zeichnen. Wenn ein Bild mit einer PB Nummer verknüpft ist, können Sie den Ausgabekanal auf Dieses umleiten und darauf zeichnen.

Nachdem das Bild erstellt wurde und ich den Ausgabekanal zum zeichnen gesetzt habe, kann ich frei auf dieses Bild zeichnen. Da in PureBasic alle neu erstellten Bilder einen schwarzen Hintergrund haben, ändere ich diesen Hintergrund zuerst in weiß. Dies erledige ich mit einem 'Box()' Befehl, der ein weißes Rechteck von der Größe des neu erstellten Bildes zeichnet. Dadurch wird die schwarze Fläche komplett verdeckt. Danach zeichne ich das Quadrat, den Kreis und das Dreieck, wie im vorhergehenden Beispiel. Zum Abschluss der Operation verwende ich den sehr wichtigen 'StopDrawing Befehl'.

Damit das Bild richtig angezeigt wird, und um zu vermeiden das Bild nach einer Überdeckung neu zeichnen zu müssen, verwende ich ein Image Gadget. Dadurch wird das Betriebssystem dazu veranlasst, das Fenster bei Überlappung eigenständig neu zu zeichnen.

Ändern des Zeichenmodus

Im nächsten Beispiel zeige ich Ihnen wie Sie den Zeichenmodus von bestimmten Befehlen ändern können, inklusive Text. Mit dem 'DrawingMode()' Befehl können Sie die Formenbefehle in den Umriss Modus umschalten, Text mit einem transparenten Hintergrund darstellen oder auch die Ausgabe von Zeichenbefehlen mit dem Hintergrund vermischen (XOr). Hier ein passendes Beispiel:

```
Enumeration
  #FENSTER
  #BILD_GADGET
  #BILD_HAUPT
  #SCHRIFT_HAUPT
EndEnumeration

Global BildBreite.i = 401
Global BildHoehe.i  = 201
Global XPos.i, YPos.i, Breite.i, Hoehe.i, Rot.i, Gruen.i, Blau.i
Global Text.s = "PureBasic - 2D Zeichen Beispiel"

Procedure.i Zufallswert(Maximum.i)
  Repeat
    Zahl.i = Random(Maximum)
  Until (Zahl % 10) = 0
  ProcedureReturn Zahl
EndProcedure

If CreateImage(#BILD_HAUPT, BildBreite, BildHoehe)
  If StartDrawing(ImageOutput(#BILD_HAUPT))
    For x.i = 0 To 1500
      XPos   = Zufallswert(BildBreite) + 1
      YPos   = Zufallswert(BildHoehe) + 1
      Breite = (Zufallswert(100) - 1) + 10
      Hoehe  = (Zufallswert(100) - 1) + 10
      Rot    = Random(255)
      Gruen  = Random(255)
      Blau   = Random(255)
      Box(XPos, YPos, Breite, Hoehe, RGB(Rot, Gruen, Blau))
      DrawingMode(#PB_2DDrawing_Outlined)
      Box(XPos - 1, YPos - 1, Breite + 2, Hoehe + 2, RGB(0, 0, 0))
      DrawingMode(#PB_2DDrawing_Default)
    Next x
    LineXY(BildBreite - 1, 0, BildBreite - 1, BildHoehe, RGB(0, 0, 0))
    LineXY(0, BildHoehe - 1, BildBreite, BildHoehe - 1, RGB(0, 0, 0))
    Box(10, 10, 230, 30, RGB(90, 105, 134))
    DrawingMode(#PB_2DDrawing_Outlined)
    Box(10, 10, 231, 31, RGB(0, 0, 0))
    DrawingMode(#PB_2DDrawing_Transparent)
    DrawText(21, 18, Text, RGB(0, 0, 0))
    DrawText(19, 18, Text, RGB(0, 0, 0))
    DrawText(21, 16, Text, RGB(0, 0, 0))
    DrawText(19, 16, Text, RGB(0, 0, 0))
    DrawText(20, 17, Text, RGB(255, 255, 255))
    StopDrawing()
  EndIf
EndIf

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER, 0, 0, BildBreite + 20, BildHoehe + 20, "Abstrakt", #FLAGS)
  ImageGadget(#BILD_GADGET, 10, 10, BildBreite, BildHoehe, ImageID(#BILD_HAUPT))
```

```
    Repeat
      Ereignis.i = WaitWindowEvent()
    Until Ereignis = #PB_Event_CloseWindow
  EndIf
  End
```

Dieses Beispiel ist einfach nur eine erweiterte Version des letzten Beispiels. Auch hier habe ich ein neues (etwas größeres) Bild erstellt, auf das ich zufällig generierte Rechtecke zeichne. Bei jedem Rechteck das ich zeichne schalte ich den Zeichenmodus auf Umriss und zeichne auf das zuvor erstellte Rechteck eine schwarze Außenlinie mit den gleichen Abmessungen. Dadurch wird das Rechteck gut hervorgehoben und hat neben der zufälligen Farbe auch noch einen schwarzen Rahmen. Um zwischen den Zeichenmodi umzuschalten verwenden Sie den 'DrawingMode()' Befehl mit einigen verschiedenen Konstanten, die als Parameter übergeben werden. Hier die verschiedenen Konstanten und was Sie bewirken:

'#PB_2DDrawing_Default'
Das ist der Standardmodus, Text wird mit einer festen Hintergrundfarbe angezeigt und grafische Formen sind gefüllt.

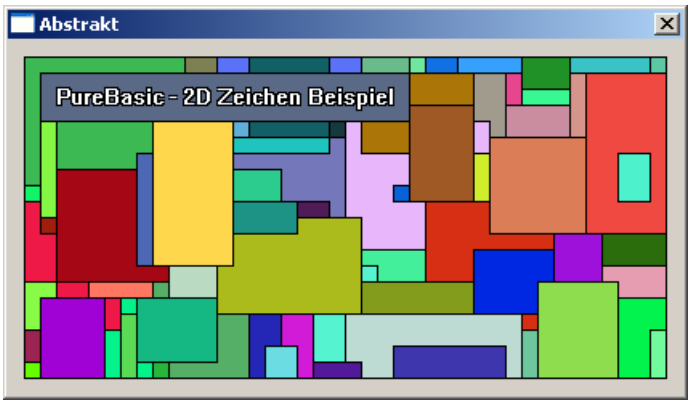
'#PB_2DDrawing_Outlined'
Dieser Parameter schaltet in den Umriss Modus für Formenbefehle wie 'Box()', 'Circle()' und 'Ellipse()'.

'#PB_2DDrawing_Transparent'
Dieser Parameter setzt den Texthintergrund auf Transparent, weshalb nur der Vordergrundfarbe Parameter des 'DrawText()' Befehls verwendet wird.

'#PB_2DDrawing_XOr'
Dieser Parameter schaltet den XOr Modus ein das bedeutet, dass alle erstellten Grafiken mit dem Bildhintergrund XOr verknüpft werden.

Die Verwendung dieses Befehls ist sehr einfach. Wenn Sie den Zeichenmodus innerhalb eines 'StartDrawing()' Blocks ändern möchten, rufen Sie einfach den Befehl mit der Konstante Ihrer Wahl als Parameter auf und der Modus ist sofort geändert. Diese Konstanten können mit dem bitweise 'Or' Operator verknüpft werden. Wenn Sie sich den Code anschauen können Sie sehen, dass ich den Modus häufig gewechselt habe.

Wenn Sie dieses Beispiel starten werden Sie ein Fenster wie in Abb. 34 sehen. Darauf befindet sich ein Bild mit vielen bunten Rechtecken, die alle einen schwarzen Rahmen haben und Sie sehen einen weißen Text in der oberen linken Ecke.



(Microsoft Windows Beispiel) Abb. 34

Zum zeichnen des Textes habe ich den '#PB_2DDrawing_Transparent' Modus verwendet, damit der Text einen transparenten Hintergrund anstelle einer festen Farbe hat. Um den eigentlichen Text zu zeichnen habe ich den 'DrawText()' Befehl verwendet. Dieser benötigt fünf Parameter, die 'x' und 'y' Position auf dem Ausgabekanal, den String, sowie die Werte für die Vordergrund- und die Hintergrundfarbe. Da ich einen transparenten Hintergrund verwende, habe ich den fünften Parameter ausgelassen.

Die Umrisslinie habe ich durch vier Zeichenvorgänge mit schwarz unter Verwendung verschiedener Offsets gezeichnet, anschließend habe ich den Text in Weiß darüber gezeichnet. Dieser Umweg war nötig, da der '#PB_2DDrawing_Outlined' Modus das zeichnen von Text nicht unterstützt.

Zeichnen von Text

Um Text auf einen Ausgabekanal zu zeichnen können Sie den 'DrawText()' Befehl verwenden (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → 2DDrawing → DrawText). Wenn Sie auf das letzte Beispiel schauen, können Sie sehen das ich diesen Befehl einige Male verwendet habe. Der erste und der zweite Parameter ist die 'x' und die 'y' Position des Textes auf dem Ausgabekanal. Der dritte Parameter ist der eigentliche String, der als Text gezeichnet werden soll. Der vierte und fünfte Parameter sind die Vordergrund- und Hintergrundfarbe des Textes. Mit Vordergrundfarbe ist die Farbe des Textes gemeint, die Hintergrundfarbe definiert den Hintergrund des Textes. Die letzten beiden Parameter sind optional, werden diese nicht angegeben, dann werden Standardwerte verwendet. Diese Standardwerte können mit folgenden Befehlen verändert werden: 'FrontColor()' und 'Backcolor()'. Die Parameter dieser Befehle sind 24 Bit Farbwerte, die einfach mit dem 'RGB()' Befehl erstellt werden können.

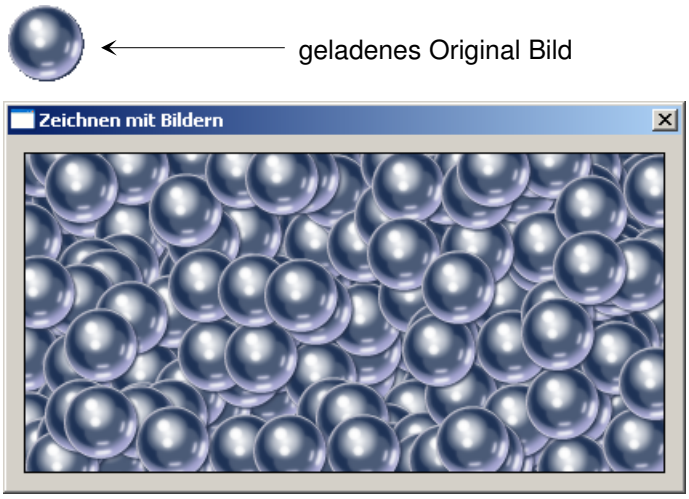
Zeichnen unter Verwendung eines Bildes

in den letzten paar Beispielen habe ich Ihnen gezeigt wie Sie einfache Formen und Linien mit den eingebauten 2D Zeichenbefehlen zeichnen, aber manchmal kann das einschränkend sein. Im nächsten Beispiel zeige ich Ihnen, wie Sie ein externes Bild laden und dieses an jeder beliebigen Position auf ein neu erstelltes Bild zeichnen.

```
Enumeration
  #FENSTER
  #BILD_GADGET
  #BILD_KLEIN
  #BILD_HAUPT
EndEnumeration

Global BildBreite.i = 400
Global BildHoehe.i  = 200
Global XPos.i, YPos.i, BildBreiteGeladen.i, BildHoeheGeladen.i

Global Datei.s
Global RequesterText.s   = "Wählen Sie ein Bild"
Global StandardDatei.s   = ""
Global Muster.s          = "Bitmap (*.bmp)|*.bmp|Icon (*.ico)|*.ico"

Datei = OpenFileRequester(RequesterText, StandardDatei, Muster, 0)

If Datei
  LoadImage(#BILD_KLEIN, Datei)
  BildBreiteGeladen = ImageWidth(#BILD_KLEIN)
  BildHoeheGeladen  = ImageHeight(#BILD_KLEIN)
  If CreateImage(#BILD_HAUPT, BildBreite, BildHoehe)
    If StartDrawing(ImageOutput(#BILD_HAUPT))
      Box(0, 0, BildBreite, BildHoehe, RGB(255, 255, 255))
      For x.l = 1 To 1000
        XPos = Random(BildBreite) - (ImageWidth(#BILD_KLEIN)  / 2)
        YPos = Random(BildHoehe)  - (ImageHeight(#BILD_KLEIN) / 2)
        DrawImage(ImageID(#BILD_KLEIN), XPos, YPos)
      Next x
      DrawingMode(#PB_2DDrawing_Outlined)
      Box(0, 0, BildBreite, BildHoehe, RGB(0, 0, 0))
      StopDrawing()
    EndIf
  EndIf
  #TEXT  = "Zeichnen mit Bildern"
  #FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
  If OpenWindow(#FENSTER, 0, 0, BildBreite + 20, BildHoehe + 20, #TEXT, #FLAGS)
    ImageGadget(#BILD_GADGET, 10, 10, BildBreite, BildHoehe, ImageID(#BILD_HAUPT))
    Repeat
      Ereignis.i = WaitWindowEvent()
    Until Ereignis = #PB_Event_CloseWindow
  EndIf
EndIf
End
```

In obigem Beispiel erlaube ich dem Benutzer ein Bild im Bitmap- oder Icon Format zu öffnen, hierzu verwende ich den OpenFileRequester()' Befehl. Wenn dieser Befehl einen gültigen Bild Dateinamen zurückgibt, lade ich diese Datei mit dem 'LoadImage()' Befehl, sehr ähnlich dem Image Gadget Beispiel aus Kapitel 9. Nachdem das Bild geladen ist, kann ich ein neues Bild zum darauf zeichnen erstellen. Danach

zeichne ich das geladene Bild mit dem 'DrawImage()' Befehl mehrere Male auf das neu erstellte Bild. Das daraus resultierende Bild zeige ich auf einem Fenster in einem Image Gadget an. Wenn Sie auf Abb. 35 schauen können Sie sehen, wie es aussieht wenn ich ein Icon mit einer Kugelform lade und dieses mehrere Male an Zufallspositionen auf das neue Bild zeichne und das daraus resultierende Bild dann anzeige.



(Microsoft Windows Beispiel) Abb. 35

Wenn Sie ein geladenes Bild auf ein neu erstelltes Bild zeichnen wollen verwenden Sie den 'DrawImage()' Befehl. Dieser Befehl benötigt fünf Parameter und ist sehr einfach zu verstehen. Der erste Parameter ist der Betriebssystem Identifikator des Bildes, das Sie zum zeichnen verwenden wollen. Dieser kann mit dem 'ImageID()' Befehl ermittelt werden. Der zweite und dritte Parameter ist die Horizontale und die Vertikale Position des zu bearbeitenden Bildes in Pixel. Die Parameter vier und fünf sind optional und ich habe sie nicht verwendet, sie definieren die Breite und Höhe des geladenen Bildes. Sie können diese Parameter verwenden, um das Bild vor dem zeichnen in der Größe zu ändern. In meinem Beispiel habe ich die meisten der Parameter mit Zufallswerten gefüllt, um das neue Bild mit eintausend Kopien des geladenen Bildes an zufälligen Positionen zu füllen.

Ich denke, das Sie an dieser Stelle das erste mal den 'OpenFileRequester()' Befehl gesehen haben, den ich im folgenden etwas genauer erklären werde. Dieser Befehl öffnet einen Standard Dialog des Betriebssystems, mit dem Sie eine Datei auswählen können. Wenn Sie eine Datei ausgewählt haben, gibt der 'OpenFileRequester()' Befehl den Namen der Datei als String zurück. Dies kann sehr schnell mit einem kleinen Codeschnipsel demonstriert werden:

```
Global Datei.s
Global RequesterText.s = "Wählen Sie ein Bild"
Global StandardDatei.s = ""
Global Muster.s        = "Bitmap (*.bmp)|*.bmp|Icon (*.ico)|*.ico"
Datei = OpenFileRequester(RequesterText, StandardDatei, Muster, 0)
Debug Datei
```

Wenn Sie auf die Hilfeseite des 'OpenFileRequester()' Befehls schauen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Requester → OpenFileRequester) sehen Sie, das dieser Befehl vier Parameter benötigt. Der erste Parameter ist der String, der in der Titelleiste des Dialoges angezeigt wird wenn er geöffnet wird. Der zweite Parameter ist ein String Parameter und dient zum definieren einer Datei, nach der gesucht werden soll. Wenn dieser Parameter definiert ist, wird der Dateiname beim Öffnen des Dialoges im Auswahlfeld angezeigt. Der dritte Parameter ist ein Dateimuster, das es dem Programmierer ermöglicht eine Datei vorselektion durchzuführen. Dadurch werden nur die definierten Dateien im Dialog angezeigt. In diesem Beispiel ist 'Bitmap' als Vorauswahl festgelegt, da der Auswahlmuster Index mit '0' beginnt.

Das Dateiauswahlmuster sieht kompliziert aus, ist aber recht einfach wenn Sie es verstanden haben. Die Dateitypen werden mit dem Weiterleitungszeichen in kleine Abschnitte unterteilt. Das Weiterleitungszeichen '|' ist identisch mit dem bitweise 'Or' Operator. Diese Abschnitte arbeiten paarweise, um die Möglichkeit zu haben, zur Dateiendung einen beschreibenden String anzuzeigen und anschließend die Dateiendung selbst zu definieren. In obigem Beispiel sieht das Dateiauswahl Muster folgendermaßen aus:

```
"Bitmap (*.bmp)|*.bmp|Icon (*.ico)|*.ico"
```

Wenn wir diese Zeile in Abschnitte unterteilen und das Weiterleitungszeichen als Separator benutzen, können wir genauer sehen, was definiert wurde:

```
Bitmap (*.bmp)      *.bmp         Icon (*.ico)        *.ico
```

Der erste Abschnitt ist ein String, der im Dateityp Feld innerhalb des Dialoges angezeigt wird und wenn er angewählt ist, verwendet er den nächsten Abschnitt um dem Dialog mitzuteilen, welche Dateitypen im Hauptfenster angezeigt werden sollen, in diesem Fall alle '.bmp' Dateien. Denken Sie daran, das Asterisk Zeichen ('*') ist ein Platzhalter, der für jeden Namen stehen kann. Der dritte Abschnitt ist ein weiterer String, der im Dateityp Feld angezeigt werden kann und wenn er angewählt ist zeigt der Dialog die Dateien an, die im vierten Abschnitt definiert sind, usw. Wenn Sie mehrere Dateiendungen für einen Dateityp definieren möchten, können Sie das Semikolon verwenden, um mehrere Dateiendungen in einem Abschnitt zu definieren, wie hier:

```
"JPEG|*.jpg;*.jpeg"
```

Dadurch werden alle Dateien mit den Erweiterungen '.jpg' oder '.jpeg' angezeigt, wenn der 'JPEG' String im Dateityp Feld angewählt ist. Denken Sie daran, wenn Sie andere Bildtypen als Bitmaps oder Icons laden wollen, müssen Sie den entsprechenden Decoder für die weiteren Formate in Ihrem Programm laden (Siehe Kapitel 9 - Einbinden von Grafiken in Ihr Programm).

Zeichnen von Bildern mit Alpha Kanal

Manchmal möchten Sie vielleicht ein Bild zeichnen, das einen Alpha Kanal enthält, um z.B. einen Schlagschatten zu erzeugen oder bestimmte Teile des Bildes transparent zu machen. In Abb. 35 habe ich ein Icon verwendet um damit auf das neu erstellte Bild zu zeichnen und standardmäßig behält PureBasic eine im Icon vorgefundene Alpha Kanal Information. Aber was machen Sie, wenn Sie anstelle eines Icons ein Bild im PNG oder TIFF Format verwenden und den Alpha Kanal erhalten wollen? Hier verwenden Sie den 'DrawAlphaImage()' Befehl. Dieser Befehl erhält alle Alpha Informationen, die im Bild gefunden werden. Bildformate die den Alpha Kanal unterstützen sind z.B. 32 Bit PNG oder TIFF Dateien.

Der 'DrawAlphaImage()' Befehl wird exakt auf die gleiche Weise verwendet wie der 'DrawImage()' Befehl, der einzige Unterschied besteht darin, dass der 'DrawAlphaImage()' Befehl nur drei Parameter hat und deshalb die dynamische Größenänderung des Bildes nicht unterstützt. Hier ein Beispiel, wie der Befehl verwendet werden kann:

```
...
DrawAlphaImage(ImageID(#BILD_PNG), XPos, YPos)
...
```

Die obige Zeile wird ein Bild im PNG Format anzeigen und seine Alpha Kanal Informationen erhalten, es also schön mit dem Hintergrund verschmelzen. Sie müssen immer daran denken, wenn Sie andere Bildformate als Bitmap oder Icon verwenden, dass Sie den entsprechenden Decoder Befehl in Ihrem Programm aufrufen müssen, so wie ich es in Kapitel 9 erklärt habe.

Speichern von Bildern

Wenn Sie ein Bild in Ihrem Programm erstellt haben, wollen Sie es vielleicht auf der Festplatte speichern. PureBasic macht dieses Vorhaben sehr einfach, durch die Bereitstellung des 'SaveImage()' Befehls (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Image → SaveImage). Dieser Befehl kann jedes Bild in Ihrem Programm speichern das eine PB Nummer besitzt. Der 'SaveImage()' Befehl benötigt vier Parameter. Der erste Parameter ist die PB Nummer des Bildes das Sie speichern wollen. Der zweite Parameter ist der Dateiname unter dem Sie das Bild speichern wollen. Der dritte Parameter ist optional und definiert das Bildformat in dem das Bild gespeichert werden soll. Der vierte Parameter dient zur Übergabe von optionalen Werten, die abhängig vom verwendeten Bildformat sind.

Wenn wir das Bild das wir im letzten Beispiel erstellt haben speichern wollen, können wir folgende Codezeile am Ende des Programms einfügen. Wenn das Programm endet, wird das Bild gespeichert:

```
...
SaveImage(#BILD_HAUPT, "Bild.bmp")
...
```

Standardmäßig speichert PureBasic das Bild im Bitmap Format, wenn die beiden optionalen Parameter ausgelassen werden. Deshalb muss der Dateiname die richtige Erweiterung haben, wenn die Datei gespeichert wird - in diesem Fall '*.bmp'.

Speichern von Bildern in anderen Formaten
Sie können Bilder in anderen Grafikformaten abspeichern indem Sie den dritten Parameter angeben. Das kann eine dieser vier eingebauten Konstanten sein:

'#PB_ImagePlugin_BMP'
Speichert das Bild im Bitmap Format. Das ist die Standardeinstellung und muss deshalb nicht unbedingt angegeben werden.

'#PB_ImagePlugin_JPEG'
Speichert das Bild im JPEG Format. (Für eine ordnungsgemäße Funktion muss der 'UseJPEGImageEncoder()' Befehl aufgerufen werden, bevor dieser Befehl benutzt wird).

'#PB_ImagePlugin_JPEG2000'
Speichert das Bild im JPEG2000 Format. (Für eine ordnungsgemäße Funktion muss der 'UseJPEG2000ImageEncoder()' Befehl aufgerufen werden, bevor dieser Befehl benutzt wird).

'#PB_ImagePlugin_PNG'
Speichert das Bild im PNG Format. (Für eine ordnungsgemäße Funktion muss der 'UsePNGImageEncoder()' Befehl aufgerufen werden, bevor dieser Befehl benutzt wird).

Wenn Sie Bilder unter Verwendung von '#PB_ImagePlugin_JPEG', '#PB_ImagePlugin_JPEG2000' oder von '#PB_ImagePlugin_PNG' speichern möchten, müssen Sie das dazugehörige Encoder Modul in Ihrem Quelltext aufrufen bevor Sie den 'SaveImage()' Befehl verwenden, damit dieser weiß, wie er das entsprechende Bild Encodieren muss. Der entsprechende Encoder muss nur einmal aufgerufen werden und ist dann im kompletten Programm verfügbar. Hier sind die entsprechenden Encoder:

'UseJPEGImageEncoder()'
Dieser Encoder fügt die Unterstützung für Bilder im JPEG Format hinzu (*.jpg/*.jpeg).

'UseJPEG2000ImageEncoder()'
Dieser Encoder fügt die Unterstützung für Bilder im JPEG2000 Format hinzu (*.jpg/*.jpeg).

'UsePNGImageEncoder()'
Dieser Encoder fügt die Unterstützung für Bilder im PNG Format hinzu (*.png).

Wenn Sie den 'UseJPEGImageEncoder()' oder den 'UseJPEG2000ImageEncoder()' Befehl verwenden, um JPEG (JPEG2000) Unterstützung hinzuzufügen, können Sie den optionalen vierten Parameter des 'SaveImage()' Befehls verwenden, um den Kompressionsgrad für das zu speichernde Bild anzugeben. Das sind die einzigen Bildformate, die zur Zeit den vierten Parameter unterstützen.

Hier ein paar Beispiele für die Verwendung des 'SaveImage()' Befehls:

```
SaveImage(#BILD_HAUPT, "Bild.bmp")
```

Dieses erste Beispiel wird ein Bild mit dem Namen 'Bild.bmp' im Standard 24 Bit Bitmap Format speichern. Beachten Sie, dass kein Encoder benötigt wird, da PureBasic das Bitmap Format als Standard unterstützt.

```
UseJPEGImageEncoder()
SaveImage(#BILD_HAUPT, "Bild.jpg", #PB_ImagePlugin_JPEG)
```

Das zweite Beispiel speichert ein Bild mit Namen 'Bild.jpg' im JPEG Format. Dies geschieht mit dem Standard Kompressionsgrad von '7', da wir im vierten Parameter keinen Kompressionsgrad definiert haben.

```
UseJPEG2000ImageEncoder()
SaveImage(#BILD_HAUPT, "Bild.jpg", #PB_ImagePlugin_JPEG2000, 10)
```

Das dritte Beispiel speichert ein Bild mit Namen 'Bild.jpg' im JPEG2000 Format. Dies geschieht mit dem Kompressionsgrad von '10' (maximale Qualität), den wir im vierten Parameter definiert haben.

```
UsePNGImageEncoder()
SaveImage(#BILD_HAUPT, "Bild.jpg", #PB_ImagePlugin_PNG)
```

Das vierte Beispiel speichert ein Bild mit Namen 'Bild.png' im PNG Format.

Einführung in 'Screens'

Wenn Sie irgendwann einmal ein eigenes Spiel oder einen Bildschirmschoner mit PureBasic programmieren wollen, dann ist das erste was Sie tun einen 'Screen' zu öffnen. Dieser Screen ist eine reine grafische Umgebung, die nur für einen Zweck erstellt wurde, nämlich zum Anzeigen von 2D Zeichnungen, geladenen Bildern, geladenen Sprites und geladenen 3D Modellen sowie 3D Welten.

Was ist ein Sprite?

Sprites waren ursprünglich spezielle, hardwarebeschleunigte Bilder, die dazu verwendet wurden, bewegte 2D Grafiken für Computerspiele zu erstellen. Als die Rechenleistung der Computer mit den Jahren wuchs, wurde die spezielle Hardware überflüssig, die zum schnellen zeichnen und bewegen der Bilder benötigt wurde. Auch heute noch ist der Begriff als Synonym für 2D Bilder die auf einen Screen gezeichnet werden im Gedächtnis, was die Grundlage für Spiele und dergleichen ist.

Heute kann ein Sprite als kleine Grafik (üblicherweise mit einem transparenten Hintergrund) beschrieben werden, die frei auf dem Bildschirm positioniert und gezeichnet werden kann, um Animationen zu simulieren oder um statistische Grafiken anzuzeigen.

Geöffnete Screens nehmen gewöhnlich die komplette Breite und Höhe des Bildschirms ein, wenn es nötig ist, können Sie aber auch einen Screen in einem bereits erstellten Fenster öffnen. Dieser wird dann als 'Windowed Screen' bezeichnet. Es kann immer nur ein Screen gleichzeitig geöffnet sein, entweder ein 'Full Screen' oder ein 'Windowed Screen'.

Der Grund dafür, dass Screens bevorzugt zum Anzeigen von Grafiken verwendet werden (anstatt diese einfach auf ein Fenster zu zeichnen) liegt in der Geschwindigkeit, Screens sind schnell, ... sehr schnell! Auf allen Plattformen die PureBasic unterstützen sind die Screens darauf optimiert, die größtmögliche Geschwindigkeit aufzubringen, unabhängig vom zugrundeliegenden Betriebssystem.

Das Öffnen Ihres ersten Screens

Um in PureBasic einen Screen zu erstellen und zu öffnen müssen Sie einer grundlegenden Codevorlage folgen. Die Handhabung von Screens und Sprites ist sehr eng miteinander verknüpft, weshalb Sie immer zuerst die Sprite Engine initialisieren müssen, bevor Sie einen Screen öffnen. Das stellt sicher, dass intern alles korrekt initialisiert ist und der Screen bereit ist für Zeichenoperationen. Nachdem die Sprite Engine initialisiert ist und der eigentliche Screen geöffnet ist, benötigen Sie eine Hauptschleife um das Programm am laufen zu halten, so wie in einem Grafischen Benutzeroberflächen Programm.

Der Code unten zeigt ein Grundgerüst für ein Screen Programm mit allen entscheidenden Teilen, um die Sprite Engine zu initialisieren, einen Screen zu öffnen, eine Hauptschleife zu initialisieren und die Initialisierung der Tastatur um eine Möglichkeit zum beenden des Programms zur Verfügung zu stellen.

```
Global Beenden.i = #False

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
Fehlerbehandlung(OpenScreen(1024, 768, 32, "Screen"), "Kann Screen nicht öffnen.")

Repeat
  ClearScreen(RGB(0, 0, 0))

  ;Zeichenoperationen erfolgen hier

  FlipBuffers()
  ExamineKeyboard()
```

```
    If KeyboardReleased(#PB_Key_Escape)
      Beenden = #True
    EndIf
  Until Beenden = #True
  End
```

Wenn Sie zuerst einen Blick auf den Sprite und Screen Bereich in der Hilfe werfen (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite & Screen), finden Sie mehr Informationen über die neuen Befehle, die in diesem Beispiel verwendet wurden.

Dieses Beispiel macht eigentlich nichts, außer das es einen blanken Screen öffnet. Wenn Sie diesen Screen sehen, können Sie das Programm durch einen Druck auf die 'ESC' Taste beenden. Dieses Beispiel ist der Ursprung für Spiele und dergleichen, deshalb lassen Sie mich erklären was in den wichtigen Teilen passiert.

Zu Beginn habe ich eine kleine Fehlerprüfprozedur erstellt, wie ich sie in Kapitel 8 (Minimieren von Fehlern und deren Behandlung) erklärt habe und überprüfe damit, dass die Befehle 'InitSprite()', 'InitKeyboard()' und 'OpenScreen()' nicht den Wert '0' zurückgeben. Wenn einer dieser Befehle den Wert '0' zurückgibt, dann ist die Ausführung dieses bestimmten Befehls fehlgeschlagen, worauf ich unmittelbar das Programm beende. Wenn ein Fehler wie dieser auftritt, ist es immer das Beste den Benutzer über das Problem zu informieren und dann das Programm zu beenden. Wenn Sie ihr Programm mit einem Fehler weiterlaufen lassen, können enorme Programmabstürze bis zum Systemabsturz auftreten.

Die Befehle werden benötigt, um die Sprite Engine und die Tastatur zu initialisieren, es handelt sich dabei um einfache Befehlsaufrufe, nämlich 'InitSprite()' und 'InitKeyboard(). Denken Sie daran, das wir zuerst die Sprite Engine initialisieren müssen, bevor wir einen Screen öffnen. Der 'OpenScreen()' Befehl benötigt vier Parameter. Es handelt sich hierbei um Breite, Höhe und Farbtiefe des Screens, als vierter Parameter wird noch ein Beschreibungstext übergeben. Dieser Beschreibungstext ist in der Taskleiste zu lesen, wenn Sie den Screen minimieren. Mit den optionalen Parametern Fünf und Sechs lassen sich die Synchronisation beim umschalten der Bildpuffer sowie die Bildwiederholrate beeinflussen. Parameter Fünf wird im nächsten Abschnitt etwas genauer beschrieben. Mehr Informationen zu diesem Befehl finden Sie in der Hilfe.

Verwenden der Tastatur

In diesem Beispiel habe ich ein paar Tastatur Befehle verwendet, um die Tastatur zu initialisieren und um Tastendrücke zu erkennen, usw. Diese Befehle finden Sie auf der Hilfeseite der 'Keyboard' Bibliothek (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Keyboard). Es handelt sich hierbei um eine sehr kleine und einfach zu handhabende Bibliothek, weshalb Sie die Befehle schnell aufnehmen sollten.

Der 'InitKeyboard()' Befehl initialisiert die Tastatur und muss vor allen anderen Tastaturbefehlen aufgerufen werden. Der 'ExamineKeyboard()' Befehl prüft den aktuellen Status der Tastatur, um zu sehen ob irgendwelche Tasten gedrückt wurden, usw. Die Befehle 'KeyboardPushed()' und 'KeyboardReleased()' geben True zurück, wenn die übergebene Taste entweder gedrückt oder losgelassen wird. Die Tastenwerte können durch eingebaute Konstanten zugewiesen werden. Eine vollständige Auflistung der möglichen Konstanten finden Sie in der Hilfe zu diesen beiden Befehlen.

Die Breite, Höhe und Farbtiefe Parameter sind sehr wichtig, da sie definieren welche Größe und welche Farbtiefe Ihr Screen haben wird. Der Computer der diesen Screen öffnen soll muss die verwendete Größe und Farbtiefe unterstützen. Die Werte für Breite und Höhe werden im allgemeinen als Auflösung bezeichnet, und diese für den Screen gewählte Auflösung muss vollständig von der Grafikkarte und dem daran angeschlossenen Monitor unterstützt werden. Ist diese Unterstützung nicht gewährleistet, tritt ein Fehler im 'OpenScreen()' Befehl auf und der Screen wird nicht geöffnet. Die von mir in diesem Beispiel verwendeten Werte werden von den meisten Computern unterstützt und sollten keine Probleme bereiten. Diese Tatsache müssen Sie sich immer vor Augen halten wenn Sie ein Programm schreiben, das auf einen Screen zurückgreift.

Abb. 36 zeigt eine Auflistung von gebräuchlichen Auflösungen und Farbtiefen, die auf jedem modernen Computer funktionieren sollten. Wie bei Computerspielen gilt auch hier: Je höher die Auflösung, desto schärfer das Bild. Da allerdings bei einer höheren Auflösung mehr Pixel zu zeichnen sind, wird das Programm langsamer. Je höher die Farbtiefe desto mehr Farben können angezeigt werden, wodurch die angezeigten Grafiken realistischer werden.

Gängige Screen Auflösungen

Breite	Höhe	Farbtiefe
640	480	8, 16 und 32 Bit
800	600	8, 16 und 32 Bit
1024	768	8, 16 und 32 Bit

Abb. 36

Anstelle der festen Programmierung von Werten wie diesen in Ihr Programm, sollten Sie bevorzugt den Zielcomputer auf mögliche Auflösungen überprüfen und diese ermittelten Werte verwenden. Die unterstützten Auflösungen und Farbtiefen eines Computers können Sie mit folgendem Code ermitteln:

```
InitSprite()
If ExamineScreenModes()
  While NextScreenMode()
    Breite.i    = ScreenModeWidth()
    Hoehe.i     = ScreenModeHeight()
    Farbtiefe.i = ScreenModeDepth()
    Debug Str(Breite) + " x " + Str(Hoehe) + " x " + Str(Farbtiefe)
  Wend
EndIf
```

Auch in diesem Beispiel initialisieren wir zuerst die Sprite Engine bevor wir den 'ExamineScreenModes()' Befehl verwenden, dadurch wird alles korrekt initialisiert und wir können diese Screen basierenden Befehle verwenden. Dann verwende ich eine 'While' Schleife, um die unterstützten Screen Modi aufzulisten. Hierzu verwende ich die Befehle 'ScreenModeWidth()', 'ScreenModeHeight()' und 'ScreenModeDepth()' um aus den zurückgegebenen Werten einen String zu konstruieren, der den aktuell ermittelten Screen Modus beschreibt. Diesen String gebe ich dann zum überprüfen im Debug Ausgabefenster aus. Das ist ein einfaches Beispiel, das Sie unter Verwendung der Hilfe (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite & Screen) gut nachvollziehen können.

Doppelt gepuffertes Rendern

Wenn Sie einen Screen verwenden um Grafiken anzuzeigen, müssen Sie immer eine Hauptschleife erstellen, um das Programm am laufen zu halten, Grafiken zu zeichnen und Benutzereingaben abzufragen, usw. Auch diese muss bei der Verwendung eines Screens einem bestimmten Muster folgen. Hier ist die Hauptschleife aus dem Screen Grundgerüst Beispiel:

```
...
Repeat
  ClearScreen(0)
  ;Zeichenoperationen erfolgen hier
  FlipBuffers()
  ExamineKeyboard()
  If KeyboardReleased(#PB_Key_Escape)
    Beenden = #True
  EndIf
Until Beenden = #True
...
```

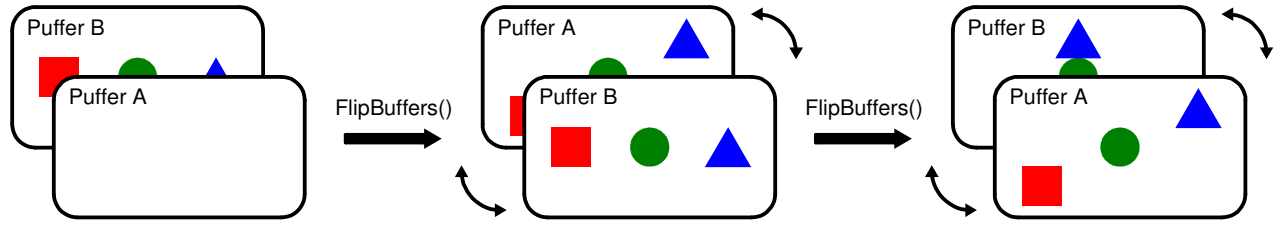
Auf den ersten Blick sieht diese Hauptschleife einer normalen Hauptschleife aus einem anderen Programm sehr ähnlich, aber es gibt ein paar Unterschiede. Es gibt zwei entscheidende Befehle die dafür sorgen das eine Grafik korrekt auf den Screen gezeichnet wird. Dies sind die Befehle 'ClearScreen()' und 'FlipBuffers()'. Bevor ich diese Befehle detaillierter erläutere, muss ich Ihnen eine grafische Technik mit Namen 'Double Buffering' erklären.

'Double Buffering' ist eine Technik, die von PureBasic Screens verwendet wird, um beschädigte Grafiken sowie Bildschirm flimmern zu vermeiden. Da Computer Bildschirme konstant das Bild aktualisieren (üblicherweise sechzig bis siebzig Mal in der Sekunde - manchmal auch mehr), ist es schwierig Änderungen auf einem Screen durchzuführen, wie zum Beispiel neue Grafiken zu zeichnen und zu bewegen, ohne dass der Bildschirm die Änderung vor der Fertigstellung anzeigt. Das Ergebnis sind zerstörte, unschöne Grafiken und andere fremdartige visuelle Artefakte. Wenn Sie dieses Problem vermeiden wollen, indem Sie vor jedem

neuzeichnen den Screen löschen, wird dies das Problem der zerstörten Grafiken beseitigen, allerdings fängt das Bild an zu flimmern.

PureBasic löst dieses Problem durch die Verwendung von Double Buffering. Wenn ein Screen geöffnet wird, werden ihm automatisch zwei Videopuffer im Speicher zugewiesen, die beide die gleiche Größe wie der geöffnete Screen haben. Wenn Sie Grafiken unter Verwendung der PureBasic Befehle auf einen Screen zeichnen, geschieht dies immer auf dem 'Back Buffer' (das Bild im Hintergrund), während das Bild im 'Front Buffer' auf dem Bildschirm angezeigt wird. Wenn der Monitor das Bild fertig angezeigt hat und das Bilde im 'Back Buffer' fertiggestellt ist, dann werden die beiden Puffer 'geflipped' (ausgetauscht). Das Vordergrundbild wird in den Hintergrund gestellt und das ehemalige Hintergrundbild wird auf dem Monitor angezeigt. Wenn das Programm erneut eine Grafik zeichnet, erfolgt dies auf dem neuen 'Back Buffer'. Dieser neue 'Back Buffer' enthält eine alte Grafik, nämlich die zuvor angezeigte. Deshalb ist es Standard, den 'Back Buffer' vor der Erstellung einer neuen Grafik mit einer einzelnen Farbe zu überschreiben (üblicherweise schwarz). Abb. 37 zeigt diese Technik in Aktion, Sie können drei Zyklen des Puffers erkennen.

Double Buffered Rendering



1

Der Screen ist geöffnet und der Zeichenvorgang beginnt auf dem Back Buffer.
(Puffer A wird auf dem Bildschirm angezeigt)

2

Die Puffer werden ausgetauscht.
Der neue Back Buffer wird gelöscht und mit der aktualisierten Grafik versehen.
(Puffer B wird auf dem Bildschirm angezeigt)

3

Die Puffer werden erneut getauscht.
Der neue Back Buffer wird gelöscht und mit der aktualisierten Grafik versehen.
(Puffer A wird auf dem Bildschirm angezeigt)

Abb. 37

In dieser Abbildung können Sie sehen, das der 'Front Buffer' ('Puffer A') beim Programmstart leer ist (indiziert mit Schritt 1), weshalb nichts auf dem Bildschirm angezeigt wird. Wenn wir Grafiken mit den PureBasic Befehlen zeichnen, geschieht die immer auf dem 'Back Buffer' (in diesem Fall 'Puffer B'). Nachdem alle Grafiken gezeichnet sind können wir den 'FlipBuffers()' Befehl aufrufen und die Puffer werden getauscht. Das wird in Schritt 2 dargestellt. Nun wird 'Puffer B' auf dem Bildschirm dargestellt und wir zeichnen auf 'Puffer A' weiter, dem neuen 'Back Buffer'. Nachdem auch hier alle Zeichenoperationen abgeschlossen sind, können wir erneut den 'FlipBuffers()' Befehl aufrufen und erreichen damit Schritt 3, in dem 'Puffer A' auf dem Bildschirm angezeigt wird und 'Puffer B' erneut zum 'Back Buffer' wird.

Wenn die Puffer mittels des 'FlipBuffers()' Befehl ausgetauscht werden, enthält der Puffer der im Hintergrund eintrifft noch alte Grafiken. Wenn wir die Illusion einer Bewegung erzeugen möchten, müssen wir zuerst den Puffer löschen und dann die Objekte an eine neue Position zeichnen bevor wir die Puffer erneut tauschen. Um die alten Grafiken zu löschen verwenden wir den 'ClearScreen()' Befehl. 'ClearScreen()' benötigt einen Parameter, es handelt sich hierbei um einen 24 Bit Farbwert, mit dem wir den Screen überschreiben wollen. Dieser Farbwert kann einfach mit dem 'RBG()' Befehl erzeugt werden.

Kommen wir noch einmal auf den 'OpenScreen()' Befehl zurück. Der im vorherigen Abschnitt erwähnte fünfte Parameter beeinflusst das verhalten des Austauschs der Puffer mit dem 'FlipBuffers()' Befehl und kann diese drei Werte annehmen:

'0' Schaltet die Vertikal Synchronisation des Bildschirms ab
'1' Schaltet die Vertikal Synchronisation des Bildschirms ein (Standard)
'2' Schaltet die Vertikal Synchronisation des Bildschirms ein und verwendet hierzu eine CPU schonende Methode (Funktioniert nur im 'Full Screen' Modus)

Wenn '0' als Parameter verwendet wird, dann tauscht der 'FlipBuffers()' Befehl die Puffer so schnell aus, wie er kann und erreicht dadurch die höchste Bildwiederholrate. Der einzige Nachteil besteht darin, dass eventuell die Bildwiederholrate des Bildschirms nicht schnell genug ist die neue Grafik in Echtzeit anzuzeigen, weshalb zum Teil Bildabrisse auftreten können, während der Monitor versucht die getauschten Puffer anzuzeigen.

Wenn '1' als Parameter verwendet wird tauscht der 'FlipBuffers()' Befehl die Puffer synchron mit der Bildwiederholrate des Monitors um sicherzustellen, dass alle Grafiken sauber auf dem Bildschirm angezeigt werden. Der Nachteil dieser Methode ist die Tatsache, dass die Bildwiederholrate des Programms niemals über der Bildwiederholrate des Monitors liegen kann. Dieser Modus wird standardmäßig verwendet wenn kein Parameter angegeben wird.

Die Verwendung von '2' als Parameter hat den gleichen Effekt wie die Verwendung von '1', schaltet aber in einen CPU schonenden Modus, der dafür sorgt das die CPU nicht zu einhundert Prozent ausgelastet ist. Dadurch haben andere Programme ebenfalls die Möglichkeit die CPU zu nutzen.

Zeichnen auf einen Screen

Hier ist ein Beispiel für 'Double Buffered Rendering' und dem erzeugen einer Animation, die durch zeichnen von aktualisierten Grafiken zwischen jedem 'FlipBuffers()' Befehl entsteht (Beenden Sie das Programm durch einen Druck auf die 'ESC' Taste):

```
#BILD_HAUPT = 1
;Setzen der Breite, Höhe und Farbtiefe des Screens
;Aufgrund der Seitenbreite wurden hier abgekürzte Variablennamen verwendet :(
Global ScrB.l    = 1024
Global ScrH.l    = 768
Global ScrT.l    = 32
Global Beenden.i = #False
XUrsprung.f      = (ScrB / 2) - 64 : YUrsprung.f = (ScrH / 2) - 64

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Initialisiere Umgebung
Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
Fehlerbehandlung(OpenScreen(ScrB,ScrH,ScrT,"Kleckse",2),"Kann Screen nicht öffnen.")
SetFrameRate(60)

;Erstelle ein Bild
If CreateImage(#BILD_HAUPT, 128, 128)
  If StartDrawing(ImageOutput(#BILD_HAUPT))
    For x.i = 255 To 0 Step - 1
      Circle(64, 64, x / 4, RGB(0, 0, 255 - x))
    Next x
    StopDrawing()
  EndIf
EndIf

;Konvertiere Grad in Bogenmaß
Procedure.f GradZuRad(Winkel.f)
  ProcedureReturn Winkel.f * #PI / 180
EndProcedure

;Hauptschleife
Repeat
  ClearScreen(RGB(0, 0, 0))
  Winkel.f + 2.0
  Radius.f = ((ScrH / 2) - 100) * Sin(GradZuRad(Winkel))

  StartDrawing(ScreenOutput())
  For x.i = 0 To 359 Step 45
    XPos.f = XUrsprung + (Radius * Cos(GradZuRad(Winkel + x)))
    YPos.f = YUrsprung + (Radius * Sin(GradZuRad(Winkel + x)))
    DrawImage(ImageID(#BILD_HAUPT), XPos, YPos)
  Next x
  StopDrawing()

  FlipBuffers()
  ExamineKeyboard()
```

```
  If KeyboardReleased(#PB_Key_Escape)
    Beenden = #True
  EndIf
Until Beenden = #True
End
```

In diesem Beispiel habe ich mit dem 'CreateImage()' Befehl ein neues Bild erstellt und dieses Bild dann später mit einem 'StartDrawing()' Block auf den Screen gezeichnet. Auch wenn dieses Beispiel etwas kompliziert aussieht, besonders beim berechnen der 'x' und 'y' Werte des gezeichneten Bildes, dient es doch ausschließlich der Demonstration des Tauschens der Puffer.

In der Hauptschleife können Sie sehen, das ich zuerst den Screen lösche. Hierzu verwende ich den 'ClearScreen()' Befehl. Dadurch kann ich meinen Zeichenvorgang auf einem leeren Puffer beginnen und habe somit keine Überreste von vorherigen Zeichenoperationen zu erwarten. Danach verwende ich ein wenig Mathematik um die Koordinaten für meine Klecks Bilder zu berechnen und verwende eine Schleife um diese zu zeichnen. Nach dem beenden der Zeichenoperation befindet sich die Zeichnung auf dem Back Buffer, deshalb muss ich den 'FlipBuffers()' Befehl verwenden, um die Zeichnung auf dem Bildschirm anzuzeigen. Und so geht es immer weiter, löschen, zeichnen, tauschen, und zwischen jedem Tauschvorgang ändere ich die Position der Zeichnung.

Sie haben vermutlich einen weiteren neuen Befehl bemerkt, den ich in diesem Beispiel verwendet habe - 'SetFrameRate()' (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite & Screen → SetFrameRate). Dieser Befehl benötigt einen Parameter, der die Anzahl der maximalen Ausführungen pro Sekunde von 'FlipBuffers()' definiert. Dadurch lässt sich ein Standard Bildwiederholrate definieren, die auch auf einem anderen Computer beibehalten wird, wenn das Programm auf diesem ausgeführt wird. In diesem Beispiel ist die Bildwiederholrate auf sechzig Aktualisierungen pro Sekunde beschränkt.

Auf diese Weise werden alle Animationen auf einem Computer erstellt, sehr ähnlich dem Daumenkino oder dem Film. Es bewegt sich eigentlich nichts auf dem Bildschirm, es ist nur eine Diashow von verschiedenen Bildern (Puffern), die Grafiken an geringfügig unterschiedlichen Positionen enthalten. Da dies alles sehr schnell passiert (sechzig Mal pro Sekunde, und mehr) sieht es so aus als würden sich die Bilder bewegen.

Ein einfaches Sternenfeld

Das ist einer der Effekte, die jeder Demo oder Spiele Programmierer beherrschen muss. Ein Sternenfeld wie dieses wurde in Hunderten von Demos und Spielen verwendet, um das Gefühl zu vermitteln, durch den Weltraum zu reisen. Es ist ein Effekt, der Hunderte von Pixeln auf den Bildschirm zeichnet und diese animiert. Hierzu werden verschiedene Farbschattierungen verwendet, um die Illusion von Raumtiefe und Bewegung zu erzeugen. Es gibt verschiedene Möglichkeiten diesen Effekt zu erzeugen, hier ist mein Beispiel:

```
#PRG_NAME        = "Sterne v1.0"
#ANZAHL_STERNE   = 10000
;Setzen der Breite, Höhe und Farbtiefe des Screens
;Aufgrund der Seitenbreite wurden hier abgekürzte Variablennamen verwendet :(
Global ScrB.i    = 1024
Global ScrH.i    = 768
Global ScrT.i    = 32
Global Beenden.i = #False

Structure STERNE
  xPos.f
  yPos.f
  xSchritt.f
  Farbe.l
EndStructure
Global Dim Sterne.STERNE(#ANZAHL_STERNE)

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Initialisiere Sterne
Procedure InitialisiereSterne()
```

```
  For x = 0 To #ANZAHL_STERNE
    Sterne(x)\xPos = Random(ScrB - 1)
    Sterne(x)\yPos = Random(ScrH - 1)
    If x < #ANZAHL_STERNE / 3
      Sterne(x)\xSchritt = (Random(10) / 100) + 0.2
      Sterne(x)\Farbe    = RGB(40, 40, 40)
    ElseIf x >= #ANZAHL_STERNE / 3 And x < (#ANZAHL_STERNE / 3) * 2
      Sterne(x)\xSchritt = (Random(10) / 100) + 0.6
      Sterne(x)\Farbe    = RGB(100, 100, 100)
    Else
      Sterne(x)\xSchritt = (Random(10) / 100) + 1.2
      Sterne(x)\Farbe    = RGB(255, 255, 255)
    EndIf
  Next x
EndProcedure

;Bewege Sterne auf der 'x' Achse
Procedure BewegeSterneX()
  For x = 0 To #ANZAHL_STERNE
    Sterne(x)\xPos - Sterne(x)\xSchritt
    If Sterne(x)\xPos < 0
      Sterne(x)\xPos = ScrB - 1
      Sterne(x)\yPos = Random(ScrH - 1)
    EndIf
  Next x
EndProcedure

;Initialisiere Umgebung
Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
Fehlerbehandlung(OpenScreen(ScrB,ScrH,ScrT,#PRG_NAME,2),"Kann Screen nicht öffnen")
SetFrameRate(60)
InitialisiereSterne()

Repeat
  ClearScreen(RGB(0, 0, 0))
  StartDrawing(ScreenOutput())
  For x = 0 To #ANZAHL_STERNE
    Plot(Sterne(x)\xPos, Sterne(x)\yPos, Sterne(x)\Farbe)
  Next x
  DrawingMode(#PB_2DDrawing_Transparent)
  DrawText(20, 20, #PRG_NAME, #White)
  DrawText(20, 40, Str(#ANZAHL_STERNE) + " animierte Sterne", #White)
  DrawText(20, 60, "Screen Auflösung: " + Str(ScrB) + " x " + Str(ScrH), #White)
  DrawText(20, 80, "Screen Farbtiefe: " + Str(ScrT) + " Bit", #White)
  StopDrawing()
  FlipBuffers()
  BewegeSterneX()

  ExamineKeyboard()
  If KeyboardReleased(#PB_Key_Escape)
    Beenden = #True
  EndIf

Until Beenden = #True
End
```

Dieses Beispiel verwendet den 2D Zeichenbefehl 'Plot()', der bei jedem Aufruf einen einzelnen Pixel zeichnet (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → 2DDrawing → Plot). Dieser Befehl benötigt drei Parameter, wobei der Dritte optional ist. Diese Parameter definieren die 'x' und 'y' Position sowie die Farbe des gezeichneten Pixels. Wenn der letzte Parameter nicht verwendet wird, dann verwendet der Befehl die Standard Vordergrundfarbe, die mit dem 'FrontColor()' Befehl definiert werden kann.

In obigem Beispiel habe ich eine Struktur verwendet, um alle Informationen für jeden einzelnen Pixel zusammenzuhalten, dann habe ich ein Array von Variablen erstellt, die diese Struktur als Typ verwenden. Jedes dieser Elemente im Array enthält die Position, den Farbwert sowie den Schrittwert eines Pixels. Dann bewege ich mich in einer Schleife durch das Array und zeichne jeden Pixel auf den Screen, indem ich die Angaben aus dem jeweiligen Array Element verarbeite (Position, Farbe, usw.). Wenn die Zeichnung fertig ist, tausche ich die Puffer aus und aktualisiere die Pixel Positionen im Array unter Verwendung des

zugewiesenen Schrittwertes. Wenn das erledigt ist, lösche ich den Puffer und zeichne die Pixel erneut, ... und das ganze geht immer so weiter. Durch diese Vorgehensweise sieht der Quelltext sauberer aus, ist leichter zu lesen und erlaubt es Ihnen den Code zu einem späteren Zeitpunkt einfach zu aktualisieren.

Öffnen eines Screen in einem Fenster

Manchmal möchten Sie vielleicht einen Screen in einem Fenster öffnen, besonders wenn Sie ein Fensterbasiertes Spiel oder eine Demo erstellen wollen. Das können Sie mit dem 'OpenWindowedScreen()' Befehl erreichen. Um einen Screen in einem Fenster erstellen zu können, müssen Sie zuerst ein Fenster erstellen und die Ereignisse des Fensters in der Hauptschleife auswerten, genauso wie das zeichnen. Hier ein Beispiel, das einen 'Windowed Screen' verwendet:

```
#FENSTER_HAUPT = 1
#BILD_HAUPT    = 1

;Setzen der Breite, Höhe und Farbtiefe des Screens
;Aufgrund der Seitenbreite wurden hier abgekürzte Variablennamen verwendet :(
Global ScrB.i      = 800
Global ScrH.i      = 600
Global ScrT.i      = 32
Global Beenden.i   = #False
Global XUrsprung.f = (ScrB / 2) - 64
Global YUrsprung.f = (ScrH / 2) - 64

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Konvertiere Grad in Bogenmaß
Procedure.f GradZuRad(Winkel.f)
  ProcedureReturn Winkel.f * #PI / 180
EndProcedure

;Initialisiere Umgebung
Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER_HAUPT, 0, 0, ScrB, ScrH, "Screen im Fenster", #FLAGS)
  If OpenWindowedScreen(WindowID(#FENSTER_HAUPT), 0, 0, ScrB, ScrH, 0, 0, 0)
    SetFrameRate(60)

    ;Erstelle ein Bild
    If CreateImage(#BILD_HAUPT, 128, 128)
      If StartDrawing(ImageOutput(#BILD_HAUPT))
        For x.i = 255 To 0 Step - 1
          Circle(64, 64, x / 4, RGB(255 - x, 0, 0))
        Next x
        StopDrawing()
      EndIf
    EndIf

    ;Hauptschleife
    Repeat
      Ereignis.i = WindowEvent()
      ClearScreen(RGB(0, 0, 0))

      Winkel.f + 2.0
      Radius.f = ((ScrH / 2) - 100) * Sin(GradZuRad(Winkel))

      StartDrawing(ScreenOutput())
      For x = 0 To 359 Step 45
        XPos.f = XUrsprung + (Radius * Cos(GradZuRad(Winkel + x)))
        YPos.f = YUrsprung + (Radius * Sin(GradZuRad(Winkel + x)))
        DrawImage(ImageID(#BILD_HAUPT), XPos, YPos)
      Next x
      StopDrawing()
```

```
        FlipBuffers()
        ExamineKeyboard()
        If KeyboardReleased(#PB_Key_Escape)
          Beenden = #True
        EndIf

      Until Ereignis = #PB_Event_CloseWindow Or Beenden = #True
    EndIf
  EndIf
  End
```

Dieses Beispiel sollte einfach zu verstehen sein, da Sie das meiste des Codes schon vorher gesehen haben. Der Hauptunterschied ist der 'OpenWindowedScreen()' Befehl, der neun Parameter benötigt! Der erste Parameter ist der Betriebssystem Identifikator des Fensters, auf dem der Screen geöffnet werden soll. Der zweite und dritte Parameter ist die 'x' und 'y' Position des Screens auf dem Fenster. Der vierte und fünfte Parameter gibt die Breite und die Höhe des neuen Screen an. Der sechste Parameter ist das automatische Größenänderungs Flag. Wenn dieser Parameter auf '0' gesetzt wird, findet keine automatische Größenanpassung statt, wenn dieser Parameter allerdings auf '1' gesetzt wird, wird der Screen sich automatisch der maximalen Fenstergröße anpassen, unabhängig von den Parametern vier und fünf. Das bedeutet, das der Screen bei Gößenänderung des Fensters automatisch seine Größe anpasst, um den gesamten verfügbaren Platz im Fenster einzunehmen. Der siebte und achte Parameter kann definiert werden, wenn Sie die automatische Größenanpassung aktiviert haben. Mit diesen beiden Parametern können Sie einen Offset definieren, wie weit der Screen vom rechten- und vom unteren Rand entfernt beginnen soll, um zum Beispiel Platz für eine Statusleiste oder für Gadgets zu lassen. Der neunte, optionale Parameter definiert wieder den Flip Modus.

Es ist sehr wichtig, das Sie bei der Verwendung eines Screen in einem Fenster die Ereignisse mit dem 'WindowEvent()' Befehl auswerten. Wenn Sie an Kapitel 9 denken, wissen Sie das dieser Befehl nicht auf ein Ereignis wartet, bevor er ein Ergebnis zurückgibt. Stattdessen versucht er immer Ereignisse zu erkennen und gibt alle zurück, die behandelt werden müssen. Die Verwendung dieses Befehls anstelle von 'WaitWindowEvent()' ist Voraussetzung für das Arbeiten mit Windowed Screens, da Sie das Austauschen der Puffer nicht über einen längeren Zeitraum verzögern dürfen.

Sprites

Sprites sind in PureBasic Bilder, die an jeder beliebigen Position auf den Screen gezeichnet werden können. Anders als Bilder haben Sprites ihren eigenen Befehlssatz, der auf Geschwindigkeit optimiert ist. Durch die Verwendung dieser Befehle ist es möglich, Sprites auf spezielle Art und Weise auf den Screen zu zeichnen und somit besondere Effekte zu erzielen. Alle normalen Sprite Befehle befinden sich in der 'Sprite & Screen' Sektion in der PureBasic Hilfe (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite & Screen), es gibt aber noch weitere Befehle in der 'Sprite3D' Sektion (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite3D). Damit Sie sich selbst ein Bild davon machen können, was mit den Sprite Befehlen möglich ist, empfehle ich Ihnen diese beiden Bereiche durchzulesen.

Der Unterschied zwischen Sprites und Sprite3D's

In PureBasic gibt es zwei unterschiedliche Arten von Sprites. Die erste Art ist das, was Sie als ein normales Sprite bezeichnen würden, welches aus einem gezeichneten oder einem geladenen Bild besteht, das angezeigt wird und mit den Befehlen aus der Standard Sprite Bibliothek manipuliert werden kann.

Die zweite Art ist eigentlich das gleiche, nur das PureBasic zur Darstellung der Sprites eine sehr kleine 3D Engine verwendet. Die Befehle zum zeichnen und manipulieren von Sprites diesen Typs ermöglichen dem Programmierer grafischen Effekte zu erzielen, die mit normalen Sprites nicht möglich sind. Diese Effekte enthalten zum Beispiel Echtzeit Zooming, 3D Transformation und die Verschmelzung von Sprites. Bei der 3D Engine, die die Umwandlung und Anzeige der Sprites vollbringt, handelt es sich nicht um die OGRE Engine, die im nächsten Kapitel behandelt wird, sondern es handelt sich um eine selbstentwickelte 3D Engine, die speziell zum Verarbeiten dieser Sprites entwickelt wurde.

Verwenden von normalen Sprites

Um ein Sprite für die Verwendung in einem normalen Programm zu erstellen, können Sie eines mit dem 'LoadSprite()' Befehl laden (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite & Screen → LoadSprite) oder ein neues mit dem 'CreateSprite()' Befehl erstellen (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite & Screen → CreateSprite).

Um ein existierendes Bild zu laden, das Sie als Sprite verwenden wollen, verwenden Sie den 'LoadSprite()' Befehl, der dem 'LoadImage()' Befehl sehr ähnlich ist und drei Parameter benötigt. Der erste Parameter ist

die PB Nummer, die Sie dem Sprite zuordnen möchten. Der zweite Parameter ist der Dateiname des Bildes, das Sie als Sprite laden wollen, denken Sie daran bei Bedarf den richtigen Bilddecoder zu laden. Der dritte Parameter gibt den Sprite Modus an, der definiert wie das Sprite verwendet wird, hierzu erfahren Sie gleich mehr.

Um ein eigenes Sprite zu erstellen verwenden Sie den 'CreateSprite()' Befehl. Auf dieses neu erstellte Sprite können Sie dann mit den 2D Zeichenbefehlen zeichnen. Dieser Befehl ist dem 'CreateImage()' Befehl sehr ähnlich und benötigt vier Parameter. Der erste Parameter ist die PB Nummer, die Sie dem Sprite zuweisen wollen. Der zweite und der dritte Parameter definiert die Breite und die Höhe des neuen Sprites in Pixel, und der vierte Parameter definiert den Sprite Modus.

Beide Befehle erstellen ein neues Sprite und beide haben einen optionalen 'Modus' Parameter. Dieser Modus definiert das interne Sprite Format, weshalb ein bestimmtes Sprite zweckmäßig unter Verwendung von anderen Sprite Befehlen angezeigt wird. Dieser Format Parameter wird üblicherweise als eingebaute Konstante übergeben. Abb. 38 listet die Modus Konstanten auf und beschreibt jeden einzelnen Modus.

Abhängig davon welchen Sprite Befehl Sie verwenden möchten um das Sprite anzuzeigen, müssen Sie mit diesem ein korrekt formatiertes Sprite verwenden, und dieses Format wird definiert wenn Sie ein Sprite erstellen oder laden.

Sprite Modi

Verwendete Konstante	Modus Beschreibung
keine	Der Standard Modus. Das Sprite wird wenn möglich ins Video RAM geladen.
#PB_Sprite_Memory	Das Sprite wird nicht ins Video RAM, sondern in den normalen RAM Speicher geladen, damit es mit dem 'StartSpecialFX()' Befehl verwendet werden kann.
#PB_Sprite_Alpha	Bei dem Sprite muss es sich um ein 8 Bit Graustufen Bild handeln, welches für die Verwendung mit den Befehlen 'DisplayAlphaSprite()' oder 'DisplaySchadowSprite()' vorbereitet wird. Wenn Sie den 'StartSpecialFX()' Befehl verwenden möchten, dann muss '#PB_Sprite_Memory' ebenfalls mit angegeben werden.
#PB_Sprite_Texture	Das Sprite wird mit 3D Unterstützung erstellt, so das Sie ein 3D Sprite mit dem 'CreateSprite3D()' Befehl erstellen können.
#PB_Sprite_AlphaBlending	Das Sprite wird mit Unterstützung für den Alphakanal erstellt. Das geladene Bild muss von einem Bildformat sein, das einen Alphakanal enthalten kann. Die einzigen zur Zeit unterstützten Bildformate mit dieser Eigenschaft sind PNG und TIFF. (Wenn Sie beabsichtigen, das Sprite in ein 3D Sprite zu konvertieren, müssen Sie auch den '#PB_Sprite_Texture' Modus mit angeben.)

Die Modi können wie üblich mit dem bitweise OR Operator '|' kombiniert werden. Abb. 38

Hier sind Beispiele, wie der Modus für jeden Sprite Anzeige Befehl zweckmäßig definiert wird. Wo es möglich war, habe ich ein Beispiel angefügt, wie ein Sprite mit den richtigen Modus Einstellungen erstellt und geladen wird, damit es zweckmäßig angezeigt wird, wenn Sie den entsprechenden Befehl davor aufrufen.

'DisplaySprite()' und 'DisplayTransparentSprite()'

```
;Standard Format
CreateSprite(#PB_NUMMER, Breite.i, Hoehe.i)
LoadSprite(#PB_NUMMER, "Bild.bmp")
```

'DisplaySprite3D()'

```
;Ohne Alphakanal
CreateSprite(#PB_NUMMER, Breite.i, Hoehe.i, #PB_Sprite_Texture)
LoadSprite(#PB_NUMMER, "Bild.bmp", #PB_Sprite_Texture)
```

```
;Mit Alphakanal
CreateSprite(#PB_NUMMER,Breite.i,Hoehe.i,#PB_Sprite_Texture|#PB_Sprite_AlphaBlending)
LoadSprite(#PB_NUMMER, "Bild.bmp", #PB_Sprite_Texture | #PB_Sprite_AlphaBlending)
```

'DisplayTranslucentSprite()'

```
;Laden in normales RAM zum bearbeiten mit dem 'StartSpecialFX()' Befehl
CreateSprite(#PB_NUMMER, Breite.i, Hoehe.i, #PB_Sprite_Memory)
LoadSprite(#PB_NUMMER, "Bild.bmp", #PB_Sprite_Memory)
```

'DisplayAlphaSprite()', 'DisplayShadowSprite()' und 'DisplaySolidSprite()'

```
;Laden in normales RAM zum bearbeiten mit dem 'StartSpecialFX()' Befehl
;und als Alpha Typ Sprite spezifizieren
CreateSprite(#PB_NUMMER, Breite.i, Hoehe.i, #PB_Sprite_Memory | #PB_Sprite_Alpha)
LoadSprite(#PB_NUMMER, "Bild.bmp", #PB_Sprite_Memory | #PB_Sprite_Alpha)
```

Diese Beispiele sollten ihnen einen guten Einblick geben, wie Sie Sprites für jeden Anzeigebefehl laden und erstellen müssen.

Sie können auch auf Sprites zeichnen

Wenn Sie ein Sprite erstellt oder geladen haben, wollen Sie vielleicht auch auf dieses Zeichnen. Ähnlich wie beim zeichnen auf ein Bild, können Sie das mit den Standard 'StartDrawing()' Befehl ermöglichen, Sie müssen nur als Ausgabekanal 'SpriteOutput()' definieren. Das erlaubt es Ihnen die Standard 2D Zeichenbefehle zu verwenden und auf die Spriteoberfläche zu zeichnen. Sie können auch ein Sprite auf ein anderes Sprite zeichnen. Dazu müssen Sie die Sprite Zeichenausgabe vom 'Back Buffer' auf das Zielsprite umschalten. Hierzu benötigen Sie den 'UseBuffer()' Befehl. Dieser Befehl benötigt einen Parameter, nämlich die PB Nummer des Sprite auf das Sie die Ausgabe umleiten wollen. Wenn umgeschaltet wurde, zeichnen alle weiteren Sprite Anzeigebefehle direkt auf das Zielsprite. Wenn Sie fertig sind, müssen Sie den 'UseBuffer()' Befehl erneut verwenden, um die Ausgabe wieder auf den 'Back Buffer' umzuleiten. Verwenden Sie diesmal als Parameter '#PB_Default'.

'SpecialFX' Sprite Befehle

In den Modus Beispielen habe ich bereits die 'SpecialFX' Sprites erwähnt. Es handelt sich hierbei um normale Sprites, die allerdings sehr viel schneller arbeiten wenn Sie von 'StartSpecialFX()' Befehlen umschlossen werden. Alle Sprites die Spezialeffekte wie diese verwenden sind in der Lage gutaussehende Grafiken zu produzieren, da allerdings alles im RAM Speicher des Computers gerendert wird, sind diese Sprites nicht so schnell wie Sprite3D's. Hier ein kurzer Codeschnipsel der die Verwendung dieser Befehle zeigt:

```
...
StartSpecialFX()
  DisplayAlphaSprite(#SPRITE_NUMMER, 64, 64)
  DisplayRGBFilter(100, 100, 100, 100, 0, 255, 0)
  DisplayTranslucentSprite(#SPRITE_NUMMER, 128, 128, 128)
  ;usw...
StopSpecialFX()
...
```

Wie Sie in dem Beispiel sehen können, leitet der 'StartSpecialFX()' Befehl den Block ein und der 'StopSpecialFX()' Befehl schließt ihn ab. Alle Spezialeffekt Befehle müssen sich zwischen diesen beiden Befehlen befinden. Durch diese Vorgehensweise erhöht sich die Rendergeschwindigkeit der Spezialeffekt Befehle, anderenfalls wäre die Anzeige Leistung sehr schwach.

Wenn Sie diese Befehle verwenden ist es wichtig zu verstehen, dass diese vor allen anderen grafischen Befehlen ausgeführt werden müssen. Das hängt mit der Art und Weise zusammen wie PureBasic die Interna von Spezialeffekt Zeichnungen verarbeitet. Wenn Sie andere Zeichenbefehle vor dem Spezialeffekt Block verwenden, werden diese auf dem Back Buffer beim Aufruf des 'StartSpecialFX()' Befehls überschrieben. Wenn Sie einen 'ClearScreen()' Befehl verwenden, um den Puffer vor dem Zeichnen zu löschen, sollten Sie diesen ebenfalls in den Spezialeffekte Block einbauen. Sie müssen weiterhin beachten, das in einer Hauptschleife nur ein Spezialeffekte Block vorhanden sein darf, da dies die Leistung erheblich beschleunigt.

Folgende Befehle verwenden einen 'StartSpecialFX()' Block:

'DisplayAlphaSprite()'
'DisplaySolidSprite()'
'DisplayShadowSprite()'
'DisplayRGBFilter()'
'DisplayTranslucentSprite()'

Mehr Informationen zu diesen Befehlen finden Sie in der Hilfedatei in der 'Sprite & Screen' Bibliothek (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite & Screen).

Anzeigen von normalen Sprites

Normale Sprites sind der Unterbau für jedes 2D Spiel das mit PureBasic erstellt wird. Dieser Typ Sprite wurde lange Zeit immer wieder verwendet um schöne optische Effekte in vielen guten Spielen zu erzeugen. Hier ist ein einfaches Beispiel, das die Erstellung eines Sprites zeigt, welches anschließend mehrfach auf den Bildschirm gezeichnet wird. Dies geschieht unter Verwendung des 'DisplayTransparentSprite()' Befehls:

```
#SPRITE_HAUPT     = 1
#ANZAHL_BAELLE    = 500

;Setzen der Breite, Höhe und Farbtiefe des Screens
;Aufgrund der Seitenbreite wurden hier abgekürzte Variablennamen verwendet :(
Global ScrB.i    = 1024
Global ScrH.i    = 768
Global ScrT.i    = 32
Global Beenden.i = #False

Structure BALL
  x.f
  y.f
  XUrsprung.l
  YUrsprung.l
  Radius.l
  Winkel.f
  Geschwindigkeit.f
EndStructure
Global Dim Ball.BALL(#ANZAHL_BAELLE)

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Konvertiere Grad in Bogenmaß
Procedure.f GradZuRad(Winkel.f)
  ProcedureReturn Winkel.f * #PI / 180
EndProcedure

;Initialisiere alle Balldaten
Procedure InitialisiereBaelle()
  For x.i = 0 To #ANZAHL_BAELLE
    Ball(x)\XUrsprung         = Random(ScrB) - 32
    Ball(x)\YUrsprung         = Random(ScrH) - 32
    Ball(x)\Radius            = Random(190)  + 10
    Ball(x)\Winkel            = Random(360)
    Ball(x)\Geschwindigkeit   = Random(2)    + 1
  Next x
EndProcedure

;Initialisiere Umgebung
Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
Fehlerbehandlung(OpenScreen(ScrB, ScrH, ScrT, "Ball", 2), "Kann Screen n. öffnen.")
SetFrameRate(60)

;Erstelle ein Bild
Global Offset.f = 32
If CreateSprite(#SPRITE_HAUPT, 64, 64)
```

```
    If StartDrawing(SpriteOutput(#SPRITE_HAUPT))
      Box(0, 0, 64, 64, RGB(255, 255, 255))
      For x.i = 220 To 1 Step - 1
        Offset + 0.025
        Circle(Offset, 64 - Offset, x / 8, RGB(0, 255 - x, 0))
      Next x
      StopDrawing()
    EndIf
  EndIf
  TransparentSpriteColor(#SPRITE_HAUPT, RGB(255, 255, 255))
  InitialisiereBaelle()

  ;Hauptschleife
  Repeat
    ClearScreen(RGB(56, 76, 104))
    For x.i = 0 To #ANZAHL_BAELLE
      Ball(x)\x = Ball(x)\XUrsprung + (Ball(x)\Radius*Cos(GradZuRad(Ball(x)\Winkel)))
      Ball(x)\y = Ball(x)\YUrsprung + (Ball(x)\Radius*Sin(GradZuRad(Ball(x)\Winkel)))
      Ball(x)\Winkel + Ball(x)\Geschwindigkeit
      DisplayTransparentSprite(#SPRITE_HAUPT, Ball(x)\x, Ball(x)\y)
    Next x
    FlipBuffers()

    ExamineKeyboard()
    If KeyboardReleased(#PB_Key_Escape)
      Beenden = #True
    EndIf

  Until Beenden = #True
  End
```

Der 'DisplayTransparentSprite()' Befehl ermöglicht Ihnen das Anzeigen eines Sprites auf einem Screen, dem Anzeigen eines Sprites mit 'DisplaySprite()' sehr ähnlich. Bei Verwendung des 'DisplayTransparentSprite()' Befehl wird allerdings eine Farbe im Bild als Transparente Farbe reserviert. Dadurch erscheint ein Sprite nicht immer als Rechteck.

In diesem Beispiel habe ich ein neues Sprite erstellt und dieses unter Verwendung eines 'Box()' Befehls mit weißer Farbe gefüllt. Danach habe ich eine schattierte grüne Kugel auf das Sprite gezeichnet. Hierzu habe ich den 'Circle()' Befehl in einer Schleife verwendet. Im Anschluss daran habe ich mit dem 'TransparentSpriteColor()' Befehl alle weißen Pixel auf dem Sprite als transparent definiert. Dieser Befehl benötigt zwei Parameter. Der erste Parameter ist die PB Nummer des Sprite, das geändert werden soll und der zweite Parameter ist die Farbe, die Sie als transparent definieren wollen. Wenn die Farbe definiert wurde, wird bei der nächsten Verwendung von 'DisplayTransparentSprite()' ein Sprite minus die transparente Farbe angezeigt. Wenn Sie dieses Beispiel starten, sehen Sie einen Bildschirm voller grüner Kugeln, die keinen weißen Rand haben. Das ist eine gute Möglichkeit, Sprites mit Transparenz anzuzeigen.

Sie werden auch bemerkt haben, das der Befehl zum anzeigen eines normalen Sprite nicht von speziellen Befehlen wie 'StartSpecialFX()' oder 'StartDrawing()', usw. umschlossen ist. Sie können die normalen Sprite Anzeigebefehle eigenständig verwenden. So lange die Sprite Engine initialisiert ist und ein Screen geöffnet wurde, können Sie die Standard Sprite Befehle verwenden um ein Sprite anzuzeigen. Die Standard Sprite Anzeige Befehle sind:

'DisplaySprite()'
'DisplayTransparentSprite()'

Verwenden von Sprite3D's

PureBasic bezeichnet seine 3D Sprites mit dem leicht verstümmelten Namen 'Sprite3D'. Jedes Sprite3D ist eine 2D Oberfläche, die aus zwei Polygonen besteht. Diese Polygone werden durch eine kleine 3D Engine erstellt und können im 3D Modus verformt werden, aber jedes Sprite3D wird letztendlich als 2D Objekt auf den Screen gezeichnet. Verwirrt? Gut, lassen Sie uns weitermachen.

Ein Sprite3D ist in PureBasic tatsächlich ein normales Sprite das 3D Unterstützung erhalten hat, und als solches benötigt es zum anzeigen eine kleine 3D Engine. Um überhaupt Sprite3D's in Ihrem Programm anzeigen zu können, müssen Sie die Sprite3D Engine initialisieren, bevor Sie einen Sprite3D Befehl verwenden können. Dies erledigen Sie mit dem 'InitSprite3D()' Befehl. Dieser Befehl ist ein Unterbefehl von 'InitSprite()', weshalb dieser zuvor aufgerufen werden muss.

Jedes Sprite3D beginnt sein Leben als normales Sprite das ebenfalls geladen oder erstellt wird, allerdings mit dem '#PB_Sprite_Texture' Modus initialisiert wird. Diese normale Version wird zu keinem Zeitpunkt angezeigt, stattdessen wird diese normale Version mit dem 'CreateSprite3D()' Befehl in eine 3D Version gewandelt. Dieser Befehl benötigt zwei Parameter. Der erste Parameter ist die PB Nummer die Sie dem Sprite zuweisen möchten. Der zweite Parameter ist die PB Nummer des zuvor erstellten oder geladenen Sprite, das Sie in ein Sprite3D konvertieren möchten.

So sieht diese Konvertierungsprozedur in PureBasic aus:

```
LoadSprite(#NORMALES_SPRITE, "Bild.bmp", #PB_Sprite_Texture)
CreateSprite3D(#SPRITE_3D, #NORMALES_SPRITE)
```

Wenn das Sprite3D auf diese Weise erstellt wurde, können wir es mit dem 'DisplaySprite3D()' Befehl auf dem Bildschirm anzeigen. Um die Sache etwas genauer zu erklären folgt hier ein Beispiel, das Sprite3D's anzeigt und manipuliert:

```
UsePNGImageDecoder()

Enumeration
  #SPRITE_2D
  #SPRITE_3D
EndEnumeration

#ANZAHL_BLUMEN = 150

;Setzen der Breite, Höhe und Farbtiefe des Screens
;Aufgrund der Seitenbreite wurden hier abgekürzte Variablennamen verwendet :(
Global ScrB.i = 1024
Global ScrH.i = 768
Global ScrT.i = 32
;Andere Globale Variablen
Global Beenden.i   = #False
Global XUrsprung.i = ScrB / 2
Global YUrsprung.i = ScrH / 2

Structure BLUME
  XPos.f
  YPos.f
  Breite.f
  Hoehe.f
  Winkel.f
  Radius.f
  RadiusSchritt.f
EndStructure
Global Dim Blumen.BLUME(#ANZAHL_BLUMEN)

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Konvertiere Grad in Bogenmaß
Procedure.f GradZuRad(Winkel.f)
  ProcedureReturn Winkel.f * #PI / 180
EndProcedure

;Initialisiere alle Blumen
Procedure InitialisiereBlumen()
  For x.i = 0 To #ANZAHL_BLUMEN
    Blumen(x)\Breite      = 0
    Blumen(x)\Hoehe       = 0
    Blumen(x)\Winkel      = Random(360)
    Blumen(x)\Radius      = 1.0
    Blumen(x)\RadiusSchritt  = (Random(30) / 10) + 1.0
  Next x
EndProcedure
```

```
;Blumen zurücksetzen
Procedure ResetBlume(Index.i)
  Blumen(Index)\Breite      = 0
  Blumen(Index)\Hoehe       = 0
  Blumen(Index)\Winkel      = Random(360)
  Blumen(Index)\Radius      = 1.0
  Blumen(Index)\RadiusSchritt  = (Random(30) / 10) + 1.0
  ProcedureReturn
EndProcedure

;Initialisiere Umgebung
Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(InitSprite3D(), "InitSprite3D() fehlgeschlagen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
Fehlerbehandlung(OpenScreen(ScrB, ScrH, ScrT, "Blume", 2), "Kann Screen n. öffnen.")
SetFrameRate(60)
Sprite3DQuality(#PB_Sprite3D_BilinearFiltering)

;Lade Sprite
LoadSprite(#SPRITE_2D, "Blume.png", #PB_Sprite_Texture | #PB_Sprite_AlphaBlending)
CreateSprite3D(#SPRITE_3D, #SPRITE_2D)

InitialisiereBlumen()

;Hauptschleife
Repeat

  ClearScreen(RGB(200, 100, 100))

  Fehlerbehandlung(Start3D(), "Start3D() fehlgeschlagen.")
  For x.i = 0 To #ANZAHL_BLUMEN
    Blumen(x)\Breite + 1.5
    Blumen(x)\Hoehe  + 1.5
    Blumen(x)\Winkel + 1.0
    If Blumen(x)\Breite > 512.0 Or Blumen(x)\Hoehe > 512.0
      Blumen(x)\Breite  = 512.0
      Blumen(x)\Hoehe   = 512.0
    EndIf
    If Blumen(x)\Radius > ScrB
      ResetBlume(x)
    EndIf
    Blumen(x)\Radius + Blumen(x)\RadiusSchritt
    Blumen(x)\XPos = XUrsprung + (Blumen(x)\Radius*Cos(GradZuRad(Blumen(x)\Winkel)))
    Blumen(x)\YPos = YUrsprung + (Blumen(x)\Radius*Sin(GradZuRad(Blumen(x)\Winkel)))
    Blumen(x)\XPos - Blumen(x)\Radius / 3.5
    Blumen(x)\YPos - Blumen(x)\Radius / 3.5
    ZoomSprite3D(#SPRITE_3D, Blumen(x)\Breite, Blumen(x)\Hoehe)
    RotateSprite3D(#SPRITE_3D, Blumen(x)\Winkel, 0)
    DisplaySprite3D(#SPRITE_3D, Blumen(x)\XPos, Blumen(x)\YPos)
  Next x
  Stop3D()

  FlipBuffers()

  ExamineKeyboard()
  If KeyboardReleased(#PB_Key_Escape)
    Beenden = #True
  EndIf

Until Beenden = #True
End
```

Um mir zu ermöglichen in meinem Beispiel Sprite3D's zu verwenden, habe ich zuerst die normale Sprite Engine initialisiert und danach mit dem 'InitSprite3D()' Befehl die Sprite3D Engine aufgerufen. Dieses Vorgehen ist essentiell für die Verwendung von allen Sprite3D Befehlen. Danach habe ich den 'LoadSprite()' Befehl verwendet, um ein Bild im PNG Format zu laden. Das Bild mit dem Namen 'Blume.png' verwendet einen Alphakanal um einen transparenten Hintergrund zu erzeugen. Das Laden eines PNG Bildes mit Alphakanal erfordert die korrekte Definition des Sprite Modus im 'LoadSprite()' Befehl. Wenn Sie auf den Beispielcode schauen sehen Sie, dass ich den Sprite Modus als '#PB_Sprite_Texture | #PB_Sprite_Alphablending' definiert habe. Dadurch wird dem Compiler mitgeteilt, dass ich dieses Sprite als

Sprite3D verwenden möchte und dass dieses einen Alphakanal enthält. Nach dem korrekten laden und definieren kann ich mit dem 'CreateSprite3D()' Befehl ein Sprite3D erstellen, dessen Alphakanal erhalten bleibt.

Wenn das Sprite erstellt wurde, ist es an der Zeit die Sprites auf den Screen zu zeichnen. Das erreichen Sie mit dem 'DisplaySprite3D()' Befehl innerhalb eines 'Start3D()' Blocks. Das sieht etwa so aus:

```
...
Start3D()
  DisplaySprite3D(#SPRITE_3D, x, y, AlphaWert)
Stop3D()
...
```

Im Blumen Beispiel können Sie sehen, das der 'Start3D()' Block mittels einer kleinen Fehlerbehandlungs-Routine überprüft wird, um sicherzustellen dass der Start erfolgreich war. Wenn dies der Fall ist, können wir beginnen die Sprites zu zeichnen. Wenn er nicht korrekt gestartet wurde dürfen wir auf keinen Fall ein Sprite3D zeichnen, sonst wäre ein ernsthafter Programmabsturz die Folge. Der 'Start3D()' Block wird mit einem 'Stop3D()' Befehl abgeschlossen. Innerhalb dieses Blocks dürfen Sie keine Befehle aus der normalen Sprite Bibliothek verwenden. Dieser Bereich ist einzig für Sprite3D Befehle reserviert.

Um das eigentliche Sprite3d anzuzeigen verwenden Sie den 'DisplaySprite3D()' Befehl, der vier Parameter benötigt. Der erste Parameter ist die PB Nummer des anzuzeigenden Sprite3D. Der zweite und der dritte Parameter ist die 'x' und 'y' Position auf dem Puffer. Der optionale vierte Parameter ist der Alphawert des Sprite. Dieser Wert gibt an, wie transparent das Sprite auf den Bildschirm gezeichnet wird. Es handelt sich um einen Ganzzahlenwert von '0' (komplett transparent) bis '255' (völlig undurchsichtig).

Im Blumen Beispiel habe ich auch die Befehle 'ZoomSprite3D()' und 'RotateSprite3D()' verwendet, um die Sprite3D's in der Größe zu ändern und diese zu drehen. Diese Befehle sind sehr einfach zu verstehen. Detailliertere Informationen erhalten Sie in der PureBasic Hilfedatei im Bereich 'Sprite3D' (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sprite3D).

Sprite3D Qualität

Während der Verwendung von Sprite3D's in Ihrem Programm ist es möglich die Render Qualität der Sprites umzuschalten. Dies wird mit dem 'Sprite3DQuality()' Befehl ermöglicht. Dieser Befehl benötigt einen Parameter, nämlich den Rendermodus aller Sprite3D's in Ihrem Programm. Bei diesem Parameter handelt es sich um eingebaute Konstanten, die einem bestimmten Modus entsprechen.

Hier die Definitionen:

'#PB_Sprite3D_NoFiltering'
keine Filterung (schneller, aber sehr pixelig beim vergrößern und drehen)

'#PB_Sprite3D_BilinearFiltering'
Bilineare Filterung (langsamer, aber vermischt die Pixel, so das beim Zoomen und Rotieren saubere Übergänge entstehen)

In meinem Beispiel habe ich 'Sprite3DQuality(#PB_Sprite3D_BilinearFiltering)' verwendet. Damit wird die Bilineare Filterung eingeschaltet, die den Sprite3D's beim vergrößern und drehen ein sauberes, geschmeidiges Aussehen verleiht. Wenn Sie diesen Befehl nicht verwenden, ist die Standardqualität im Programm '#PB_Sprite3D_NoFiltering'. In diesem Fall ist dann keine Filterung angewählt, weshalb die Sprite3D's beim manipulieren pixelig werden.

11. 3D Grafik

In diesem Kapitel werde ich über 3D Grafik berichten, und wie PureBasic dazu die OGRE Engine verwendet, um diese auf den Bildschirm zu zeichnen. 3D Grafik verwendet imaginäre 3D Modelle um realistische Welten zu erzeugen, die den Eindruck von Breite, Höhe und Tiefe vermitteln. Der Bildschirm ist in diesem Fall ein Fenster durch das Sie auf diese 3D Welt blicken. Das ist natürlich nicht ganz richtig, denn während ich denke, dass ich durch meinen Monitor auf eine 3D Welt blicke (besonders bei 3D Spielen), weiß ich doch, dass ausgeklügelte mathematische Routinen diese imaginären 3D Modelle auf meinen Monitor zeichnen und mir den Eindruck von drei Dimensionen vermitteln. Glücklicherweise bewahrt Sie PureBasic vor all der benötigten Mathematik und verwendet zum zeichnen von 3D Grafiken auf Ihren Bildschirm die OGRE Engine. Das erlaubt es Ihnen auf einfache Weise komplexe Befehle aufzurufen, die komplizierte 3D Grafikeffekte erstellen. Das macht das Programmieren von 3D Grafiken in PureBasic schnell und sehr einfach.

Dieses Kapitel wird leider nicht allzu ausführlich über die Programmierung von 3D Grafiken berichten, da man mit diesem Thema ein eigenes Buch füllen könnte. Dieses Kapitel ist ein fundierter Einstieg in die Verwendung der Befehle und bietet Lösungen für verbreitete Probleme die hauptsächlich Anfänger betreffen.

Ein Überblick zur 'OGRE Engine'

Wie ich bereits erwähnt habe, sind für die Darstellung von dreidimensionalen Grafiken auf dem Bildschirm sehr komplizierte mathematische Routinen nötig, die die Modellformen, Modelltexturen, Positionen und Beleuchtung, usw. berechnen. Das kann sehr zeitaufwändig sein und ist harte Arbeit für das Gehirn. Dankenswerter Weise verwendet PureBasic die OGRE Engine, die diese stumpfsinnige Arbeit von Ihnen fernhält, so dass Sie sich voll auf Ihr Programm konzentrieren können. OGRE (Open source GRaphics Engine) ist eine quelloffene Echtzeit 3D Rendering Engine, die von Torus Knot Software Limited erstellt und entwickelt wurde. Die Engine wird frei im Internet angeboten und ist für jeden kostenlos verfügbar, für jegliche Verwendung.

Nach einem Blick auf den Quellcode der OGRE Engine stellte der Hauptprogrammierer von PureBasic (Frederic Laboureur) fest, dass er von so guter Qualität war, sodass er ihn in den PureBasic Befehlssatz mit einband. Die Engine selbst ist in PureBasic als Dynamic Linked Library (DLL) integriert, auf die die 3D Bibliothek von PureBasic zugreift. Der einzige Nachteil daran ist die Tatsache, dass die Ogre Engine eine Programmierschnittstelle (API) mit Hunderten von Befehlen zur Verfügung stellt, von denen PureBasic aber nicht alle unterstützt. Das hängt mit einigen Inkompatibilitäten mit der Entwicklungssprache der OGRE Engine zusammen und mit der Zeit die das PureBasic Team benötigt neue Befehle einzubinden und zu testen. Vielleicht werden in Zukunft mehr Befehle eingefügt, das PureBasic Team hat zumindest ein Interesse bekundet, den OGRE Befehlssatz für PureBasic Anwender zu erweitern. Ich denke, die Zeit wird es zeigen. Sie können den momentanen OGRE Befehlssatz in der Hilfe unter der Rubrik '3D Spiele & Multimedia Libraries' betrachten (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries).

Das OGRE Mesh Format

Damit Sie 3D Modelle mit der OGRE Engine anzeigen und manipulieren können benötigen Sie diese Modelle in einem ganz bestimmten Format. Das ist das OGRE 'Mesh' Format und diese Modelldateien haben die Dateiendung '*.mesh'.

Erstellen von eigenen Meshes

Um eigene Meshes zu erstellen benötigen Sie etwas, was als 3D Moddeling Programm bekannt ist. Diese Art von Programm ermöglicht es Ihnen ein Modell mit einer dreidimensionalen Anzeige zu erstellen. Darin können Sie in Echtzeit eine aktuelle Voransicht ihres Modells aus jedem beliebigen Winkel betrachten. Das Programm stellt weiterhin Werkzeuge zur Verfügung, die es Ihnen ermöglichen Bild Texturen zu Ihrem Modell hinzuzufügen, um diesem beim Rendern mit der 3D Engine eine Oberfläche zu geben. Fortgeschrittenere 3D Programme erlauben es Ihnen, Texturen sehr genau auszurichten, indem sie das Modell in eine 2D Form abwickeln können. Diese Abwicklung kann dann als Bild exportiert werden und stellt damit eine Vorlage für Ihre Textur zur Verfügung. Wenn Sie die daraus erstellte Textur dann auf das Original 3D Modell legen, passt sich die Form der Textur perfekt an die Form des Modells an. Sie können jedes beliebige 3D Programm verwenden, solange es in der Lage ist die fertigen Modelle im OGRE Mesh Format abzuspeichern.

Wenn Sie auf die OGRE Webseite schauen, finden Sie im Downloadbereich jede Menge Werkzeuge und Erweiterungen von Drittherstellern, die verschiedene Programme zum erstellen von 3D Modellen mit

Extrafunktionen ausstatten. Diese Erweiterungen ermöglichen es den Programmen 3D Modelle im richtigen Mesh-Format zu erstellen und zu laden. Es ist auf jeden Fall eine gute Idee hier vorbeizuschauen, bevor Sie sich für ein Programm zur Erstellung von 3D Modellen entscheiden.

OGRE Textur Formate

Jedes 3D Modell das mit einer 3D Engine angezeigt werden soll benötigt eine Textur. Diese Texturen sind nichts anderes als Bilder, die um das 3D Modell gewickelt werden und somit eine detaillierte Oberfläche erzeugen. Einige Menschen sprechen bei Texturen auch von 'Haut'. Manchmal haben Modelle mehr als eine Textur, um eine größerer Fläche zu bedecken oder um eine detailliertere Oberfläche zu erzeugen.

Wenn Sie Texturen zum erzeugen von Oberflächen für Ihr Modell verwenden, müssen Sie darauf achten, dass Sie das richtige Bildformat verwenden, da die Textur sonst nicht in Ihrem fertigen Programm zu sehen ist. Das ist wichtig zu beachten, da die meisten 3D Programme die Fähigkeit besitzen, mehrere Bildformate zu unterstützen, diese aber nicht von OGRE unterstützt werden. Die OGRE Engine, die von PureBasic verwendet wird, unterstützt PNG, TGA oder JPG Dateien. Wenn Sie diese Bildformate als Texturen für Ihre Modelle verwenden wissen Sie, dass alles wie erwartet funktionieren wird.

OGRE's 3D Koordinatensystem

Wenn Sie Objekte in einen 3D Raum zeichnen und manipulieren wollen, benötigen Sie ein geeignetes Koordinatensystem um die Objekte im Raum zu platzieren. OGRE verwendet ein Standard rechte Hand 'x, y, z' Koordinatensystem. Lassen Sie mich das kurz erklären. In einem Koordinatensystem wie diesem wird jede Dimension durch einen Buchstaben gekennzeichnet, daher die Buchstaben 'x, y, z'. Es ist wichtig zu wissen, welcher Buchstabe einer bestimmten Dimension zugeordnet ist. Abb. 39 zeigt die Dimensionen mit den zugeordneten Buchstaben. Die Abbildung zeigt ebenfalls, wie Sie mit drei Werten einfach auf jeden Punkt im Raum zugreifen können. Wenn Sie annehmen, dass sich eine 3D Szene innerhalb dieses Würfels befindet, können Sie jeden Punkt durch einen Satz von 'x, y, z' Werten beschreiben.

Koordinaten in OGRE's 3D Raum

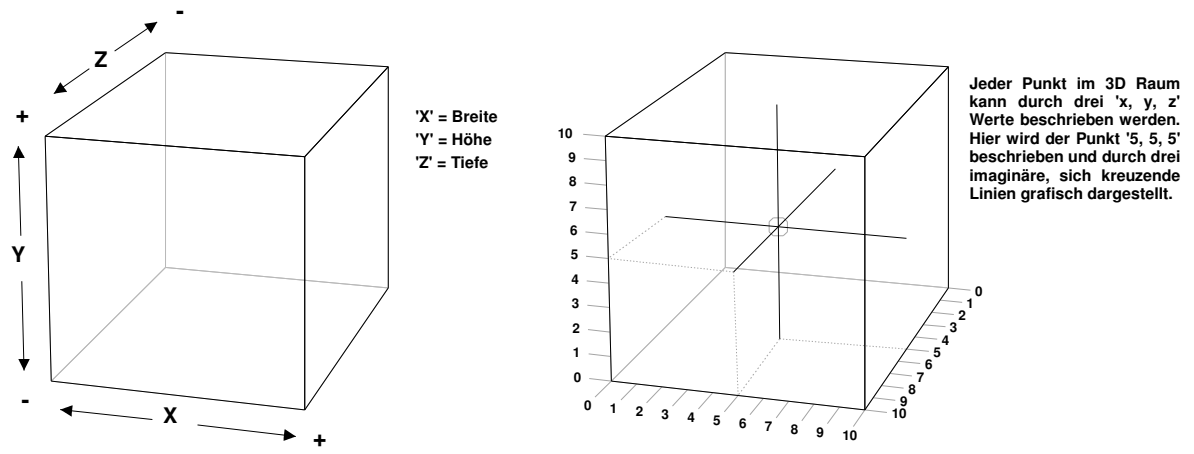


Abb. 39

Die Koordinaten selbst sind einfach zu verstehen, jede Dimension kann mit einem positiven oder negativen Wert angesprochen werden und die OGRE Engine verarbeitet die in einer ganz bestimmten Weise. Der linke Würfel in Abb. 39 verdeutlicht das durch die '+' und '-' Zeichen. Nehmen Sie einmal an, die linke untere Ecke ihres Bildschirms hätte die 3D Koordinaten '0, 0, 0'. Wenn Sie sich auf Ihrem Bildschirm nach rechts bewegen, steigt der 'x' Wert an, eine Bewegung nach oben lässt den 'y' Wert ansteigen. Die 'z' Koordinate verhält sich etwas anders, sie steigt an wenn Sie sich vom Monitor weg, auf sich selbst zu bewegen. Das wird im rechten Würfel in der Abbildung verdeutlicht, schauen Sie auf die Zahlen an der Seite.

Manche Menschen bevorzugen die Koordinaten '0, 0, 0' als den Mittelpunkt eines 3D Raumes, und die Koordinaten, die sich auf andere Objekte beziehen, können sowohl positive als auch negative Werte haben, abhängig davon, wo sich der von Ihnen beschriebene Punkt vom Mittelpunkt betrachtet befindet. Ich hoffe die Abbildung verdeutlicht Ihnen, wie die 'x, y, z' Koordinaten zu handhaben sind.

OGRE API Identifikatoren

Während Sie PureBasic und OGRE verwenden, um 3D Ansichten zu erstellen, verwenden Sie das OGRE API (Application Programming Interface) um die Sachen hinter der Szene zu verwalten. PureBasic hält das

OGRE API von Ihnen fern und bietet Ihnen die API Befehle in einer benutzerfreundlicheren Form von Basic Befehlen an. Zu bestimmten Anlässen müssen Sie OGRE Objekte über ihren OGRE Identifikator ansprechen. Bei diesen Identifikatoren handelt es sich um Werte vom Typ Integer, sehr ähnlich den Betriebssystem Identifikatoren, die ebenfalls von einem API benötigt werden (Win32 API). Um die OGRE Identifikatoren zu ermitteln können Sie folgende PureBasic Befehle verwenden:

'MaterialID(#MATERIAL)'
Gibt den OGRE Identifikator des Materials zurück, das Sie als PB Nummer im Parameter übergeben haben.

'MeshID(#MESH)'
Gibt den OGRE Identifikator des Mesh zurück, den Sie als PB Nummer im Parameter übergeben haben.

'TextureID(#TEXTURE)'
Gibt den OGRE Identifikator der Textur zurück, die Sie als PB Nummer im Parameter übergeben haben.

Ein sachter Einstieg

Im folgenden Beispiel zeige ich Ihnen, wie Sie eine 3D Umgebung initialisieren, einen 3D Mesh laden, eine Textur auf diesen legen und ihn dann auf dem Bildschirm anzuzeigen. Um die Sache etwas spannender zu machen, habe ich dem Modell eine Drehung verpasst und der Szene eine Lichtquelle hinzugefügt, damit das Modell bei der Anzeige nicht zu flach wirkt.

```
Enumeration
  #MESH
  #TEX
  #MAT
  #OBJEKT_INVADER
  #LICHT
  #KAMERA_EINS
EndEnumeration

;Setzen der Breite, Höhe und Farbtiefe des Screens
;Aufgrund der Seitenbreite wurden hier abgekürzte Variablennamen verwendet :(
Global ScrB.i = 1024
Global ScrH.i = 768
Global ScrT.i = 32
;Andere Globale Variablen
Global Beenden.i = #False

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Initialisiere Umgebung
Fehlerbehandlung(InitEngine3D(), "InitEngine3D() fehlgeschlagen.")
Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(OpenScreen(ScrB, ScrH, ScrT, ""), "Kann Screen n. öffnen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
SetFrameRate(60)

Add3DArchive("Data\", #PB_3DArchive_FileSystem)

Fehlerbehandlung(LoadMesh(#MESH, "Invader.mesh"), "Kann Mesh nicht laden")
Fehlerbehandlung(LoadTexture(#TEX, "Invader.png"), "Kann Textur nicht laden")
Fehlerbehandlung(CreateMaterial(#MAT, TextureID(#TEX)), "Kann Material n. erstell.")
CreateEntity(#OBJEKT_INVADER, MeshID(#MESH), MaterialID(#MAT))

CreateLight(#LICHT, RGB(255, 255, 255), 0, 5, 0)

CreateCamera(#KAMERA_EINS, 0, 0, 100, 100)
CameraLocate(#KAMERA_EINS, 0, 1, 2.5)
RotateCamera(#KAMERA_EINS, -15, 0, 0)
```

```
;Hauptschleife
Repeat

  y.i + 2
  RotateEntity(#OBJEKT_INVADER, 0, y, 0)

  RenderWorld()
  FlipBuffers()

  ExamineKeyboard()
  If KeyboardReleased(#PB_Key_Escape)
    Beenden = #True
  EndIf

Until Beenden = #True
End
```

Dieses Beispiel ist sehr nützlich für Sie, da es alle notwendigen Schritte aufführt, die Sie vollziehen müssen um ein 3D Modell auf dem Bildschirm anzuzeigen. Wie immer ist der Code recht überschaubar, allerdings werde ich die Hauptpunkte noch einmal ansprechen, damit Sie nichts übersehen.

Nachdem die Konstanten, Variablen und Prozeduren deklariert sind, initialisiere ich die 3D Umgebung mit dem 'InitEngine3D()' Befehl. Dieser Befehl ist notwendig, um die 3D Engine im Compiler Verzeichnis zu laden. Diese DLL Datei enthält die OGRE Engine in einer vorkompilierten Form, damit Sie diese einfach in Ihren Programmen verwenden können.

Wenn die 3D Umgebung initialisiert ist, kann ein Screen auf die übliche Art geöffnet werden, zuerst die Sprite Engine initialisieren und dann den Screen öffnen. Das muss alles in dieser Reihenfolge geschehen, da sonst ein Compiler Fehler auftritt und die IDE entsprechende Fehlermeldungen ausgibt. Wenn dies der Fall ist, müssen Sie das Programm beenden und die Initialisierung in der richtigen Reihenfolge durchführen. Danach können Sie das Programm erneut starten.

Einbinden von OGRE in Ihr 3D Programm

Wenn Sie mit PureBasic ein 3D Programm erstellen und wollen dieses an andere Personen weitergeben (entweder kommerziell oder als Freeware), müssen Sie daran denken, die OGRE 3D Engine mit Ihrem Programm mitzuliefern.

PureBasic verwendet OGRE als eine kompilierte DLL, um alle 3D Funktionen in einer kompakten und portablen Art zur Verfügung zu stellen. Wenn Sie Ihr Programm kompilieren, sind die OGRE Befehle nicht wie die anderen PureBasic Befehle direkt in Ihr Programm eingebunden, sondern sie werden dynamisch aus einer DLL Datei aufgerufen. Das bedeutet, das sich die DLL immer bei Ihrem Programm befinden muss damit es korrekt funktioniert.

Das geschieht während der Entwicklung automatisch, denn wenn Sie in der IDE 'F5' drücken und Ihr Programm kompilieren und starten, dann wird ein temporäres Executable im '\Compilers' Verzeichnis erstellt, wo sich auch die OGRE Engine befindet. Das stellt sicher, das dass testen der 3D Befehle so einfach wie möglich ist. Wenn Sie dann Ihr fertiges Executable erstellen, muss die OGRE Engine in das gleiche Verzeichnis kopiert werden.

Damit Ihr 3D Programm ordnungsgemäß funktioniert, müssen Sie die Datei 'Engine3D.dll' mit Ihrem Programm weitergeben.

Diese DLL finden Sie im '\Compilers' Verzeichnis innerhalb Ihres PureBasic Ordners. Diese Datei muss sich im selben Verzeichnis wie Ihr Programm befinden, andere Ordner sind nicht möglich.

Erstellen eines 3D Archivs das Medien enthält

Nachdem alle Initialisierungen durchgeführt sind, können wir die 3D Befehle verwenden, und das erste was wir nun erledigen müssen, ist ein '3D Archiv' zu spezifizieren. Das ist ein Pfad, in dem OGRE nach externen Dateien zum laden sucht, wie zum Beispiel Modelle, Texturen, Maps, usw. Sie müssen mindestens ein 3D Archiv in Ihrem Programm spezifizieren, sonst wird der Compiler einen Fehler melden. Um ein solches Archiv zu spezifizieren verwenden Sie den 'Add3DArchive()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Engine3D → Add3DArchive). Dieser Befehl benötigt zwei Parameter, der

erste ist ein String, der einen Dateipfad spezifiziert und der zweite ist eine eingebaute Konstante, die dem Compiler mitteilt, von welcher Art das Verzeichnis ist. Es gibt folgende eingebaute Konstanten:

'#PB_3DArchive_FileSystem'
Teilt dem Compiler mit, das es sich beidem übergebenen Pfad um ein normales Verzeichnis handelt.

'#PB_3DArchive_Zip'
Teilt dem Compiler mit, das es sich beidem übergebenen Pfad um eine komprimierte ZIP Datei handelt.

In meinem Invader Beispiel habe ich ein 3D Archiv mit der Angabe eines relativen Pfades erzeugt. Dadurch wird OGRE mitgeteilt, das sich im Programmverzeichnis ein Ordner mit dem Namen '\Data' befindet. Wenn der Ordner auf diese Art spezifiziert wurde, erfolgen von nun an alle Medienzugriffe auf dieses Verzeichnis. Das erlaubt dann die einfache Verwendung von Befehlen wie:

```
...
LoadMesh(#MESH, "Invader.mesh")
...
```

Wenn das Programm auf einen Befehl trifft der externe Medien benötigt, dann schaut OGRE automatisch im spezifizierten 3D Archiv nach der angeforderten Datei. In diesem Fall sucht OGRE im '\Data' Ordner nach der Datei 'Invader.mesh'.

Wenn Sie ihre Medien in einem etwas genauer untergliederten Dateisystem ablegen wollen und dem '\Data' Ordner noch Unterverzeichnisse hinzufügen wollen, können Sie das nach Ihren eigenen Bedürfnissen durchführen. Sie müssen dann nur die Medien von ihrem neuen Speicherort laden, wie hier:

```
...
LoadMesh(#MESH, "Meshes\Invader.mesh")
LoadTexture(#TEX, "Texturen\Invader.png")
...
```

Hier sucht das Programm im Original 3D Archiv ('\Data') nach den Unterordnern '\Meshes' und '\Texturen' und lädt die Medien daraus. Wenn Ihnen die Syntax etwas holprig erscheint, können Sie anstelle der Ordnernamen im Befehlsaufruf auch weitere 3D Archive hinzufügen, die OGRE dann nach weiteren Medien durchsucht, wie hier:

```
...
Add3DArchive("Data\", #PB_3DArchive_FileSystem)
Add3DArchive("Data\Meshes\", #PB_3DArchive_FileSystem)
Add3DArchive("Data\Texturen\", #PB_3DArchive_FileSystem)
...
```

Nun können wir auf einfache Weise die Ladebefehle verwenden:

```
...
LoadMesh(#MESH, "Invader.mesh")
LoadTexture(#TEX, "Invader.png")
...
```

OGRE sucht nun in den Ordnern '\Data', '\DataMeshes' und '\Data\Texturen' nach diesen Dateien.

Sie können auch ZIP Dateien als 3D Archiv verwenden. Das kann sehr nützlich sein, wenn Sie Ihr Programm weitergeben und sich alle Medien in einer komprimierten Datei befinden. Wenn Sie diese Art von Archiv verwenden möchten, müssen Sie bei der Erstellung spezifizieren, dass es sich bei dem Archiv um eine ZIP Datei handelt. Das erreichen Sie mit der eingebauten Konstante '#PB_Archive_Zip', die Sie dem 'Add3DArchive()' Befehl als Parameter übergeben. Wenn Sie in der ZIP Datei Unterordner angelegt haben, können Sie auf diese im Standardformat zugreifen:

```
...
Add3DArchive("Data.zip", #PB_3DArchive_Zip)
LoadMesh(#MESH, "Meshes\Invader.mesh")
LoadTexture(#TEX, "Texturen\Invader.png")
...
```

Erstellen eines 3D Objektes
Wenn Sie ein 3D Archiv erstellt haben, können Sie damit beginnen, ein Objekt zu erstellen, das Ihre Medien verwenden kann. Dieses Objekt wird dann auf dem Bildschirm angezeigt und kann mit verschiedenen

eingebauten Befehlen in drei Dimensionen manipuliert werden. OGRE bezeichnet diese 3D Objekte als 'Entities'.

Ein 'Entity' ist ein Baustein aus einem Spiel oder einer Demo. Es handelt sich um ein 3D Modell das auf dem Bildschirm angezeigt wird, bestehend aus einem Mesh als Gerüst und einer Textur als 'Haut'. Ein Entity kann alles mögliche sein, das als 3D Objekt hergestellt werden kann. Das können Landschaften oder die Darsteller in einem Spiele sein.

Prozess zum erzeugen eines 'Entity'

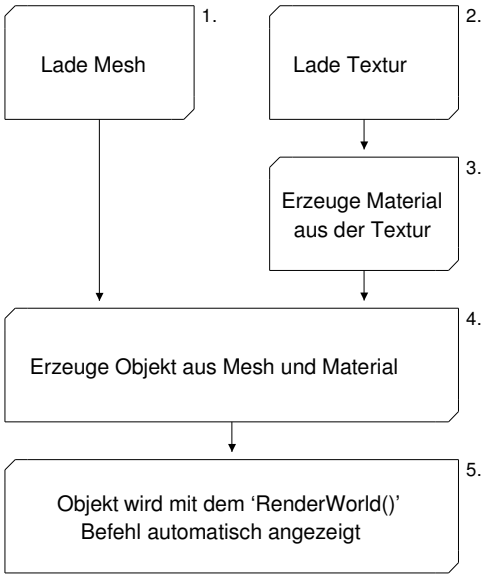


Abb. 40

Das Wissen, wie Sie diese 'Entities' zweckmäßig erstellen ist essentiell für jedes 3D Objekt, das Sie auf dem Bildschirm darstellen. Abb. 40 zeigt den Prozess wie ein 3D Objekt korrekt erstellt wird, und in welcher Reihenfolge die dazu nötigen Schritte erledigt werden müssen. Dieses Flussdiagramm habe ich in meinem vorherigen Invader Beispiel genau befolgt. Das ganze sieht als Code folgendermaßen aus:

```
...
Fehlerbehandlung(LoadMesh(#MESH, "Invader.mesh"), "Kann Mesh nicht laden")
Fehlerbehandlung(LoadTexture(#TEX, "Invader.png"), "Kann Textur nicht laden")
Fehlerbehandlung(CreateMaterial(#MAT, TextureID(#TEX)), "Kann Material n. erstell.")
CreateEntity(#OBJEKT_INVADER, MeshID(#MESH), MaterialID(#MAT))
...
```

Die Befehle zum vervollständigen dieses Prozesses sind sehr einfach. Der erste ist der 'LoadMesh()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Mesh → LoadMesh), der zwei Parameter benötigt. Der erste ist die PB Nummer, die mit dem Mesh verknüpft wird und der zweite ist die eigentliche Mesh Datei, die aus dem 3D Archiv geladen wird. Der nächste Befehl ist 'LoadTexture()' (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Texture → LoadTexture), der ebenfalls zwei Parameter benötigt. Der erste ist die PB Nummer, die mit der neuen Textur verknüpft wird und der zweite ist der Name der Bilddatei, die als Textur verwendet werden soll. Wenn Sie Bilder als Texturen verwenden, müssen Sie die Abmessungen des Bildes in Pixeln beachten. Ältere Grafikkarten unterstützen nur quadratische Texturen, deren Wert für die Seitenlänge eine Potenz von '2' ist. Um eine maximale Kompatibilität mit älteren Grafikkarten zu gewährleisten empfehle ich Ihnen sich nach diesen Vorgaben zu richten. Abb. 41 listet Standard Texturgrößen auf, die Sie für eine sichere Materialerstellung verwenden können.

Der Nächste Befehl lädt nichts, er erstellt lediglich ein Material. Dieses Material ist eine Textur im speziellen OGRE Format, die verschiedene Eigenschaften haben kann. Wir erstellen hier ein Material aus einer Textur, hierbei handelt es sich um die einfachste Form eines Materials. Über die ein wenig fortgeschritteneren Materialien berichte ich ein wenig später.

Standard Textur Bildgrößen

Breite in Pixel	Höhe in Pixel	Farbtiefe in Bit
32	32	8, 16 und 32
64	64	8, 16 und 32
128	128	8, 16 und 32
256	256	8, 16 und 32
512	512	8, 16 und 32
1024	1024	8, 16 und 32

Alle Standard Texturgrößen sind Potenzen von '2' Abb. 41

Zum erzeugen eines Materials verwenden Sie den 'CreateMaterial()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Material → CreateMaterial), der zwei Parameter benötigt. Der erste ist die PB Nummer die mit dem Material verknüpft wird und der zweite ist der OGRE Identifikator einer Textur. Dieser Identifikator kann mit dem 'TextureID()' Befehl ermittelt werden, der die PB Nummer einer Textur als Parameter benötigt.

Wenn wir die nötigen 'Zutaten' haben können wir unser 3D Objekt erzeugen. Dies erledigen wir mit dem 'CreateEntity()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Entity → CreateEntity), wie hier:

```
...
CreateEntity(#OBJEKT_INVADER, MeshID(#MESH), MaterialID(#MAT))
...
```

Wie Sie in dieser Zeile sehen können benötigt dieser Befehl drei Parameter. Der erste ist die PB Nummer mit der das Objekt verknüpft wird. Der zweite ist der OGRE Identifikator des Mesh, aus dem wir unser 3D Objekt erstellen möchten und der dritte Parameter ist der OGRE Identifikator des Materials mit dem wir den Mesh umhüllen möchten.

Die letzten beiden Identifikatoren können mit den Befehlen 'MeshID()' und 'MaterialID()' ermittelt werden. Diese beiden Befehle benötigen jeweils die PB Nummer als Parameter und geben den entsprechenden OGRE Identifikator zurück.

Nachdem das 3D Objekt erstellt wurde steht es in der Warteschlange, bereit um an den Koordinaten '0, 0, 0' auf den Screen gezeichnet zu werden. Bis dahin werden wir mit der Aufschlüsselung des Invader Beispiels weitermachen.

Erleuchte den Weg

Nachdem das Entity erstellt wurde habe ich eine Lichtquelle erzeugt um die Szene besser auszuleuchten. Es handelt sich um eine glänzende Lichtquelle die jede beliebige Farbe haben kann, um das Entity besser herauszustellen. Wenn Sie eine 3D Umgebung erstellen, enthält diese standardmäßig schon eine gewisse Menge Umgebungslicht, allerdings wird dieses Licht nicht von einem bestimmten Punkt ausgestrahlt, sondern es ist überall gleichmäßig auf die Szene verteilt. Dadurch wirkt das Entity etwas flach, da nicht definiert ist welche Seite beleuchtet wird und welche im Schatten liegt. Wenn Sie der Szene eine Lichtquelle hinzufügen, ermöglichen Sie es dem Entity zu glänzen oder schattig zu wirken, abhängig vom Standort der Lichtquelle in Relation zum Mesh. Zum Erzeugen der Lichtquelle habe ich den 'CreateLight()' Befehl verwendet (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Light → CreateLight).

```
...
CreateLight(#LICHT, RGB(255, 255, 255), 0, 5, 0)
...
```

Dieser Befehl benötigt fünf Parameter. Der erste ist die PB Nummer die der Lichtquelle zugewiesen wird. Der zweite ist die Farbe in der das Licht ausgestrahlt werden soll, dieser Parameter muss als 24 Bit Wert übergeben werden (dafür kann der 'RGB()' Befehl verwendet werden). Die Parameter Drei, Vier und Fünf sind optionale Parameter. Mit ihnen kann die Position der Lichtquelle im 3D Raum unter Verwendung von 'x, y, z' Koordinaten spezifiziert werden. Wenn diese Parameter nicht übergeben werden hat die Lichtquelle die Position '0, 0, 0', was der Mitte des 3D Raumes entspricht. Im Invader Beispiel habe ich die Position

'0, 5, 0' verwendet, wodurch sich die Lichtquelle um fünf Einheiten vom Boden abhebt. In Abb. 39 sehen Sie eine grafische Darstellung der Verwendung des 'x, y, z' Koordinatensystems im 3D Raum.

Die Kamera rollt

Nun da alle 3D Elemente platziert sind müssen wir einen Blickpunkt erstellen, durch den wir auf unsere 3D Szene schauen. OGRE nennt diese Blickpunkte Kameras, allerdings müssen wir diese beiden Begriffe trennen um Verwirrungen zu vermeiden. Ein Blickpunkt ist ein 2D Punkt auf dem aktuell geöffneten Screen, durch den wir das sehen worauf die 3D Kamera gerade blickt. Wenn ein Blickpunkt erstellt wird generiert dieser automatisch eine 3D Kamera die auf den Koordinaten '0, 0, 0' positioniert wird. Dieser Blickpunkt zeigt dann an auf was die 3D Kamera gerade blickt. Die Kamera kann positioniert und gedreht werden, der Blickpunkt bleibt jedoch immer an der selben Stelle auf dem Screen.

Um einen Blickpunkt - und damit automatisch eine 3D Kamera - in der aktuellen 3D Welt zu erstellen verwenden Sie den 'CreateCamera()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Camera → CreateCamera). Dieser Befehl benötigt fünf Parameter, und wie üblich ist der erste die PB Nummer die diesem Kameraobjekt zugewiesen wird. Der zweite und dritte Parameter sind die 'x' und 'y' Position des Blickpunktes von der linken oberen Ecke auf dem 2D Screen. Der vierte und fünfte Parameter sind die Breite und die Höhe des Blickpunktes durch den wir die 3D Szene betrachten. Wenn Sie diesen Befehl verwenden ist es wichtig das Sie verstanden haben welche Werte er verwendet, da diese doch stark von den anderen PureBasic Befehlen abweichen. Die Parameter Werte für den 'CreateCamera()' Befehl werden nicht in Pixel angegeben sondern in Prozentwerten, bezogen auf den aktuellen Screen. Im Invader Beispiel habe ich die Kamera folgendermaßen erstellt:

```
...
CreateCamera(#KAMERA_EINS, 0, 0, 100, 100)
...
```

Um das vollständig zu verstehen, lassen Sie uns den Befehlsaufruf auf eine individuelle Parameter Basis Herunterbrechen. Der erste Parameter ist die PB Nummer, welche einfach genug zu verstehen ist, so dass wir zügig zum nächsten Parameter springen können. Der zweite Parameter wurde mit '0' angegeben. Das bedeutet, das dieser Parameter auf Null Prozent der Screen Breite von '1024' berechnet wird, was dem Wert '0' entspricht. Der dritte Parameter ist ebenfalls mit '0' definiert. Es handelt sich hierbei ebenfalls um einen Prozentwert, der sich allerdings auf die Screen Höhe ('768') bezieht. Der vierte Parameter wurde mit '100' definiert. Das entspricht einhundert Prozent der Screen Breite, die in diesem Fall '1024' Bildpunkten entspricht. Der fünfte Parameter ist ebenfalls mit '100' definiert, was einhundert Prozent der Screen Höhe entspricht - in diesem Fall '768'. Dadurch ergeben sich unsere kompletten Screen Koordinaten für den Kamera Blickpunkt, nämlich 'x = 0, y = 0, Breite = 1024, Höhe = 768'. Dadurch wird der komplette Screen ausgefüllt und wir erhalten einen Vollbild Kamera Blickpunkt auf die 3D Welt.

Für diesen Befehl werden Prozentwerte verwendet, um die Position des Kamera Blickpunktes von der Screen Größe komplett unabhängig zu machen. Dadurch ist es auf einfache Weise möglich einen Kamera Blickpunkt zu erstellen der verschiedene Bildschirmauflösungen unterstützt. Wenn Sie zum Beispiel einen Kamera Blickpunkt mit halber Screenbreite erstellen wird dieser immer die Hälfte des Screens einnehmen, unabhängig davon welche Bildschirmauflösung verwendet wird.

Mehrere Kamera Blickpunkte

Die Verwendung von Prozentwerten ist eine gute Möglichkeit mehrere Kamera Blickpunkte zu verwenden. Nehmen wir einmal an Sie wollen ein Spiel mit geteiltem oder gevierteltem Bildschirm programmieren, dann würden Sie eine gute Berechnungsroutine benötigen um die Größe der Blickpunkte für jede Bildschirmauflösung zu berechnen (wenn diese vom Benutzer ausgewählt werden kann). Das wird alles für Sie von PureBasic erledigt. Durch die Angabe von Prozentwerten wird immer gewährleistet, dass jede Kamera den zugeteilten Platz beansprucht, unabhängig von der verwendeten Bildschirmauflösung. Wenn Sie mehrere Kameras erstellen ist die Reihenfolge der Erstellung ebenfalls sehr wichtig. Alle später im Code erstellten Blickpunkte erscheinen immer vor den zuvor erstellten Blickpunkten. Schauen Sie zum Beispiel auf diesen Code:

```
...
CreateCamera(#KAMERA_EINS,  0, 0, 100, 100)
CreateCamera(#KAMERA_ZWEI, 75, 0,  25,  25)
...
```

Hier habe ich einen Kamera Blickpunkt erstellt der den vollen Bildschirm einnimmt und einen weiteren Kamera Blickpunkt der im Vordergrund in der oberen rechten Ecke erscheint. Die Position des kleineren Blickpunktes befindet sich drei viertel der Bildschirmbreite von der linken oberen Ecke entfernt und hat ein

sechzehntel der Fläche des aktuellen Screens. Wenn ich den kleineren Blickpunkt zuerst erstellt hätte, würde dieser von dem Großen überlagert werden.

Jede der mehreren erstellten Kameras kann unabhängig voneinander in der 3D Welt bewegt und gedreht werden, wodurch dem Benutzer mehrere Blickwinkel auf die Szene geboten werden können. Sie können zum Beispiel eine zweite Kamera als Rückspiegel in einem 'First Person' 3D Auto-Rennspiel verwenden. Dieser Kamera Blickpunkt könnte oben im 2D Screen positioniert werden und Sie hätten die Möglichkeit stets zu sehen was hinter dem Auto passiert.

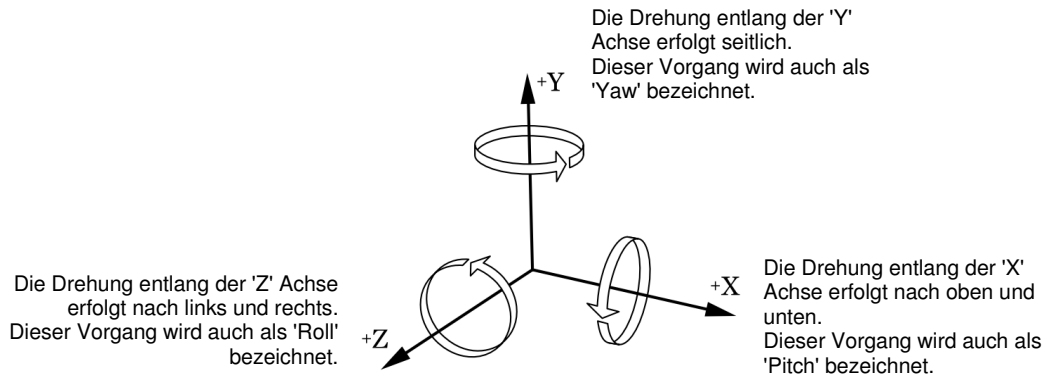
Bewegen und drehen von Kameras

Wenn Sie einen Blickpunkt erstellen generieren Sie automatisch eine 3D Kamera die frei in der 3D Szene bewegt und gedreht werden kann. Sie können die Kamera jederzeit bewegen und dort positionieren wo Sie wollen indem Sie Positionsbefehle in der Hauptschleife verwenden. In meinem Invader Beispiel habe ich zwei Befehle für die Feinpositionierung verwendet, wie hier:

```
...
CameraLocate(#KAMERA_EINS, 0, 1, 2.5)
RotateCamera(#KAMERA_EINS, -15, 0, 0)
...
```

Der Befehl 'CameraLocate()' bewegt die Kamera an eine absolute Position in der 3D Szene und benötigt vier Parameter. Der erste Parameter ist die PB Nummer der Kamera die Sie bewegen wollen, während die Parameter zwei, drei und vier die 'x, y, z' Koordinaten der Zielposition sind. Der zweite von mir verwendete Befehl ist 'RotateCamera()', der die Kamera um eine spezifizierte Achse dreht. Dieser Befehl benötigt ebenfalls vier Parameter, wovon der erste die PB Nummer der zu drehenden Kamera ist. Die Parameter zwei, drei und vier sind die relativen Rotationswinkel auf den Achsen 'x, y, z'.

Rotation im 3D Raum



Die Pfeile zeigen die positive Rotationsrichtung an

Abb. 42

Wenn Sie auf Abb. 42 schauen können Sie sehen, welche Achse zu welcher Rotationsbewegung gehört. Wenn ich zum Beispiel meine Kamera von der einen Seite auf die andere schwenken möchte, dann rotiere ich die 'y' Achse. Wenn Sie Werte für die 'x, y, z' Koordinaten in einem 3D Rotationsbefehl eingeben, dann können diese positive- und negative Werte haben, womit Sie die Drehrichtung umkehren können. Die Rotationspfeile in Abb. 42 zeigen die Drehrichtung für positive Werte. Bei der Verwendung von negativen Werten wird die Drehrichtung umgekehrt. Wenn ich zum Beispiel meine Kamera um fünfundvierzig Grad nach links schwenken möchte, dann gebe ich für den 'y' Rotationsparameter den Wert '45' an, bei einem fünfundvierzig Grad Schwenk nach rechts würde ich für den 'y' Rotationsparameter den Wert '-45' angeben.

Alle Bewegungs- und Rotationsbefehle können sowohl innerhalb als auch außerhalb der Hauptschleife verwendet werden um Kameras sowie andere 3D Objekte zu bewegen oder zu drehen. Die Verwendung der Befehle außerhalb der Hauptschleife ist sinnvoll wenn Sie zum Beispiel statische Objekte positionieren möchten. Die Verwendung innerhalb der Hauptschleife ist nötig, wenn Sie Ihre 3D Objekte in Echtzeit bewegen oder drehen möchten (oft zur Reaktion auf Benutzereingaben).

Alle Befehle in PureBasic die 3D Rotationen nutzen verwenden das gleiche 'x, y, z' System. Wenn Sie in die Hauptschleife des Invader Beispiels schauen, dort habe ich den 'RotateEntity()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Entity → RotateEntity) verwendet um das Invader-Entity während der Programm Laufzeit zu drehen. Auch dieser Befehl verwendet den gleichen Satz

Rotationsachsen wie in Abb. 42 beschrieben. Wenn Sie einen Befehl verwenden, der 3D Rotation benutzt, dann bietet Abb. 42 eine gute Referenz dafür welche Rotation benötigt wird.

In der PureBasic Hilfe können Sie noch mehr über die verfügbaren Befehle zur Verwendung mit OGRE Kameras erfahren (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Camera).

Rendern der 3D Welt auf den Screen

Wenn Sie 3D Befehle verwenden um eine 3D Welt zu erstellen, dann werden Ihre Meshes, Lichtquellen, Partikel, usw. nicht wie im 2D Modus direkt auf den Screen gezeichnet. Stattdessen liegen diese Objekte im Speicher, bereit zum erfassen und zeichnen. Wenn alles in Ihrer 3D Welt platziert ist müssen Sie den 'RenderWorld()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Engine3D → RenderWorld) benutzen um alle aktuellen 3D Kameras anzuweisen einen Schnappschuss von der aktuellen Szene zu machen. Diese Schnappschüsse werden dann auf den Backbuffer gezeichnet, skaliert und positioniert in Abhängigkeit vom jeweiligen Kamera Blickpunkt. Danach wird der 'FlipBuffers()' Befehl aufgerufen. Der Backbuffer wird in den Vordergrund gebracht und auf den Bildschirm gezeichnet. Sie können dieses Vorgehen im Invader Beispiel betrachten. Wenn Sie den 'RenderWorld()' Befehl nicht verwenden, dann wird keine 3D Grafik auf den Backbuffer gezeichnet. Ich hoffe dass damit das Invader Beispiel vollständig erklärt ist. Sie sollten nun, mit diesem Beispiel als Vorlage, in der Lage sein eigene Modelle zu laden und anzuzeigen.

Eine einfache 'First Person' Kamera

Nach dem Start des Invader Beispiels sind Sie vermutlich begierig darauf Ihr eigenes 3D Programm oder sogar ein 3D Spiel zu programmieren. Das erste was ich mich gefragt habe als ich die 3D Möglichkeiten von PureBasic gesehen habe war, ob sich damit ein 'First Person Shooter' Spiel erstellen lässt. Das nächste Code Beispiel zeigt wie Sie mit PureBasic eine 'First Person' Kamera in einer 3D Welt programmieren. Zugegeben, es ist kein vollwertiges Spiel, weit davon entfernt, aber es ist ein gutes Lernbeispiel für Anfänger. Dieser Code zeigt Ihnen wie Sie ein Terrain erstellen und eine Kamera darin mit der Tastatur und der Maus bewegen:

```
Enumeration
  #TEXTUR_GLOBAL
  #TEXTUR_DETAIL
  #MATERIAL_TERRAIN
  #KAMERA_EINS
EndEnumeration

#BEWEGUNG_GESCHWINDIGKEIT = 1

;Setzen der Breite, Höhe und Farbtiefe des Screens
;Aufgrund der Seitenbreite wurden hier abgekürzte Variablennamen verwendet :(
Global ScrB.i = 1024
Global ScrH.i = 768
Global ScrT.i = 32

;Andere Globale Variablen
Global Beenden.i = #False
Global MausXRotation.f, MausYRotation.f, TasteX.f, TasteZ.f, WunschKameraHoehe.f
Global AktuelleKamXPos.f = 545
Global AktuelleKamZPos.f = 280

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Initialisiere Umgebung
Fehlerbehandlung(InitEngine3D(), "InitEngine3D() fehlgeschlagen.")
Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(OpenScreen(ScrB, ScrH, ScrT, ""), "Kann Screen n. öffnen.")
Fehlerbehandlung(InitMouse(), "InitMouse() fehlgeschlagen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
SetFrameRate(60)
```

```
;Definiere 3D Archiv
Add3DArchive("Data\", #PB_3DArchive_FileSystem)

;Erstelle Terrain
Fehlerbehandlung(LoadTexture(#TEXTUR_GLOBAL, "Global.png"), "Kann Textur n. laden.")
Fehlerbehandlung(LoadTexture(#TEXTUR_DETAIL, "Detail.png"), "Kann Textur n. laden.")
CreateMaterial(#MATERIAL_TERRAIN, TextureID(#TEXTUR_GLOBAL))
AddMaterialLayer(#MATERIAL_TERRAIN, TextureID(#TEXTUR_DETAIL), #PB_Material_Add)
CreateTerrain("Terrain.png", MaterialID(#MATERIAL_TERRAIN), 1, 2, 1)

;Erstelle Standpunkt und 3D Kamera
CreateCamera(#KAMERA_EINS, 0, 0, 100, 100)
WunschKameraHoehe.f = TerrainHeight(AktuelleKamXPos, AktuelleKamZPos) + 10
CameraLocate(#KAMERA_EINS, AktuelleKamXPos, WunschKameraHoehe, AktuelleKamZPos)

;Hauptschleife
Repeat

  ;Aktualisiere Maus
  If ExamineMouse()
    MausYRotation - MouseDeltaX() / 10
    MausXRotation + MouseDeltaY() / 10
  EndIf
  RotateCamera(#KAMERA_EINS, MausXRotation, MausYRotation, 0)
  ;Registriere Tastendrücke und aktualisiere kontinuierlich die Kameraposition
  If ExamineKeyboard()
    If KeyboardPushed(#PB_Key_Left)  : TasteX = - #BEWEGUNG_GESCHWINDIGKEIT : EndIf
    If KeyboardPushed(#PB_Key_Right) : TasteX =   #BEWEGUNG_GESCHWINDIGKEIT : EndIf
    If KeyboardPushed(#PB_Key_Up)    : TasteZ = - #BEWEGUNG_GESCHWINDIGKEIT : EndIf
    If KeyboardPushed(#PB_Key_Down)  : TasteZ =   #BEWEGUNG_GESCHWINDIGKEIT : EndIf
    MoveCamera(#KAMERA_EINS, TasteX, 0, TasteZ)
    TasteX              = 0
    TasteZ              = 0
    AktuelleKamXPos.f   = CameraX(#KAMERA_EINS)
    AktuelleKamZPos.f   = CameraZ(#KAMERA_EINS)
    WunschKameraHoehe.f = TerrainHeight(AktuelleKamXPos, AktuelleKamZPos) + 10
    CameraLocate(#KAMERA_EINS, AktuelleKamXPos, WunschKameraHoehe, AktuelleKamZPos)
  EndIf

  RenderWorld()
  FlipBuffers()

  If KeyboardReleased(#PB_Key_Escape)
    Beenden = #True
  EndIf

Until Beenden = #True
End
```

Dieses Beispiel ähnelt dem Invader Beispiel, indem es ebenfalls eine 3D Umgebung auf die gleiche Weise initialisiert, somit sollten Sie mit dieser Vorgehensweise vertraut sein. Der Hauptunterschied besteht darin, dass ich eine Landschaft (Terrain) erstellt habe, auf der wir uns wie auf einem Boden bewegen können.

Terrains

Um ein Terrain zu erstellen benötigen Sie zuerst ein Material mit mehreren Lagen das Sie als Terrain Oberfläche verwenden. Um ein Material mit mehreren Lagen zu erstellen müssen Sie zunächst ein normales Material erstellen und diesem dann weitere Materiallagen hinzufügen. Wenn Sie auf das 'First Person Kamera' Beispiel schauen sehen Sie folgende Code Zeilen:

```
...
Fehlerbehandlung(LoadTexture(#TEXTUR_GLOBAL, "Global.png"), "Kann Textur n. laden.")
Fehlerbehandlung(LoadTexture(#TEXTUR_DETAIL, "Detail.png"), "Kann Textur n. laden.")
...
```

Diese beiden Befehle laden unter Verwendung des 'LoadTexture()' Befehls zwei Bilder im PNG Format als Standardtexturen. Das erste Bild ('Global.png') wird zum Hauptmaterial, das wie eine Große Decke über die Terrain Oberfläche gelegt wird. Das zweite Bild ('Detail.png') ist eine weitere Textur, die mit der ersten verschmolzen wird und in Kachelform auf das Terrain gelegt wird. Dadurch erhöhen sich die Details auf der Oberfläche. Um das eigentliche mehrschichtige Material zu erstellen verwende ich diese beiden Befehle:

```
...
CreateMaterial(#MATERIAL_TERRAIN, TextureID(#TEXTUR_GLOBAL))
AddMaterialLayer(#MATERIAL_TERRAIN, TextureID(#TEXTUR_DETAIL), #PB_Material_Add)
...
```

Die erste Zeile erstellt ein einschichtiges Standardmaterial. Diesen Vorgang habe ich bereits im Invader Beispiel beschrieben. Die zweite Zeile lädt eine weitere Textur und fügt diese als zweite Lage dem zuvor erstellten Material hinzu. Der 'AddMaterialLayer()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Material → AddMaterialLayer) benötigt drei Parameter. Der erste ist die PB Nummer des Materials dem Sie eine weitere Lage hinzufügen wollen. Der zweite Parameter ist der OGRE Identifikator der Textur die Sie als weitere Lage hinzufügen möchten. Der dritte Parameter ist der Mischmodus, der bestimmt wie die neue Textur mit dem vorhandenen Material verschmolzen wird. Dieser Parameter kann eine der vier eingebauten Konstanten sein:

'#PB_Material_Add'
Verschmilzt die neue Materiallage mit der existierenden mittels einer 'Hinzufügen' Operation.

'#PB_Material_Replace'
Verschmilzt die neue Materiallage mit der Existierenden mittels einer 'Ersetzen' Operation.

'#PB_Material_Add'
Verschmilzt die neue Materiallage mit der existierenden mittels einer 'Hinzufügen' Operation. Dieser Modus wertet zusätzlich alle in der Textur gefundenen Alpha Informationen mit aus. Dies ist zur Zeit nur bei Texturen in den Bildformaten PNG und TGA möglich.

'#PB_Material_Modulate'
Verschmilzt die neue Materiallage mit der existierenden mittels einer 'Modulieren' Operation.

Wenn Sie Ihr mehrschichtiges Material erstellt haben sind Sie bereit ein Terain mittels des 'CreateTerrain()' Befehls (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Terrain → CreateTerrain) zu erstellen. Im 'First Person Kamera' Beispiel habe ich das folgendermaßen gemacht:

```
...
CreateTerrain("Terrain.png", MaterialID(#MATERIAL_TERRAIN), 1, 2, 1)
...
```

Der 'CreateTerrain()' Befehl benötigt sechs Parameter. Der erste Parameter ist der Name eines Bildes, das als Höhenkarte für das Terrain verwendet wird. Der zweite Parameter ist der OGRE Identifikator des mehrschichtigen Materials, das als Terrain Oberfläche verwendet wird. Die Parameter drei, vier und fünf sind optional und dienen der Skalierung des Terrains im 'x, y, z' Koordinatensystem. Diese Parameter sind Multiplikatoren, das heißt wenn der Wert '1' verwendet wird bleibt der Wert auf dieser Achse so wie er ist. Wenn der Wert '2' ist, dann wird diese Achse verdoppelt, usw. Der sechste und letzte Parameter ist ein optionaler Tarrain Qualitätswert, der im Bereich von '1' bis '20' liegt. Je kleiner der übergebene Wert desto höher ist die Terrain Renderqualität und dementsprechend höher auch die CPU Belastung um diese zu zeichnen. Höhere Werte haben den umgekehrten Effekt.

Terrain Höhenkarte Details
Wenn Sie ein Höhenkarten Bild verwenden um ein Terrain zu generieren, dann müssen Sie einige Regeln beim erstellen des Bildes für die Höhenkarte beachten. Zuerst einmal muss das Bild quadratisch sein, da OGRE in PureBasic nur quadratische Terains erstellen kann (auch wenn Sie diese während der Erstellung skalieren können). Zweitens muss das Bild im 8 Bit Graustufen Format vorliegen um '256' Graustufen zur Verfügung zu stellen. Reines Schwarz entspricht einer Höhe von '0', während reines Weiß als eine Höhe von '255' interpretiert wird. Drittens muss das Bild in einer bestimmten Größe vorliegen, da die Größe auch die Anzahl der Dreiecke (Polygone) des im Terrain verwendeten Mesh vorgibt. Die Größe eines Höhenkarten Bildes entspricht der Größe einer Textur plus ein Pixel in jeder Dimension. Dadurch wird sichergestellt, das die Terrainhöhe vom Bild korrekt berechnet wird. Abb. 43 zeigt empfohlene Größen von Höhenkarten Bildern und gibt die dazugehörige Anzahl von Polygonen an, die zur Erstellung des Terrain Mesh verwendet werden.

Wenn Sie auf Abb. 43 schauen, sehen Sie wie die Anzahl der Polygone bei höheren Auflösung der Höhenkarte zunimmt. Eine höhere Auflösung der verwendeten Höhenkarte kann die Render Genauigkeit von der Höhenkarte zum Terrain erhöhen, allerdings zu Lasten der Gesamtleistung. Große Höhenkarten Auflösungen wie '1025 x 1025' werden wegen der hohen Polygonzahl normalerweise nicht verwendet.

Terrain Höhenkarten Bildgrößen

Breite in Pixel	Höhe in Pixel	Terrain Polygone
65	65	8192
129	129	32768
257	257	131072
513	513	524288
1025	1025	2097152

Alle Standard Texturgrößen sind Potenzen von '2' + '1'

Abb. 43

Terrains und die automatischen Detail-Level

Alle Terrains die in PureBasic mit OGRE erstellt wurden haben automatische und dynamische Detail-Level. Das bedeutet, wenn ein Terrain nicht durch eine Kamera angezeigt wird oder einige Terrainbereiche andere verdecken (wie zum Beispiel Hügel), dann werden die verdeckten Bereiche nicht gezeichnet. Außerdem nimmt mit der Entfernung der Kamera zu einem bestimmten Terrainbereich die Komplexität und damit auch die Polygonzahl des Terrainbereichs ab. Dadurch ist es leistungsschonender große Szenen zu zeichnen. Das können Sie im 'First Person Kamera' Beispiel sehr einfach beobachten. Fokussieren Sie einen bestimmten Hügel mit der Kamera, entfernen sich dann rückwärts von diesem Objekt und beobachten Sie seine Struktur. Sie werden feststellen, das der Hügel seine Komplexität reduziert je weiter Sie sich von ihm mit der Kamera entfernen.

Die 'Ich' Perspektive

Nachdem das Terrain erstellt wurde und das 'First Person Kamera' Beispiel läuft, verwende ich verschiedene Befehle um Informationen von der Maus und der Tastatur zu sammeln. Diese Informationen verwende ich um die Kamera kontinuierlich zu positionieren. Die Maus Befehle sind:

'MouseDeltaX()'
(Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Mouse → MouseDeltaX)
Dieser Befehl gibt die Anzahl der Pixel zurück die die Maus auf der 'x' Bildschirm Achse (links und rechts) seit dem letzten Durchlauf der Hauptschleife zurückgelegt hat.

'MouseDeltaY()'
(Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Mouse → MouseDeltaY)
Dieser Befehl gibt die Anzahl der Pixel zurück die die Maus auf der 'y' Bildschirm Achse (hoch und runter) seit dem letzten Durchlauf der Hauptschleife zurückgelegt hat.

Diese zwei zurückgegebenen Werte werden dann in den 'x' und 'y' Parametern des 'RotateCamera()' Befehls platziert um die 3D Kamera in Abhängigkeit von der Mausbewegung zu drehen. Hier der tatsächliche Code:

```
...
If ExamineMouse()
  MausYRotation - MouseDeltaX() / 10
  MausXRotation + MouseDeltaY() / 10
EndIf
RotateCamera(#KAMERA_EINS, MausXRotation, MausYRotation, 0)
...
```

Das erstellt unsere Mausgesteuerte Kamera. Beachten Sie das Minus Zeichen vor dem 'MouseDeltaX()' Befehl. Dieses invertiert den von diesem Befehl zurückgegebenen Wert. Wenn Sie sich ins Gedächtnis zurückrufen, Sie benötigen positive Werte für eine Bewegung nach links und negative Werte für eine Bewegung nach rechts. Der 'MouseDeltaX()' Befehl gibt diese Werte invertiert zurück, da er 2D Bildschirm Koordinaten an Stelle von Rotationswinkeln zurückgibt.

Um die Kamera in der 3D Welt herum zu schwenken und somit den Effekt zu erzeugen, wir würden über das Terrain wandern, habe ich folgenden Code verwendet:

```
...
If ExamineKeyboard()
  If KeyboardPushed(#PB_Key_Left)  : TasteX = - #BEWEGUNG_GESCHWINDIGKEIT : EndIf
  If KeyboardPushed(#PB_Key_Right) : TasteX =   #BEWEGUNG_GESCHWINDIGKEIT : EndIf
  If KeyboardPushed(#PB_Key_Up)    : TasteZ = - #BEWEGUNG_GESCHWINDIGKEIT : EndIf
```

```
    If KeyboardPushed(#PB_Key_Down)  : TasteZ =   #BEWEGUNG_GESCHWINDIGKEIT : EndIf
    MoveCamera(#KAMERA_EINS, TasteX, 0, TasteZ)
    TasteX              = 0
    TasteZ              = 0
    AktuelleKamXPos.f   = CameraX(#KAMERA_EINS)
    AktuelleKamZPos.f   = CameraZ(#KAMERA_EINS)
    WunschKameraHoehe.f = TerrainHeight(AktuelleKamXPos, AktuelleKamZPos) + 10
    CameraLocate(#KAMERA_EINS, AktuelleKamXPos, WunschKameraHoehe, AktuelleKamZPos)
  EndIf
  ...
```

Hier verwende ich den 'KeyboardPushed()' Befehl um zu prüfen ob eine Taste gedrückt wird. Wenn dies der Fall ist weise ich den relevanten Variablen einen Wert zu und verwende diese dann als Parameter im 'MoveCamera()' Befehl. Dadurch wird die Kamera durch die 3D Welt bewegt, allerdings wird hierbei nicht die Höhe des Terrains berücksichtigt.

Um die Kamera hoch und runter zu bewegen, so dass sie immer die gleiche Distanz zum Terrain hat, verwende ich den 'CameraLocate()' Befehl. Da dieser Befehl absolute Werte benötigt muss ich die aktuellen 'x' und 'z' Koordinaten der Kamera erfassen und übergebe diese dann wenn ich den 'y' Wert verändere. um die aktuellen Kamera Koordinaten zu ermitteln verwende ich die Befehle 'CameraX()' und 'CameraZ()' (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Camera). Diese beiden Befehle benötigen jeweils nur einen Parameter. Es handelt sich um die PB Nummer der Kamera von der die Werte gelesen werden sollen. Wenn ich diese Werte ermittelt habe muss ich herausfinden wie hoch das Terrain an dieser Stelle ist. Um diese zu ermitteln verwende ich den 'TerrainHeight()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Terrain → TerrainHeight). Dieser Befehl benötigt zwei Parameter. Es handelt sich hierbei um die 'x' und 'z' Koordinaten des Punktes im Terrain von dem wir die Höhe ermitteln wollen. Wenn ich diesen Wert ermittelt habe füge ich diesem noch '10' hinzu um die Kamera ein wenig vom Boden zu heben, und habe damit alle drei 3D Koordinaten die ich für den 'LocateCamera()' Befehl benötige. Während jedem Durchlauf der Hauptschleife wird die 'y' Position der Kamera neu berechnet, um die Kamera Höhe bei der Bewegung über das Terrain ständig anzupassen.

Ein wenig fortgeschrittener

In diesem Abschnitt werde ich ein wenig über die etwas fortgeschritteneren Funktionen von OGRE und PureBasic erzählen. Diese Funktionen enthalten Partikel und ein Material Skript. Partikel werden in 3D Engines für viele verschiedene Dinge eingesetzt, hauptsächlich für zufällige visuelle Effekte wie Schnee, Regen, Feuer und Rauch. Ich habe sie im folgenden Beispiel zum simulieren eines Feuers verwendet. Material Skripte sind eine Möglichkeit alle Material Eigenschaften in einer Material Skriptdatei zu kapseln. Dadurch wird Ihr PureBasic Code sehr viel übersichtlicher und leichter lesbar. Material Skripte erlauben es Ihnen einen Mesh zu laden der bereits eine Referenz zu einem Material enthält und wenn das Material in einer Skriptdatei auftaucht, dann verwendet der Mesh dieses Material ohne weitere Anweisungen in Ihrem Quelltext. Damit wird eine einfache Handhabbarkeit und Modifizierbarkeit sichergestellt, da bestimmte fortgeschrittene Funktionen nur durch die Verwendung von Skripten erreichbar sind. Im folgenden Beispiel habe ich ein Materialskript verwendet in dem ich definiere, dass der geladene Mesh 'Kugel Textur-Mapping' verwenden soll - eine Funktion die ohne Skript nicht verfügbar ist. Hier der Programmcode:

```
Enumeration
  #MESH
  #TEXTUR
  #MATERIAL
  #ENTITY
  #KAMERA_EINS
  #LICHT_EINS
  #LICHT_ZWEI
  #PARTIKEL_EINS
EndEnumeration

;Setzen der Breite, Höhe und Farbtiefe des Screens
;Aufgrund der Seitenbreite wurden hier abgekürzte Variablennamen verwendet :(
Global ScrB.i = 1024
Global ScrH.i = 768
Global ScrT.i = 32
;Andere Globale Variablen
Global Beenden.i = #False

;Einfache Prozedur zur Fehlerprüfung
Procedure Fehlerbehandlung(Ergebnis.i, Text.s)
```

```
  If Ergebnis = 0
    MessageRequester("Fehler", Text, #PB_MessageRequester_Ok)
    End
  EndIf
EndProcedure

;Konvertiere Grad in Bogenmaß
Procedure.f GradZuRad(Winkel.f)
  ProcedureReturn Winkel.f * #PI / 180
EndProcedure

;Initialisiere Umgebung
Fehlerbehandlung(InitEngine3D(), "InitEngine3D() fehlgeschlagen.")
Fehlerbehandlung(InitSprite(), "InitSprite() fehlgeschlagen.")
Fehlerbehandlung(OpenScreen(ScrB, ScrH, ScrT, ""), "Kann Screen n. öffnen.")
Fehlerbehandlung(InitKeyboard(), "InitKeyboard() fehlgeschlagen.")
SetFrameRate(60)

Add3DArchive("Data\", #PB_3DArchive_FileSystem)
Parse3DScripts()
CreateEntity(#ENTITY, LoadMesh(#MESH, "Statue.mesh"), #PB_Material_None)

LoadTexture(#TEXTUR, "Flamme.png")
CreateMaterial(#MATERIAL, TextureID(#TEXTUR))
DisableMaterialLighting(#MATERIAL, 1)
MaterialBlendingMode(#MATERIAL, #PB_Material_Add)

CreateParticleEmitter(#PARTIKEL_EINS, 2, 2, 0, #PB_Particle_Point, 12.9, 69, 15.7)
ParticleSize(#PARTIKEL_EINS, 5, 5)
ParticleMaterial(#PARTIKEL_EINS, MaterialID(#MATERIAL))
ParticleEmissionRate(#PARTIKEL_EINS, 50)
ParticleTimeToLive(#PARTIKEL_EINS, 0.25, 0.25)
ParticleColorRange(#PARTIKEL_EINS, RGB(255, 0, 0), RGB(255, 200, 0))
ParticleVelocity(#PARTIKEL_EINS, 1, 10)

CreateLight(#LICHT_EINS, RGB(255, 255, 255))
CreateLight(#LICHT_ZWEI, RGB(255, 200, 0), 12.9, 72, 15.7)
CreateCamera(#KAMERA_EINS, 0, 0, 100, 100)

;Hauptschleife
Repeat

  Winkel.f + 0.5
  PosX.f =  75 * Sin(GradZuRad(Winkel))
  PosY.f = (50 * Sin(GradZuRad(Winkel / 2))) + 65
  PosZ.f =  75 * Cos(GradZuRad(Winkel))
  LightLocate(#LICHT_EINS, PosX, PosY + 100, PosZ)
  LightColor(#LICHT_ZWEI, RGB(255, Random(200), 0))
  CameraLocate(#KAMERA_EINS, PosX, PosY, PosZ)
  CameraLookAt(#KAMERA_EINS, 0, 60, 0)
  RenderWorld()
  FlipBuffers()

  ExamineKeyboard()
  If KeyboardReleased(#PB_Key_Escape)
    Beenden = #True
  EndIf
Until Beenden = #True
End
```

Dieses Beispiel sollte sehr gut nachvollziehbar sein, da es dem Stil der letzten paar 3D Beispiele folgt. Zuerst initialisieren wir die Umgebung, öffnen einen Screen und spezifizieren ein 3D Archiv. Nachdem dies erledigt ist verwende ich einen neuen Befehl mit Namen 'ParseScripts()'. Dieser Befehl schaut beim Aufruf in alle definierten 3D Archive und liest alle darin enthaltenen Skript Dateien. Es gibt viele Arten von OGRE Skript Dateien, zurzeit unterstützt PureBasic allerdings nur Material Skripte.

Material Skripte

Nachdem der 'ParseScripts()' Befehl aufgerufen wurde werden alle Skripte aus allen spezifizierten 3D Archiven gelesen und analysiert. Das heißt das jedes Skript gelesen wird um festzustellen welche Material Definitionen es enthält. Danach werden die ganzen Materialeigenschaften in den Speicher eingelesen. Wenn

dann später ein Mesh geladen wird, der ein Material mit dem gleichen Namen verwendet wie im Material Skript definiert, dann verwendet er die Texturen und Eigenschaften die im Material Skript definiert sind. Der zu verwendende Materialname kann vom 3D Modellierungsprogramm direkt mit dem Mesh gespeichert werden. Das erlaubt es Ihnen bereits bei der Erstellung eines Mesh ein Material zu spezifizieren. Diesen können Sie dann einfach in Ihrem PureBasic 3D Programm laden, wobei er alle seine Materialeigenschaften behält.

Material Skripte können per Hand mit einem einfachen Texteditor erstellt werden oder Sie können von einigen 3D Programmen zusammen mit dem Mesh ausgegeben werden. Die Syntax von Material Skriptdateien wird auf der OGRE Webseite beschrieben. Einen Link zu dieser Seite finden Sie im Anhang A (Nützliche Internet Links). Hier ist das Material Skript das ich für das Statue Modell im Statue Beispiel verwendet habe. Das Skript heißt 'Statue.material':

```
material Statue
{
  technique
  {
    pass
    {
      texture_unit
      {
        texture KugelKarte.png
        env_map spherical
        filtering trilinear
      }
    }
  }
}
```

Diese einfache Datei beschreibt das Material der Statue, genauer gesagt informiert es OGRE darüber, dass das Bild 'KugelKarte.png' als Textur verwendet werden soll. Weiterhin soll 'Kugel Textur-Mapping' und trilineare Filterung verwendet werden. Damit der Statue-Mesh dieses Material verwendet habe ich im 3D Programm bereits bei seiner Erstellung den Namen dieser Datei spezifiziert.

Wenn Sie auf das Statue Beispiel schauen können Sie sehen, das ich ein volltexturiertes 3D Objekt mit einer einzigen Zeile erzeugt habe, anders als im Invader Beispiel, wo viele Codezeilen für das laden der Texturen und das erzeugen des Materials nötig waren. Diese Zeile ist:

```
...
CreateEntity(#ENTITY, LoadMesh(#MESH, "Statue.mesh"), #PB_Material_None)
...
```

Diese Zeile lädt die 'Statue.mesh' Datei und verwendet das im Materialskript definierte Material. Da wir ein Materialskript zur Materialdefinition verwenden muss als Material ID Parameter im 'CreateEntity()' Befehl die eingebaute Konstante '#PB_Material_None' übergeben werden. Diese Konstante stellt sicher, dass dem Entity kein Material zugewiesen wird das nicht im Skript definiert ist.

Partikel Effekte

Das Erstellen von Partikeln kann eine komplizierte Geschichte sein, gerade deshalb weil man so viele Effekte mit ihnen erzeugen kann. Momentan unterstützt PureBasic neunzehn Befehle um benutzerdefinierte Partikel auf nahezu jede erdenkliche Weise zu erstellen.

Partikel Skripte?

Wie Materialskripte sind Partikelskripte eine Möglichkeit komplizierte Eigenschaften unter einem einfachen Namen zu bündeln. Wenn Sie dann das nächste mal einen Partikel Emitter erstellen können Sie einfach den entsprechenden Namen angeben und alle unter diesem Namen definierten Eigenschaften werden auf den Emitter angewandt. Das macht das testen und erstellen von Partikel Effekten schneller und einfacher.

Zur Zeit unterstützt PureBasic keine Partikel Skripte, die OGRE Engine allerdings schon. Auf Nachfrage antwortete das PureBasic Team dass die Unterstützung für Partikel Skripte in der näheren Zukunft eingebaut werden soll. Vielleicht hat PureBasic zu der Zeit in der Sie das lesen bereits diese Unterstützung. Für weitere Informationen schauen Sie auf die OGRE Webseite und ins PureBasic Forum. Beide Adressen finden Sie im Anhang A (Nützliche Internet Links).

Partikel brauchen wie Meshes eine Material Zuordnung damit sie richtig angezeigt werden, und ich habe im Statue Beispiel eine erstellt, indem ich das Bild 'Flamme.png' als Textur verwendet habe. Dieses Bild wird für jeden einzelnen Partikel verwendet der den Emitter verlässt. Ein Partikel Emitter ist eine Koordinate im 3D Raum. Von diesem Punkt werden alle Partikel ausgestoßen. In meinem Beispiel habe ich diesen folgendermaßen definiert:

```
...
CreateParticleEmitter(#PARTIKEL_EINS, 2, 2, 0, #PB_Particle_Point, 12.9, 69, 15.7)
...
```

Der 'CreateParticleEmitter()' Befehl (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Particle → CreateParticleEmitter) benötigt acht Parameter. Der erste ist wie üblich die PB Nummer die mit dem Partikelstrahler verknüpft wird. Die Parameter zwei, drei und vier definieren wie weit vom Mittelpunkt die Partikel entlang der 'x', 'y' und 'z' Achse ausgestrahlt werden. Der fünfte Parameter definiert den Partikel Modus, welcher '#PB_Particle_Point' oder '#PB_Particle_Box' sein kann. Emitter Punkte werden üblicherweise für Feuer und Rauch, usw. benutzt, also dann wenn die Partikel von einem bestimmten Punkt ausgehen, während Emitter Boxen üblicherweise für Flächeneffekte wie Regen oder Schnee verwendet werden. Die letzten drei Parameter definieren den Ort des Emitters im Koordinatensystem der 3D Welt. Diese drei Parameter sind optional. Wenn sie nicht angegeben werden wir der Strahler an den Koordinaten '0, 0, 0' platziert. Sie können den Emitter später noch mit dem 'ParticleEmitterLocate()' Befehl bewegen.

Die anderen im Statue Beispiel verwendeten Partikel Befehle dienen zur Konfiguration des Partikelstrahlers um den gewünschten Effekt zu erzielen. Diese Befehle sind:

```
...
ParticleSize(#PARTIKEL_EINS, 5, 5)
ParticleMaterial(#PARTIKEL_EINS, MaterialID(#MATERIAL))
ParticleEmissionRate(#PARTIKEL_EINS, 50)
ParticleTimeToLive(#PARTIKEL_EINS, 0.25, 0.25)
ParticleColorRange(#PARTIKEL_EINS, RGB(255, 0, 0), RGB(255, 200, 0))
ParticleVelocity(#PARTIKEL_EINS, 1, 10)
...
```

All diese Befehle sind selbsterklärend und alle sind imstande Tausende von verschiedenen Effekten zu erzeugen. Detailliertere Informationen zu diesen Befehlen finden Sie in der PureBasic Hilfe (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Particle).

Blick auf einen bestimmten Punkt

Im Statue Beispiel habe ich einen neuen Kamera Befehl verwendet der sehr nützlich sein kann. Es ist der Befehl 'CameraLookAt()' (Hilfe Datei: Referenz-Handbuch → 3D Spiele & Multimedia Libraries → Camera → CameraLookAt). Dieser Befehl ermöglicht es die Perspektive der Kamera im 3D Raum mit nur einer Anweisung zu ändern. In meinem Beispiel habe ich den Befehl folgendermaßen verwendet:

```
...
CameraLookAt(#KAMERA_EINS, 0, 60, 0)
...
```

Der erste Parameter ist die PB Nummer der Kamera die Sie drehen möchten. Die Parameter zwei, drei und vier geben die 'x, y, z' Koordinaten an auf die Sie mit der Kamera schauen wollen. In meinem Beispiel blickt die Kamera direkt in die Mitte der Statue, während die Kamera um die Statue rotiert.

Dynamische Beleuchtung

Im Statue Beispiel habe ich auch ein paar Beleuchtungs Tricks angewandt die dabei helfen die Szene etwas aufzuwerten. Der erste ist eine weiße Lichtquelle die immer der Kamera folgt. dadurch ist der Statue Mesh immer voll beleuchtet. Als zweiten Beleuchtungs Trick habe ich ein flackerndes Licht etwas oberhalb des Partikel Emitters platziert. Diese Lichtquelle ändert mit jedem Durchlauf der Hauptschleife seine Farbe. Dazu habe ich folgenden Befehl in der Hauptschleife platziert:

```
...
LightColor(#LICHT_ZWEI, RGB(255, Random(200), 0))
...
```

Dieser Befehl ändert die spezifizierte Lichtfarbe bei jedem Aufruf. In der Hauptschleife des Statue Beispiels habe ich die Grün Komponente der Lichtquelle mit einem Zufallsgenerator versehen, der diesen Teil der Farbe mit Werten zwischen '0' und '200' versorgt. Da dir rote Komponente auf '255' gesetzt ist und der blau

Anteil '0' ist schwankt die Farbe zwischen rot und gelb. nachdem das Programm gestartet wurde können Sie das Farbflackern auf der Oberfläche der Statue beobachten.

Mit wenigen Tricks, wie Material Skripte zum erstellen von komplexeren Materialien und die Verwendung von Partikeln zum Erzeugen von erstaunlichen Effekten, können Sie Ihre Demo oder Ihr Spiel erheblich in der Qualität und im Stil aufwerten. Ich habe Ihnen nur einen kleinen Teil von dem gezeigt was mit diesen Befehlen möglich ist. Jetzt liegt es an Ihnen ob Sie mehr über diese Befehle lernen möchten und versuchen Ihre eigenen 3D Programme zu schreiben. Noch ein Tip zum Abschluss: Experimentieren Sie immer nur mit wenigen Einstellungen, besonders bei den Partikel Befehlen.

Was kommt als nächstes?

Ich hoffe dass dies ein hilfreiches Kapitel war, wenn auch nur in Kurzform, das Sie bei der Programmierung von 3D Grafiken mit PureBasic unterstützt. Ich hoffe dass Sie angeregt wurden mehr darüber zu erfahren was mit PureBasic unter Verwendung der OGRE Engine alles möglich ist. Ich würde Ihnen empfehlen den Abschnitt '3D Spiele & Multimedia Libraries' in der PureBasic Hilfe vollständig durchzulesen, damit Sie verstehen was PureBasic in diesem Bereich alles zu bieten hat. Dieses Kapitel hat nicht jeden Befehl abgedeckt der in diesem Bereich enthalten ist, da dies den Rahmen dieses Buches sprengen würde. Ich hoffe aber dass ich Ihnen eine gute Einführung gegeben habe, den Rest schaffen Sie nun selbst.

Das Erlernen der Programmierung von 3D Grafiken kann sehr kniffelig sein, da es manchmal auf komplizierten Mathematischen Routinen beruht und auch gute 3D Moddelierungs Fähigkeiten voraussetzt. Die in den Beispielen verwendete Mathematik ist noch relativ einfach, so dass sie von jedem Abiturienten verstanden werden sollte. Ich würde jedem Anfänger in der 3D Programmierung raten sich zuerst mit der dazugehörigen Mathematik zu beschäftigen. Es wird Zeiten geben, in denen Sie ein 3D Projekt abbrechen müssen, da ihre mathematischen Fähigkeiten nicht an Ihre Programmierfähigkeiten heranreichen. Deshalb ist es das Beste wenn Sie vorbereitet sind.

Werfen Sie einen Blick ins Internet, da dort Hunderte von Anleitungen für Mathematik im 3D Bereich zu finden sind die Sie selbständig durcharbeiten können. Weiterhin gibt es unzählige Bücher zum lesen die diese Punkte im Detail aufgreifen.

12. Sound

An bestimmten Stellen in Ihrem Programm müssen Sie vielleicht einmal einen Klang abspielen. Das kann eine akustische Glocke sein, die signalisiert dass ein Prozess abgeschlossen ist oder ein Klangeffekt in einem Spiel. Egal für was Sie Klänge brauchen, Sie müssen Wissen wie Sie bestimmte Sounddateien laden und abspielen können. Dieses Kapitel beschreibt wie Sie verschiedene Sounddateien laden und liefert Beispiele wie Sie diese in einem Programm abspielen.

Wave Dateien

Das Wave Format ist eines der gängigsten Soundformate auf Personal Computern, es entstand in einer Zusammenarbeit von Microsoft und IBM. Wave Dateien mit der Dateiendung '*.wav' sind das ursprüngliche Soundformat, das von allen PC's verwendet wird. Obwohl dieses Format keine Kompression für die Sounddaten besitzt, wird es doch für alltägliche Zwecke eingesetzt.

Das folgende Beispiel lädt eine Wave Datei mit dem Namen 'Intro.wav' und spielt diese ab:

```
#SOUND_DATEI = 1
If InitSound()

  LoadSound(#SOUND_DATEI, "Intro.wav")
  PlaySound(#SOUND_DATEI)

  StartZeit.i = ElapsedMilliseconds()
  Repeat
    Delay(1)
  Until ElapsedMilliseconds() > StartZeit + 8000
EndIf
End
```

Dieses Beispiel würde in einem echten Programm nicht verwendet werden, aber es zeigt die erforderlichen Schritte zum abspielen einer Wave Datei. Zuerst müssen wir die Sound Umgebung mit dem 'InitSound()' Befehl initialisieren. Damit wird die zum abspielen der Datei benötigte Hardware und Software korrekt initialisiert. Wenn dieser Befehl True zurückgibt wissen wir dass alles richtig initialisiert wurde und wir können fortfahren. Wenn die Initialisierung fehlschlägt gibt es keine Möglichkeit mit Sound fortzufahren, im Computer ist wahrscheinlich keine Soundkarte installiert.

Wenn die Initialisierung erfolgreich war laden wir die Sound Datei mit dem 'LoadSound()' Befehl (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sound → LoadSound). Dieser Befehl lädt die Sound Datei in den Speicher, bereit zum abspielen. Der 'LoadSound()' Befehl benötigt zwei Parameter, wovon der erste die PB Nummer ist mit der Sie die Sound Datei verknüpfen wollen und der zweite Parameter ein String ist der den Dateinamen der zu ladenden Datei enthält.

Wenn die Wave Datei geladen wurde, können Sie sie jederzeit in Ihrem Programm mit dem 'PlaySound()' Befehl abspielen (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sound → PlaySound). Dieser Befehl benötigt einen Parameter. Hierbei handelt es sich um die PB Nummer der Sound Datei die Sie abspielen möchten.

Wenn Sie etwas genauer auf das Wave Datei Beispiel schauen sehen Sie, dass ich eine gut durchdachte Schleife am Ende des Programms eingefügt habe. Diese verhindert das vorzeitige Programmende. Wenn das Programm endet werden alle zur Zeit abgespielten Sounds beendet und aus dem Speicher entfernt. Da ich das verhindern möchte habe ich diese Schleife konstruiert, die dem Programm acht Sekunden Zeit zum abspielen der Sound Datei lässt und dann das Programm beendet. Diese Konstruktion werden Sie in keinem echten Programm finden, da ein Solches eine Hauptschleife hat die das Programm am 'leben' hält während die Sound Datei abgespielt wird.

Einbinden von Wave Dateien

Manchmal möchten Sie vielleicht keine externe Wave Datei laden sondern diese Datei in Ihr Programm einbinden, so das Sie alle Bestandteile Ihres Programms in einer Datei weitergeben können. Das können Sie erreichen indem Sie die Wave Datei in einer 'DataSection' einbinden, ähnlich dem Einbetten eines Bildes was ich in Kapitel 9 (Einbinden von Grafiken in Ihr Programm) erklärt habe. Es gibt allerdings einen Unterschied, an Stelle von 'CatchImage()' verwenden Sie 'CatchSound()' um die Datei aus dem 'Data' Bereich zu laden. Hier ein Beispiel:

```
#SOUND_DATEI = 1
If InitSound()

  CatchSound(#SOUND_DATEI, ?SoundDatei)
  PlaySound(#SOUND_DATEI)

  StartZeit.i = ElapsedMilliseconds()
  Repeat
    Delay(1)
  Until ElapsedMilliseconds() > StartZeit + 8000
  End
EndIf

DataSection
  SoundDatei:
  IncludeBinary "Intro.wav"
EndDataSection
```

Die Sound Datei wird mit dem 'IncludeBinary' Befehl auf die exakt gleiche Weise in die 'DataSection' eingebunden wie die Bilddatei. Die Sprungmarke 'SoundDatei:' markiert den Beginn der Datei im Speicher. Um die Datei zur Laufzeit des Programms aus der 'DataSection' zu laden verwenden wir den 'CatchSound()' Befehl (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sound → CatchSound). Dieser Befehl benötigt zwei Parameter. Der erste ist die PB Nummer die dem Sound zugewiesen wird, der zweite ist die Adresse im Speicher von der der Sound geladen werden soll. Diese Adresse ist die Adresse der Sprungmarke, da diese den Beginn der Sound Datei im Speicher markiert. Um die Adresse einer beliebigen Sprungmarke zu ermitteln verwenden wir ein Fragezeichen vor dem Sprungmarken Namen, zum Beispiel '?SoundDatei'. Beachten Sie dass wir den Doppelpunkt nach dem Sprungmarken Namen nicht benötigen. Nachdem Sie eine Wave Datei auf diese Weise 'geangelt' haben können Sie diese wie jede andere Wave Datei verwenden. In diesem Fall habe ich die Wave Datei mit dem 'PlaySound()' Befehl abgespielt.

Sie können auf diese Weise mehrere Wave Dateien in Ihr Programm einbinden, Sie müssen nur darauf achten das Sie jeder Datei eine einmalige Sprungmarke zuweisen damit diese mit dem 'CatchSound()' Befehl gefunden werden kann.

Manipulieren von Sounds in Echtzeit

Die Verwendung der Befehle aus der PureBasic Sound Bibliothek ermöglichen Ihnen die Lautstärke zu verändern, die Balance zu variieren und die Frequenz zu verschieben. Lautstärke erhöht oder verringert die Lautstärke des Sound, Balance schwenkt den Sound von einem Lautsprecher zum anderen, üblicherweise nach links oder nach rechts und das Ändern der Frequenz wirkt sich in einer Beschleunigung oder Abbremsung der Abspielgeschwindigkeit aus. Um diese Effekte zu demonstrieren habe ich ein kleines Sound Player Programm erstellt, das es ihnen in Echtzeit ermöglicht die Lautstärke, die Balance und die Frequenz zu verändern. Probieren Sie es selbst aus. Öffnen Sie das Programm, laden Sie eine Wave Datei, drücken Sie die 'Spiele Datei' Schaltfläche und verändern Sie die einzelnen Regler.

```
Enumeration
  #FENSTER_HAUPT
  #SOUND_DATEI
  #TEXT_DATEI
  #KNOPF_DATEI_WAEHLEN
  #TEXT_LAUTSTAERKE
  #TRACKBAR_LAUTSTAERKE
  #TEXT_BALANCE
  #TRACKBAR_BALANCE
  #TEXT_FREQUENZ
  #TRACKBAR_FREQUENZ
  #KNOPF_SPIELE_DATEI
  #KNOPF_STOPPE_DATEI
EndEnumeration

Global DateiName.s = ""

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER_HAUPT, 0, 0, 500, 250, "Sound Player", #FLAGS)
  TextGadget(#TEXT_DATEI, 10, 10, 480, 20, "", #PB_Text_Border)
  ButtonGadget(#KNOPF_DATEI_WAEHLEN, 10, 40, 150, 20, "Wähle Wave Datei...")

  TextGadget(#TEXT_LAUTSTAERKE, 10, 70, 480, 20, "Lautstärke")
```

```
    TrackBarGadget(#TRACKBAR_LAUTSTAERKE, 10, 90, 480, 20, 0, 100)
    SetGadgetState(#TRACKBAR_LAUTSTAERKE, 100)

    TextGadget(#TEXT_BALANCE, 10, 120, 480, 20, "Balance")
    TrackBarGadget(#TRACKBAR_BALANCE, 10, 140, 480, 20, 0, 200)
    SetGadgetState(#TRACKBAR_BALANCE, 100)

    TextGadget(#TEXT_FREQUENZ, 10, 170, 480, 20, "Frequenz")
    TrackBarGadget(#TRACKBAR_FREQUENZ, 10, 190, 480, 20, 100, 10000)
    SetGadgetState(#TRACKBAR_FREQUENZ, 4400)

    ButtonGadget(#KNOPF_SPIELE_DATEI, 10, 220, 100, 20, "Spiele Datei")
    ButtonGadget(#KNOPF_STOPPE_DATEI, 130, 220, 100, 20, "Stoppe Sound")

    If InitSound()
      Repeat
        Ereignis.i = WaitWindowEvent()
        Select Ereignis
          Case #PB_Event_Gadget

            Select EventGadget()
              Case #KNOPF_DATEI_WAEHLEN
                DateiName=OpenFileRequester("Wählen","","Wave Datei (*.wav)|*.wav", 0)
                If DateiName <> ""
                  SetGadgetText(#TEXT_DATEI, GetFilePart(DateiName))
                  LoadSound(#SOUND_DATEI, DateiName)
                EndIf

              Case #TRACKBAR_LAUTSTAERKE
                If DateiName <> ""
                  SoundVolume(#SOUND_DATEI, GetGadgetState(#TRACKBAR_LAUTSTAERKE))
                EndIf

              Case #TRACKBAR_BALANCE
                If DateiName <> ""
                  SoundPan(#SOUND_DATEI, GetGadgetState(#TRACKBAR_BALANCE) - 100)
                EndIf

              Case #TRACKBAR_FREQUENZ
                If DateiName <> ""
                  SoundFrequency(#SOUND_DATEI, GetGadgetState(#TRACKBAR_FREQUENZ) * 10)
                EndIf

              Case #KNOPF_SPIELE_DATEI
                If DateiName <> ""
                  PlaySound(#SOUND_DATEI)
                EndIf

              Case #KNOPF_STOPPE_DATEI
                If DateiName <> ""
                  StopSound(#SOUND_DATEI)
                EndIf
            EndSelect

        EndSelect
      Until Ereignis = #PB_Event_CloseWindow
    EndIf
  EndIf
  End
```

Dieses Beispiel führt drei neue Sound Befehle ein:

'SoundVolume()'
Dieser Befehl wird verwendet um die Lautstärke des geladenen Sound zu regeln. Er verändert in keiner Weise die Original Sound Datei, er ändert lediglich die Lautstärke beim abspielen der Sound Datei. Zum verwenden dieses Befehles müssen Sie ihm zwei Parameter übergeben. Der erste Parameter ist die PB Nummer des Sound den Sie verändern möchten und der zweite gibt das Lautstärke Niveau an. Diese kann Werte von '0' bis '100' annehmen, wobei '0' Stille repräsentiert und '100' die lauteste Wiedergabe bedeutet.

'SoundPan()'
Dieser Befehl schwenkt den Sound zum und vom linken und rechten Kanal und benötigt zwei Parameter. Der erste Parameter ist die PB Nummer des Sound den Sie schwenken möchten und der zweite Parameter ist der Schwenk-Wert. Dieser Wert kann im Bereich von '-100' bis '100' liegen. Wenn Sie einen Wert von '-100' verwenden dann ist der Sound vollständig auf die linke Seite geschwenkt, bei Verwendung von '100' wird der Sound vollständig auf die rechte Seite geschwenkt.

'SoundFrequency()'
Dieser Befehl ändert die Frequenz des abgespielten Sound. Die Frequenz des Sound wird in Hertz gemessen und beschreibt wie oft eine bestimmte Wellenform in der Sekunde gelesen wird. Die Musik auf einer CD wurde zum Beispiel mit einer Samplerate von 44,1 kHz (Kilo Hertz) abgespeichert. Das bedeutet dass die Wellenform die die Soundinformation enthält vierundvierzigtausendeinhundert mal pro Sekunde abgetastet wurde. Das stellt sicher dass eine ausreichend hohe Auflösung vorhanden ist um selbst die geringste Variation im Sound darstellen zu können. Wenn ein Sound mit 44,1 kHz encodiert ist und Sie verwenden diesen Befehl um die Frequenz auf 22.050 Hz zu ändern, dann wird der Sound nur mit der Hälfte der Original Geschwindigkeit abgespielt. Um in PureBasic mit diesem Befehl die Frequenz eines geladenen Sound zu verändern müssen Sie ihm zwei Parameter übergeben. Der erste Parameter ist die PB Nummer des Sound den Sie verändern möchten und der zweite Parameter ist ein numerischer Wert der die neue Frequenz in Hertz angibt. Der Wert des zweiten Parameters muss zwischen '1.000' und '100.000' liegen.

Um mehr über die Befehle zum manipulieren von Wave Dateien zu erfahren schauen Sie in der PureBasic Hilfe im Bereich der Sound Bibliothek nach (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Sound).

Modul Dateien

Diese Dateien repräsentieren Musik in Form von digitalen Mustern. Intern speichern sie mehrere Seiten mit Musikmustern, ähnlich einem Arbeitsblatt. Diese Muster enthalten die Notenwerte, Instrumentennummern und Steuernachrichten die dem Programm mitteilen welche Samples verwendet werden und wie lange diese gespielt werden. Modul Dateien enthalten weiterhin eine Liste die definiert in welcher Reihenfolge die Muster abgespielt werden. Die Anzahl der gleichzeitig spielbaren Instrumente hängt von der Anzahl der Spuren in den Mustern ab. Die ersten verfügbaren Programme die dem Benutzer ermöglichten eigene Moduldateien zu erstellen verwendeten vier Spuren. Der größte Vorteil von Modul Dateien gegenüber Standard Sound Dateien (wie z.B. MIDI Dateien) liegt darin, dass sie ihre eigenen Audio Samples einbinden und deshalb überall gleich klingen.

Modul Dateien werden oft auch als 'Tracker Module' bezeichnet und die Kunst Module zu komponieren wird als 'Tracking' bezeichnet. Das ist historisch bedingt, da das erste Programm das dem Benutzer ermöglichte Modul Dateien zu erzeugen den Namen 'Soundtracker' hatte. Das Programm, obwohl es Anfangs schlecht angenommen wurde, wurde irgendwann als Public Domain herausgegeben und danach viele Male kopiert. Dadurch wuchs der Funktionsumfang und es entstanden verschiedene neue Produkte wie 'Noisetracker' oder 'Protracker'. Diese wurden sehr populär, besonders bei den Spiele und Demo Herstellern auf dem Commodore Amiga. Programme die Module erstellen werden heute kollektiv als 'Tracker' bezeichnet.

Modul Dateien können viele verschiedene Dateiendungen haben, da sie in vielen verschiedenen Formaten vorkommen. Diese Dateiendungen entschlüsseln häufig das Programm mit dem das Modul erstellt wurde. PureBasic unterstützt unter anderem folgende verbreitete Modulformate:

FastTracker ('*.xm')
Scream Tracker ('*.s3m')
Protracker ('*.mod')
Impulse Tracker ('*.it')

Diese unterschiedlichen Modulformate werden in Ihrem PureBasic Programm auf die gleiche Art und Weise geladen und abgespielt. Hier ein Beispiel das demonstriert wie Sie jede beliebige unterstützte Modul Datei laden und abspielen können:

```
#MODUL_DATEI = 1

If InitSound()

  If LoadModule(#MODUL_DATEI, "Eighth.mod")

    PlayModule(#MODUL_DATEI)
```

```
    StartZeit.i = ElapsedMilliseconds()
    Repeat
      Delay(1)
    Until ElapsedMilliseconds() > StartZeit + 15000

    StopModule(#MODUL_DATEI)

  EndIf
EndIf
End
```

Zuerst müssen wir wie im Wave Beispiel die Sound Umgebung mit dem 'InitSound()' Befehl initialisieren. Danach müssen wir die Abspielfähigkeit von PureBasic für Modul Dateien initialisieren. Hierzu verwenden wir den 'InitModule()' Befehl. Beide Befehle sollten auf korrekte Initialisierung geprüft werden.

Nachdem die Umgebung initialisiert ist können wir mit dem 'LoadModule()' Befehl (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Module → LoadModule) ein Modul laden. Dieser Befehl benötigt zwei Parameter. Der erste ist die PB Nummer die Sie dem Modul zuweisen möchten und der zweite ist der Dateiname der Modul Datei die Sie laden möchten.

Wenn das Modul geladen wurde können wir es mit dem 'PlayModule()' Befehl abspielen (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Module → PlayModule). Wie der 'PlaySound()' Befehl benötigt dieser einen Parameter, nämlich die PB Nummer des Moduls das Sie abspielen möchten. Zum beenden des Abspielvorgangs können Sie den Befehl 'StopModule()' verwenden.

Einbinden von Modul Dateien

Wie Wave Dateien können Sie auch Modul Dateien in Ihr fertiges Programm einbinden. Auch hierzu wird das Modul in einer 'Datasection' abgelegt und wird dann aus dem Speicher geladen. Hier ein Beispiel:

```
#MODUL_DATEI = 1

If InitSound()

  If CatchModule(#MODUL_DATEI, ?ModulDatei, ?ModulDateiEnde - ?ModulDatei)

    PlayModule(#MODUL_DATEI)

    StartZeit.i = ElapsedMilliseconds()
    Repeat
      Delay(1)
    Until ElapsedMilliseconds() > StartZeit + 15000

    StopModule(#MODUL_DATEI)

  EndIf
EndIf
End

DataSection
  ModulDatei:
    IncludeBinary "Eighth.mod"
  ModulDateiEnde:
EndDataSection
```

Dieses Beispiel ist dem Beispiel zum einbinden von Wave Dateien sehr ähnlich, allerdings ist ein kleiner Unterschied zu beachten. Das Modul wird in diesem Beispiel mit dem 'CatchModule()' Befehl (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Module → CatchModule) aus der DataSection geladen. Dieser benötigt (anders als der 'CatchSound()' Befehl) drei Parameter. Der erste Parameter ist die PB Nummer die Sie dem Modul zuweisen möchten. Der zweite Parameter ist die Startadresse des Moduls im Speicher. Diese ermitteln wir wieder über die Sprungmarke vor der 'IncludeBinary' Anweisung in der DataSection. Zum ermitteln der Adresse ist auch hier wieder dem Sprungmarkennamen ein Fragezeichen voranzustellen. Der Dritte Parameter ist die Größe der Moduldatei in Byte. Diese können Sie mit einem kleinen Trick ermitteln. Wie Sie sehen können habe ich in der DataSection unterhalb der 'IncludeBinary' Anweisung eine weitere Sprungmarke definiert ('ModulDateiEnde:'). Durch diese Sprungmarke kann ich das Ende der Modul Datei im Speicher ermitteln. Wenn ich nun von der Endadresse die Startadresse subtrahiere erhalte ich genau die Länge der zwischen den beiden Sprungmarken abgelegten Datei in Byte und habe

somit meinen dritten Parameter ermittelt. Das Abspielen der Modul Datei erfolgt dann wieder mit dem 'PlayModule()' Befehl.

Um mehr über die Befehle zum manipulieren von Modul Dateien zu erfahren schauen Sie in der PureBasic Hilfe im Bereich der Module Bibliothek nach (Hilfe Datei: Referenz-Handbuch → 2D Spiele & Multimedia Libraries → Module).

MP3's

MP3 Dateien wurden schnell zum beliebtesten Sound Dateiformat. Das liegt daran, dass MP3 Dateien mittlerweile der Quasi Standard für nahezu alle aus dem Internet ladbaren Musikdateien sind. MP3 ist eine Abkürzung für 'MPEG-1 Audio Layer 3', es ist also verständlich warum man es abgekürzt hat.

MP3 Dateien werden in PureBasic etwas anders gehandhabt. Zum abspielen müssen wir die Befehle aus der 'Movie' Bibliothek verwenden (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Movie). Es erscheint vielleicht etwas verwirrend, dass wir die Movie Befehle zum abspielen von MP3 Dateien verwenden müssen, diese Befehle können allerdings sehr viel mehr abspielen als einfach nur Filme. Die Movie Befehle bieten eine Möglichkeit nahezu alle Medien Dateien zu laden und abspielen, für die ein entsprechender Codec auf dem Rechner installiert ist. Neben Video Formaten können Sie mit diesen Befehlen auch Audio Formate abspielen.

Hier ist eine Auflistung der verbreiteteren Formate die die 'Movie' Bibliothek abspielen kann, vorausgesetzt Sie haben zuvor den entsprechenden Codec installiert. Es ist wahrscheinlich auch möglich mehr als die hier angegebenen Formate abzuspielen:

Video Dateien:

Audio Video Interleave	('*.avi')
MPEG Video	('*.mpg')

Audio Dateien:

MIDI Dateien	('*.mid')
MP3 Dateien	('*.mp3')
Ogg Vorbis	('*.ogg')
Wave Dateien	('*.wav')

Die 'Movie' Bibliothek scheint eine 'Eierlegende Wollmilchsau' zum abspielen von Medien Dateien zu sein und manche Anwender haben darum gebeten sie in 'Medien' Bibliothek umzubenennen, aber Sie müssen sich immer vor Augen halten dass eine Medien Datei, die auf Ihrem Computer gut abgespielt wird, auf einem anderen Computer unter Umständen nicht abspielbar ist. Das hängt damit zusammen welche Codecs auf den Rechnern installiert (oder nicht installiert) sind. Ich behaupte allerdings das obige Liste auf jedem Standard Rechner abgespielt werden kann, nageln Sie mich aber nicht darauf fest.

Im nächsten Beispiel habe ich die Movie Befehle verwendet um einen einfachen MP3 Player zu erstellen, der einen Lautstärke Regler und einen Balance Regler enthält. Er reicht zwar nicht an 'WinAmp' heran, aber er zeigt wie einfach es ist mit PureBasic einen Media Player zu erstellen.

```
Enumeration
  #FENSTER_HAUPT
  #SOUND_DATEI
  #TEXT_DATEI
  #KNOPF_WAEHLE_DATEI
  #TEXT_LAUTSTAERKE
  #TRACKBAR_LAUTSTAERKE
  #TEXT_BALANCE
  #TRACKBAR_BALANCE
  #KNOPF_SPIELE_DATEI
  #KNOPF_PAUSIERE_DATEI
  #KNOPF_STOPPE_DATEI
EndEnumeration

Global DateiName.s     = ""
Global DateiPausiert.i = #False

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER_HAUPT, 0, 0, 500, 215, "MP3 Player", #FLAGS)
  TextGadget(#TEXT_DATEI, 10, 10, 480, 20, "", #PB_Text_Border)
```

```
ButtonGadget(#KNOPF_WAEHLE_DATEI, 10, 40, 150, 20, "Wähle MP3 Datei...")
TextGadget(#TEXT_LAUTSTAERKE, 10, 70, 480, 20, "Lautstärke")
TrackBarGadget(#TRACKBAR_LAUTSTAERKE, 10, 90, 480, 20, 0, 100)
SetGadgetState(#TRACKBAR_LAUTSTAERKE, 100)
TextGadget(#TEXT_BALANCE, 10, 120, 480, 20, "Balance")
TrackBarGadget(#TRACKBAR_BALANCE, 10, 140, 480, 20, 0, 200)
SetGadgetState(#TRACKBAR_BALANCE, 100)
ButtonGadget(#KNOPF_SPIELE_DATEI, 10, 180, 100, 20, "Spielen")
ButtonGadget(#KNOPF_PAUSIERE_DATEI, 130, 180, 100, 20, "Pause")
ButtonGadget(#KNOPF_STOPPE_DATEI, 250, 180, 100, 20, "Stop")

If InitMovie()
  Repeat
    Ereignis.i = WaitWindowEvent()
    Select Ereignis
      Case #PB_Event_Gadget

        Select EventGadget()
          Case #KNOPF_WAEHLE_DATEI
            DateiName = OpenFileRequester("Choose","","MP3 File (*.mp3)|*.mp3",0)
            If DateiName <> ""
              SetGadgetText(#TEXT_DATEI, GetFilePart(DateiName))
              LoadMovie(#SOUND_DATEI, DateiName)
            EndIf

          Case #TRACKBAR_LAUTSTAERKE, #TRACKBAR_BALANCE
            If DateiName <> ""
              Lautstaerke.i = GetGadgetState(#TRACKBAR_LAUTSTAERKE)
              Balance.i     = GetGadgetState(#TRACKBAR_BALANCE) - 100
              MovieAudio(#SOUND_DATEI, Lautstaerke, Balance)
            EndIf

          Case #KNOPF_SPIELE_DATEI
            If DateiName <> ""
              PlayMovie(#SOUND_DATEI, #Null)
              DateiPausiert = #False
              SetGadgetText(#KNOPF_PAUSIERE_DATEI, "Pause")
            EndIf

          Case #KNOPF_PAUSIERE_DATEI
            If DateiName <> ""
              If DateiPausiert = #False
                PauseMovie(#SOUND_DATEI)
                DateiPausiert = #True
                SetGadgetText(#KNOPF_PAUSIERE_DATEI, "Fortsetzen")
              Else
                ResumeMovie(#SOUND_DATEI)
                DateiPausiert = #False
                SetGadgetText(#KNOPF_PAUSIERE_DATEI, "Pause")
              EndIf
            EndIf

          Case #KNOPF_STOPPE_DATEI
            If DateiName <> ""
              StopMovie(#SOUND_DATEI)
              DateiPausiert = #False
              SetGadgetText(#KNOPF_PAUSIERE_DATEI, "Pause")
            EndIf

        EndSelect

    EndSelect
  Until Ereignis = #PB_Event_CloseWindow
  EndIf
EndIf
End
```

Wenn Sie auf diese kleine Beispiel schauen, dann sehen Sie das Sie zum verwenden der Movie Befehle auch zuerst wieder die Umgebung initialisieren müssen, genau wie bei den Wave Dateien und den Modulen. Um die Movie Befehle zu initialisieren verwenden Sie den 'InitMovie()' Befehl. Nachdem Sie diesen

aufgerufen haben können Sie alle anderen Befehle aus der 'Movie' Bibliothek verwenden. Wie die anderen Initialisierungsbefehle muss er auf erfolgreiche Initialisierung überprüft werden. Wenn die Initialisierung fehlgeschlagen ist können Sie die Movie Befehle nicht verwenden.

Um ein Video (oder in diesem Fall eine MP3 Datei) zu laden verwenden wir den 'LoadMovie()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Movie → LoadMovie). Dieser Befehl benötigt zwei Parameter. Der erste ist die PB Nummer die Sie mit dem Medium verknüpfen wollen und der zweite ist ein String, der den Namen der zu ladenden Datei enthält.

Nachdem die Medien Datei geladen wurde können wir sie mit dem 'PlayMovie()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Movie → PlayMovie) abspielen. Dieser Befehl benötigt zwei Parameter um Unterstützung für Video- und Audio Medien bereitzustellen. Der erste Parameter ist die PB Nummer des Mediums das Sie abspielen möchten, während der zweite Parameter der Betriebssystem Identifikator von einem Fenster ist. Dieser Betriebssystem Identifikator wird benötigt wenn Sie ein Video abspielen möchten, da das Video beim abspielen auf dieses Fenster gezeichnet wird. Wenn Sie reine Audio Dateien (wie zum Beispiel MP3's) abspielen möchten, dann können Sie die eingebaute Konstante '#Null' als Betriebssystem Identifikator verwenden. Dadurch wird die Wiedergabe nicht an ein vorhandenes Fenster gebunden:

```
...
PlayMovie(#SOUND_DATEI, #Null)
...
```

Ich habe in dem Beispiel auch die Befehle 'PauseMovie()' und 'ResumeMovie()' verwendet, die sehr einfach zu verwenden sind. Beide benötigen lediglich die PB Nummer des Mediums das Sie Anhalten oder Fortsetzen möchten.

Um Ihnen die Möglichkeit zum Abbrechen der Wiedergabe der Datei zu geben habe ich in diesem Beispiel den 'StopMovie()' Befehl verwendet. Dieser ist ebenfalls sehr einfach zu verwenden, da er auch nur die PB Nummer des Mediums als Parameter benötigt um die Wiedergabe abzubrechen.

Auch wenn das nur ein Gerüst eines Media Players ist und er nur MP3's unterstützt, ist es ein Leichtes für Sie den Code so weit auszubauen, dass er alle oben genannten Formate unterstützt. Warum probieren Sie es nicht einfach aus?

CD Audio

Das Abspielen von Audio CD's ist eine gute Möglichkeit qualitativ hochwertige Musik für ein Spiel oder eine Anwendung bereitzustellen. Im Internet gibt es viele freie Werkzeuge die es Ihnen ermöglichen Audio CD's zusammenzustellen und diese zu brennen. Die Verwendung von CD's zum anbieten von Musik hat den Vorteil, das beim abspielen nur sehr wenig Systemleistung beansprucht wird und die Qualität sehr hoch ist. Hier ist ein Beispiel das die PureBasic 'AudioCD' Bibliothek verwendet um einen einfachen CD Spieler zu erstellen:

```
Enumeration
  #FENSTER_HAUPT
  #KNOPF_VORHERIGER
  #KNOPF_SPIELEN
  #KNOPF_STOP
  #KNOPF_NAECHSTER
  #KNOPF_AUSWERFEN
  #TEXT_STATUS
  #FORTSCHRITT_LIED
  #LISTE_TITEL
EndEnumeration

;Globale Variablen, usw.
Global AnzahlTitel.i
Global AktuellerTitel.i

;Konvertiere Sekunden in einen String der Minuten enthält
Procedure.s KonvertiereZuMin(Sekunden.i)
  ProcedureReturn Str(Sekunden / 60) + ":" + Str(Sekunden % 60)
EndProcedure

;Setze den aktuellen Titel
Procedure AktualisiereStatusText(Titel.i)
```

```
  If AnzahlTitel > 0
    TitelLaenge.i       = AudioCDTrackLength(Titel)
    TitelLaengeString.s = KonvertiereZuMin(TitelLaenge)
    TitelZeit.s         = " (" + TitelLaengeString + ")"
    SetGadgetText(#TEXT_STATUS, "Titel: " + Str(Titel) + TitelZeit)
    SetGadgetState(#FORTSCHRITT_LIED, 0)
    If AudioCDStatus() > 0
      VergangeneZeit.i = AudioCDTrackSeconds()
      TitelZeit.s=" ("+KonvertiereZuMin(VergangeneZeit)+" / "+TitelLaengeString+")"
      SetGadgetText(#TEXT_STATUS, "Titel: " + Str(Titel) + TitelZeit)
      Fortschritt.f = (100 / TitelLaenge) * VergangeneZeit
      SetGadgetState(#FORTSCHRITT_LIED, Fortschritt)
    EndIf
    SetGadgetState(#LISTE_TITEL, Titel - 1)
  Else
    SetGadgetText(#TEXT_STATUS, "Bitte legen Sie eine Audio CD ein")
  EndIf
EndProcedure

;Gehe zum nächsten Titel
Procedure NaechsterTitel()
  If AktuellerTitel < AnzahlTitel
    AktuellerTitel + 1
    AktualisiereStatusText(AktuellerTitel)
    If AudioCDStatus() > 0
      PlayAudioCD(AktuellerTitel, AnzahlTitel)
    EndIf
  EndIf
EndProcedure

;Gehe zum vorherigen Titel
Procedure VorherigerTitel()
  If AktuellerTitel > 1
    AktuellerTitel - 1
    AktualisiereStatusText(AktuellerTitel)
    If AudioCDStatus() > 0
      PlayAudioCD(AktuellerTitel, AnzahlTitel)
    EndIf
  EndIf
EndProcedure

;Fülle die Liste, die alle Titel auf der CD anzeigt
Procedure FuelleTitelListe()
  ClearGadgetItems(#LISTE_TITEL)
  AnzahlTitel = AudioCDTracks()
  If AnzahlTitel > 0
    For x.i = 1 To AnzahlTitel
      TitelLaenge.s = KonvertiereZuMin(AudioCDTrackLength(x))
      AddGadgetItem(#LISTE_TITEL, -1, "Titel " + Str(x) + " (" + TitelLaenge + ")")
    Next x
    If AktuellerTitel = 0
      AktuellerTitel = 1
    EndIf
  Else
    AktuellerTitel = 0
  EndIf
EndProcedure

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER_HAUPT, 0, 0, 320, 250, "CD Player", #FLAGS)
  ButtonGadget(#KNOPF_VORHERIGER, 10, 10, 60, 20, "Vorheriger")
  ButtonGadget(#KNOPF_SPIELEN, 70, 10, 60, 20, "Spielen")
  ButtonGadget(#KNOPF_STOP, 130, 10, 60, 20, "Stop")
  ButtonGadget(#KNOPF_NAECHSTER, 190, 10, 60, 20, "Nächster")
  ButtonGadget(#KNOPF_AUSWERFEN, 250, 10, 60, 20, "Auswerfen")
  TextGadget(#TEXT_STATUS, 10, 40, 300, 20, "", #PB_Text_Center)
  ProgressBarGadget(#FORTSCHRITT_LIED,10,65,300,10,0,100,#PB_ProgressBar_Smooth)
  ListViewGadget(#LISTE_TITEL, 10, 90, 300, 150)

  If InitAudioCD()
    FuelleTitelListe()
```

```
      StartZeit.i = ElapsedMilliseconds()

      Repeat
        Ereignis.i = WindowEvent()
        Select Ereignis
          Case #PB_Event_Gadget
            Select EventGadget()
              Case #KNOPF_VORHERIGER
                VorherigerTitel()
              Case #KNOPF_SPIELEN
                If AnzahlTitel > 0
                  PlayAudioCD(AktuellerTitel, AnzahlTitel)
                EndIf
              Case #KNOPF_STOP
                StopAudioCD()
              Case #KNOPF_NAECHSTER
                NaechsterTitel()
              Case #KNOPF_AUSWERFEN
                EjectAudioCD(#True)
                FuelleTitelListe()
              Case #LISTE_TITEL
                If EventType() = #PB_EventType_LeftDoubleClick
                  AktuellerTitel = GetGadgetState(#LISTE_TITEL) + 1
                  AktualisiereStatusText(AktuellerTitel)
                  PlayAudioCD(AktuellerTitel, AnzahlTitel)
                EndIf
            EndSelect
        EndSelect

        AktuelleZeit.i = ElapsedMilliseconds()
        If AktuelleZeit > StartZeit + 1000
          FuelleTitelListe()
          AktualisiereStatusText(AktuellerTitel)
          StartZeit.i = ElapsedMilliseconds()
        EndIf
        Delay(1)

      Until Ereignis = #PB_Event_CloseWindow
      StopAudioCD()
    EndIf
  EndIf
  End
```

Dieses Beispiel ist ein sehr einfacher CD Spieler der ein Minimum an Funktionalität zum Steuern und abspielen von Audio Cd's bereitstellt. Alle verwendeten Befehle aus der 'AudioCD' Bibliothek finden Sie in der PureBasic Hilfe (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → AudioCD).

Um die 'AudioCD' Befehle verwenden zu können müssen Sie zuerst die dafür nötige Umgebung initialisieren. Das erreichen wir mit dem 'InitAudioCD()' Befehl. Diesen Befehl müssen wir auf erfolgreiche Initialisierung überprüfen. Als Nebeneffekt gibt der Befehl die Anzahl der installierten CD Laufwerke zurück die in der Lage sind Musik abzuspielen. Wenn der Rückgabewert '0' ist, dann sind im Computer keine CD Laufwerke installiert oder es ist ein anderes Problem aufgetreten, dass das CD Laufwerk am abspielen von Musik hindert. Wenn der Rückgabewert größer als '0' ist, dann sollte alles in Ordnung sein.

Nachdem die Audio CD Umgebung initialisiert ist können Sie alle Befehle aus der 'AudioCD' Bibliothek verwenden.

Hier ist eine Auflistung der Befehle die ich in meinem CD Spieler Beispiel verwendet habe:

'PlayAudioCD()'
Dieser Befehl wird verwendet um den aktuellen Titel der eingelegten CD abzuspielen. Der Befehl benötigt zwei Parameter. Der erste ist der Titel mit dem der Abspielvorgang gestartet wird und der zweite ist der Titel nach dem die Wiedergabe abgebrochen wird. Wenn ich den Befehl zum Beispiel folgendermaßen Aufrufe:

```
PlayAudioCD(1, 3)
```

dann beginnt die Wiedergabe mit Titel '1' und geht dann zum nächsten Titel über. Die Wiedergabe wird abgebrochen wenn der dritte Titel fertig abgespielt ist.

'StopAudioCD()'
Dieser Befehl kann jederzeit verwendet werden um die Wiedergabe des aktuell abgespielten Titels zu stoppen.

'EjektAudioCD()'
Dieser Befehl öffnet oder schließt das CD Laufwerk, abhängig davon welcher Parameter übergeben wird. Dieser Befehl benötigt einen Parameter. Wenn Sie als Parameter den Wert '1' übergeben, dann wird die Schublade des CD Laufwerkes geöffnet, genauer gesagt wird die Wiedergabe abgebrochen und die CD ausgeworfen. Wenn Sie als Parameter den Wert '0' übergeben, dann wird die Laufwerksschublade geschlossen und die eingelegte CD geladen.

'AudioCDStatus()'
Dieser Befehl benötigt keinen Parameter und gibt uns einen Echtzeit Status der CD im Laufwerk zurück. Wenn dieser Befehl beim Aufruf '-1' zurückgibt, dann ist das Laufwerk nicht bereit, was bedeutet dass keine CD eingelegt ist oder die Laufwerksschublade nicht geschlossen ist. Wenn der Rückgabewert '0' ist, dann wurde die CD ordnungsgemäß erkannt, wird aber zur Zeit nicht abgespielt. Wenn der Rückgabewert größer als '0' ist, dann wird gerade ein Titel abgespielt. Beim Rückgabewert handelt es sich um die Nummer des aktuell abgespielten Titels.

'AudioCDTracks()'
Dieser Befehl benötigt keine Parameter und gibt beim Aufruf die Anzahl der abspielbaren Titel auf der geladenen CD zurück.

'AudioCDTrackLength()'
Dieser Befehl benötigt einen Parameter, nämlich die Titel Nummer einer geladenen CD. Mit dieser Information gibt der Befehl die Länge des Titels in Sekunden zurück.

'AudioCDTrackSeconds()'
Dieser Befehl benötigt keinen Parameter. Beim Aufruf gibt er die bereits vergangene Zeit des aktuell abgespielten Titels in Sekunden zurück.

Wenn Sie etwas genauer auf mein CD Spieler Beispiel schauen, werden Sie feststellen dass ich den Befehl 'WindowEvent()' an Stelle von 'WaitWindowEvent()' verwendet habe. Das bedeutet dass die Hauptschleife ständig durchlaufen wird, unabhängig davon ob ein Ereignis auftrat. Das ist in meinem Programm nötig, da ich eine Zeitabhängige Prozedur in meiner Hauptschleife aufrufe, die die Anzeige auf der grafischen Benutzeroberfläche des Programms aktualisiert. Wenn ich das nicht so gemacht hätte, würde die Statusanzeige nur bei einem Fensterereignis aktualisiert werden. Alternativ hätte ich auch den 'WaitWindowEvent()' Befehl mit einem optionalen 'Timeout' Parameter verwenden können, z.B:

```
...
Ereignis.i = WaitWindowEvent(10)
...
```

Die Befehle 'WindowEvent()' und 'WaitWindowEvent()' werden etwas detaillierter in Kapitel 9 beschrieben (Verstehen von Ereignissen).

IV. Fortgeschrittene Themen

In diesem Abschnitt beschreibe ich Dinge die etwas mehr fortgeschritten sind als die anderen Abschnitte in diesem Buch. Dieser Abschnitt ist ferner für diejenigen gedacht, die Fragen zu einem speziellen Aspekt von PureBasic haben, oder einfach mehr darüber erfahren möchten was "unter der Haube" passiert.

Sie benötigen nichts aus diesem Abschnitt um gute, stabile und funktionsreiche Programme zu erstellen, aber für alle die sich etwas tiefer mit der Materie beschäftigen möchten habe ich ihn hier angefügt.

13. Jenseits der Grundlagen

Diese Kapitel enthält eine Mischung aus verschiedenen Themen, da ich dachte dass alle diese Punkte erklärt werden sollten, aber nicht jeder Punkt den Inhalt für ein vollständiges Kapitel hergibt. Diese Themen sind nicht essentiell zum erstellen von PureBasic Programmen, aber sie sind wichtig für Programmierer die ihr Wissen über die fortgeschrittenen Funktionen von PureBasic erweitern wollen.

Diese Kapitel enthält folgende Themen: Compiler Direktiven und wie Sie den Compiler mit Code steuern; Fortgeschrittene Compiler Optionen, die mehr aus dem Compiler herausholen; Übergabe von Kommandozeilen Parametern an Ihre Programme, zum schreiben von Kommandozeilen Tools; wie speichert PureBasic numerische Datentypen; was sind Zeiger und wie werden sie verwendet; was sind Threads und wie werden sie verwendet; was sind 'Dynamic Linked Libraries' (DLLs) und wie werden sie erstellt; und wie nutzen Sie das 'Windows Application Programming Interface' (WIN32 API) direkt in PureBasic.

Compiler Direktiven und Funktionen

Compiler Direktiven sind Befehle die den PureBasic Compiler während des Kompilierens steuern. Diese Direktiven geben Ihnen die volle Kontrolle darüber welche Teile des Quelltextes kompiliert werden sollen und geben Ihnen die Möglichkeit Informationen vom Compiler abzurufen. Compiler Direktiven haben nur während des Kompilierens eine Auswirkung, nur zu dieser Zeit werden sie ausgewertet und bearbeitet. Zur Laufzeit des Programmes können sie nicht ausgewertet werden.

Compiler Funktionen unterscheiden sich von den Direktiven, da es sich hierbei um Befehle handelt die Ihnen die Möglichkeit geben Informationen über etwas bereits kompiliertes zu sammeln (üblicherweise ein Datentyp). Anders als Direktiven, können Compiler Funktionen zur Laufzeit des Programms ausgewertet werden.

Bereits zuvor in diesem Buch haben Sie schon Compiler Direktiven und Compiler Funktionen gesehen, aber Sie haben es nicht bemerkt. Ich habe Ihnen zum Beispiel in Kapitel 8 (Minimieren von Fehlern und deren Behandlung) den 'EnableExplicit' Befehl vorgestellt. Diese Compiler Direktive schaltet den Compiler in den Expliziten Modus und stellt sicher, dass alle Variablen explizit deklariert wurden, das heißt dass alle Geltungsbereiche und Typen von neuen Variablen definiert sind. Sie haben weiterhin den 'SizeOf()' und den 'OffsetOf()' Befehl in Kapitel 5 gesehen. Diese Compiler Funktionen sammeln Informationen über kompilierte Strukturen und Interfaces, wie z.B. die Größe einer Struktur im Speicher (Angabe in Byte) oder ermitteln den Offset eines bestimmten Feldes innerhalb einer Struktur oder eines Interface.

Es gibt noch einige Direktiven und Funktionen die man dieser Liste hinzufügen könnte, aber bevor ich weiter erkläre muss ich Ihnen die eingebauten, reservierten Compiler Konstanten erklären die diese fast alle benutzen.

Reservierte Compiler Konstanten

Diese eingebauten, reservierten Konstanten gleichen eigentlich all den anderen Konstanten, der Unterschied besteht darin, dass ihr Wert stark vom momentanen Compiler Status abhängt. Alle diese Konstanten können während des Kompilierens abgefragt werden um Entscheidungen zu treffen wie die Kompilierung ablaufen soll. Einige Konstanten stellen auch Informationen über den Kompilierungsprozess selbst zur Verfügung. Hier ist eine vollständige Liste der reservierten Compiler Konstanten und welche Informationen sie enthalten.

`#PB_Compiler_OS`

Der Wert dieser speziellen Konstante ist abhängig davon, auf welcher Plattform der Compiler läuft. Sie hat auch ein paar assoziierte Konstanten, die Ihnen helfen zu ermitteln, was der Wert bedeutet. Wenn die Konstante zum Beispiel eine Wert von entweder '#PB_OS_Windows', '#PB_OS_Linux', '#PB_OS_AmigaOS' oder '#PB_OS_MacOS' hat, dann läuft der Compiler auf dieser bestimmten Plattform.

#PB_Compiler_Processor
Mit dieser speziellen Konstante können Sie bestimmen, für welchen Prozessortyp das Programm kompiliert wird. Auch diese hat ein paar assoziierte Konstanten, die bei der Auswahl des Prozessortyps helfen. Sie können Programme für die Prozessorarchitekturen '#PB_Processor_x86', '#PB_Processor_x64', '#PB_Processor_PowerPC' oder '#PB_Processor_mc68000' erstellen.

#PB_Compiler_Date
Der Wert dieser Konstante enthält ein Datum in numerischer Form. Dieses Datum ist der Zeitpunkt zu dem das Programm kompiliert wurde. Dieser Wert ist im PureBasic Datumsformat hinterlegt, damit er einfach mit den Befehlen aus der 'Date' Bibliothek weiterverarbeitet werden kann. (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Date)

#PB_Compiler_File
Der Wert dieser Konstante ist ein String und enthält den vollen Pfad und Namen der Datei (*.pb) die gerade kompiliert wird.

#PB_Compiler_FilePath
Der Wert dieser Konstante ist ein String und enthält den vollen Pfad der Datei (*.pb) die gerade kompiliert wird.

#PB_Compiler_Line
Der Wert dieser Konstante ist die Zeilennummer der aktuellen Datei die kompiliert wird.

#PB_Compiler_Procedure
Der Wert dieser Konstante ist ein String und enthält den Prozedurnamen (wenn sich die Zeile innerhalb einer Prozedur befindet) der Datei (*.pb) die gerade kompiliert wird.

#PB_Compiler_Version
Der Wert dieser Konstante ist ein ganzzahliger Wert, der die Compiler Version angibt, mit der das Programm kompiliert wurde (z.B.: '451' für Compiler Version '4.51')

#PB_Compiler_Home
Der Wert dieser Konstante ist ein String und enthält den vollständigen Pfad des PureBasic Installationsordners auf Ihrem Computer.

#PB_Compiler_EnumerationValue
Der Wert dieser Konstante speichert den nächsten Wert einer Aufzählung. Damit ist es möglich mehrere Aufzählungen aus z.B. mehreren Quelltexten miteinander zu verketten.

#PB_Compiler_Debugger
Diese Konstante hat einen von zwei Werten. Wenn dieser Wert gleich '#True' ist, dann ist der Debugger eingeschaltet. Wenn dieser Wert gleich '#False' ist, dann war der Debugger vor der Kompilierung abgeschaltet.

#PB_Compiler_Thread
Diese Konstante kann einen von zwei Werten haben. Wenn dieser Wert gleich '#True' ist, dann war der Threadsichere Modus vor der Kompilierung eingeschaltet. Wenn dieser Wert gleich '#False' ist, dann wurde der Threadsichere Modus nicht verwendet.

#PB_Compiler_Unicode
Diese Konstante kann einen von zwei Werten haben. Wenn dieser Wert gleich '#True' ist, dann war der Unicode Modus vor der Kompilierung eingeschaltet. Wenn dieser Wert gleich '#False' ist, dann wurde der Unicode Modus nicht verwendet.

Alle diese reservierten Compiler Konstanten können in normalen Anweisungen und Ausdrücken verwendet werden, tatsächlich können Sie sie überall verwenden, wo Sie auch normale Konstanten verwenden. Allerdings sind die meisten Compiler Konstanten nur dann richtig nützlich, wenn sie mit folgenden Direktiven verwendet werden.

Die 'CompilerIf' Direktive

Wenn Sie verstehen wie Sie eine 'If' Anweisung in PureBasic verwenden müssen, dann werden Sie auch verstehen wie Sie 'CompilerIf' verwenden müssen. Eine 'CompilerIf' Anweisung verhält sich genauso wie jede andere 'If' Anweisung, aber sie wird niemals kompiliert. Es ist einfach die Compiler Version einer 'If' Anweisung.

Anders als normale 'If' Anweisungen entscheidet 'CompilerIf' welcher Teil des Codes kompiliert wird. Diese Entscheidung wird aufgrund des Ergebnisses aus einem Konstanten Ausdruck getroffen, Variablen sind nicht erlaubt. Im Gegensatz dazu entscheidet eine normale 'If' Anweisung welcher Teil des (bereits kompilierten) Codes ausgeführt wird, abhängig von einem Variablen Ausdruck.

Lassen Sie mich Ihnen ein Beispiel zur Verwendung von 'CompilerIf' in einer gängigen Form demonstrieren:

```
CompilerIf #PB_Compiler_OS = #PB_OS_Windows
  MessageRequester("Info", "Dieser Compiler läuft unter Microsoft Windows")
CompilerElse
  MessageRequester("Info", "Dieser Compiler läuft NICHT unter Microsoft Windows")
CompilerEndIf
```

Hier prüft das 'CompilerIf' diesen Konstanten Ausdruck: '#PB_Compiler_OS = #PB_OS_Windows'. Wenn dieser Ausdruck True zurückgibt, dann wissen wir dass dieser Code auf einem Computer mit Microsoft Windows kompiliert wurde. Wenn er False zurückgibt ist das nicht der Fall, so einfach ist das. Das wirkt sehr geradlinig, aber das eigentlich geniale bei der Benutzung von 'CompilerIf' ist die Tatsache, dass nur der Codeabschnitt kompiliert wird der die Bedingung erfüllt. Wenn wir das im Hinterkopf behalten, und unser Beispiel unter Microsoft Windows kompiliert haben, dann wird die Zeile:

```
MessageRequester("Info", "Dieser Compiler läuft NICHT unter Microsoft Windows")
```

niemals kompiliert und deshalb auch nie in die ausführbare Datei gepackt.

Eine 'CompilerIf' Anweisung besteht aus drei Befehlen, die alle in obigem Beispiel verwendet wurden. Die 'CompilerElse' Komponente die mit dem normalen 'Else' vergleichbar ist, ist optional, aber die gesamte Anweisung muss mit 'CompilerEndIf' abgeschlossen werden.

Manche Programmierer verwenden 'CompilerIf' um die Funktion Ihrer Programme zu beschränken, besonders wenn sie dieses als Testversion kompilieren. Wenn ich zum Beispiel ein nützliches Programm schreibe und möchte den Benutzern eine Demo geben, damit sie das Programm vor dem Kauf testen können, dann muss ich mit der Angst leben, das einige versuchen werden das Programm zu Cracken um die Einschränkungen zu entfernen. Ich kann mein Programm aber auch folgendermaßen erstellen:

```
#DEMO = #True

CompilerIf #DEMO

  ;Demo Code
  MessageRequester("Info", "Das ist eine Demo, Sie müssen die Vollversion kaufen.")

CompilerElse

  ;Vollversion Code
  Procedure.d MeinPI()
    ProcedureReturn ACos( - 1)
  EndProcedure

  Test.s = "Das ist die Vollversion." + #LF$ + #LF$
  Test.s + "Der Wert von Pi ist: " + StrD(MeinPI(), 16)
  MessageRequester("Info", Test)

CompilerEndIf
```

Das einfache Ändern des Wertes der Konstante '#DEMO' veranlasst den Compiler völlig verschiedene Versionen des Codes zu kompilieren. Das ist sehr nützlich, da es das Cracken Ihrer Programme verhindern kann, weil die Demoversion nicht den Code der Vollversion enthält. Versuchen Sie den Wert der Konstante '#DEMO' auf False zu setzen und kompilieren Sie die Vollversion des Programms.

Die 'CompilerSelect' Direktive

Die 'CompilerSelect' Anweisung ist eine Compiler Version der populären 'Select' Anweisung. Das bedeutet, dass mehrere Abschnitte des Quelltextes in Abhängigkeit von verschiedenen Werten einer Konstanten, kompiliert werden können. Ähnlich wie 'CompilerIf', kann 'CompilerSelect' nur Konstante Werte prüfen, da diese Prüfung auch wieder zur Kompilierzeit und nicht zur Laufzeit des Programms durchgeführt wird. Hier eine weitere Betriebssystemprüfung, diesmal mit 'CompilerSelect':

```
CompilerSelect #PB_Compiler_OS

  CompilerCase #PB_OS_Windows
    ;Windows spezifischer Code
    MessageRequester("Info", "Die Kompilierung erfolgte unter Microsoft Windows.")

  CompilerCase #PB_OS_Linux
    ;Linux spezifischer Code
    MessageRequester("Info", "Die Kompilierung erfolgte unter Linux.")

  CompilerCase #PB_OS_MacOS
    ;MacOS spezifischer Code
    MessageRequester("Info", "Die Kompilierung erfolgte unter MacOS X")

  CompilerCase #PB_OS_AmigaOS
    ;AmigaOS spezifischer Code
    MessageRequester("Info", "Die Kompilierung erfolgte unter Amiga OS.")

CompilerEndSelect
```

Mit einem solchen Konstrukt haben Sie die Möglichkeit ihr Programm individuell an die Bedürfnisse des jeweiligen Betriebssystems anzupassen auf dem sie es kompilieren. Sie können außerdem Befehle aus dem API des jeweiligen Betriebssystems verwenden. Da diese Befehle nicht Plattformübergreifend gültig sind, haben Sie mit einem 'CompilerSelect' die Möglichkeit die individuellen API Befehle des jeweiligen Systems zu separieren und können das gleiche Ergebnis für alle Plattformen erzeugen.

PureBasic stellt diverse Schlüsselwörter bereit damit Sie mit einem 'CompilerSelect' die gleiche Funktionalität wie mit einer normalen 'Select' Anweisung erhalten. Diese Schlüsselwörter sind 'CompilerSelect', 'CompilerCase', 'CompilerDefault' und 'CompilerEndSelect'. Sie haben bereits drei dieser Befehle in obigem 'CompilerSelect' Beispiel gesehen. Das vierte, 'CompilerDefault' wird als Ausweich-Schlüsselwort verwendet, um eine Ausweichmöglichkeit zur Verfügung zu stellen, falls kein 'CompilerCase' ein True zurückgibt, wie hier:

```
CompilerSelect #PB_Compiler_OS

  CompilerCase #PB_OS_AmigaOS
    ;AmigaOS spezifischer Code
    MessageRequester("Fehler", "Dieser Quelltext unterstützt kein Amiga OS.")

  CompilerDefault
    ;Dieser Code wird auf allen anderen Betriebssystemen kompiliert.
    MessageRequester("Info", "Dieser Code wird auf diesem System kompiliert.")

CompilerEndSelect
```

In dieses Beispiel habe ich eine Fehlermeldung eingebaut, falls jemand versucht dieses Beispiel unter AmigaOS zu kompilieren, auf allen anderen Systemen (geregelt durch das 'CompilerDefault' Schlüsselwort) funktioniert der Code einwandfrei.

Obwohl fast alle meine Beispiele für 'CompilerIf' und 'CompilerSelect' nur die '#PB_Compiler_OS' Konstante verwendet haben, sollten Sie sich merken dass jede beliebige Konstante, Compiler reserviert oder nicht, mit diesen Befehlen geprüft werden kann. Damit können Sie den Kompilierungsprozess mit PureBasic Code in so vielfältiger Weise steuern, wie Sie auf den ersten Blick überhaupt nicht erkennen können.

Die 'CompilerError' Direktive

Diese Direktive kann für sich alleinstehend oder in Direktiven wie 'CompilerIf' oder 'CompilerSelect' verwendet werden, um einen Compiler Fehler (mit hilfreichem Text) zu melden. Das ist praktisch wenn Sie den Kompilierungsprozess stoppen wollen und dem Anwender eine aussagekräftige Compiler Fehlermeldung präsentieren möchten. Hier ist das obige 'CompilerSelect' Beispiel in modifizierter Form, so das es eine reelle Compiler Fehlermeldung erzeugt:

```
CompilerSelect #PB_Compiler_OS

  CompilerCase #PB_OS_MacOS
    CompilerError "Dieser Quelltext unterstützt kein MacOS."
```

```
    CompilerDefault
      MessageRequester("Info", "Dieser Code wird auf diesem System kompiliert.")

  CompilerEndSelect
```

Wenn Sie auf dieses Beispiel schauen werden Sie feststellen, dass die 'CompilerError' Direktive anders als normale Befehle keine Klammern verwendet um ihre Parameter einzuschließen. Wir verwenden die 'CompilerError' Direktive einfach mit einem folgenden String. Dieser String ist die Fehlermeldung, die im Fehlerfall angezeigt wird. Wenn der Compiler auf eine 'CompilerError' Direktive trifft, wird die Kompilierung gestoppt und die Fehlermeldung angezeigt.

Die 'Subsystem()' Compiler Funktion

Um diese Compiler Funktion zu erklären, muss ich generell erst einmal die Subsysteme erklären und wie sie mit PureBasic zusammenhängen. Subsysteme sind ein einfach zu erfassendes Konzept. Einfach ausgedrückt, wenn Sie mit der Funktionalität eines in PureBasic eingebauten Befehls nicht zufrieden sind können Sie ihn durch eine eigene Version überschreiben. Diese neuen Versionen werden in einem Subsystem zusammengefasst.

Wenn Sie einen Blick in das PureBasic Installationsverzeichnis werfen, werden Sie einen Ordner mit dem Namen 'PureLibraries' entdecken. Dieser Ordner enthält die Bibliotheken-Dateien, die ihrerseits die eingebauten Befehle von PureBasic enthalten. Diese Bibliotheken werden im Fall der Verwendung eines Subsystems ersetzt. Wenn ich ersetzt sage meine ich natürlich nicht, dass diese gelöscht oder überschrieben werden. Eine Subsystem Bibliothek bekommt nur eine höhere Priorität als die Standard Bibliothek.

Um ein Subsystem zu erstellen müssen Sie zuerst eine PureBasic Bibliothek neu schreiben und anschließend kompilieren (mehr Informationen darüber erhalten Sie im PureBasic Unterordner 'SDK'). Diese neue Bibliothek muss alle Befehle der Original Bibliothek enthalten, und alle Befehlsnamen müssen exakt mit den Originalen übereinstimmen. Dann müssen Sie sich vergewissern dass der Name der neu kompilierten Bibliothek mit dem Namen der zu ersetzenden Bibliothek übereinstimmt.

Wenn Sie das alles erledigt haben, sollten Sie eine nahezu identische Kopie der Original Bibliothek vorliegen haben, ausgenommen die Funktionalität der enthaltenen Befehle (in diese Befehle können Sie jede beliebige Funktionalität einbauen). Um diese Bibliothek als Subsystem zu installieren, damit Sie es während des Kompilierens nutzen können, müssen Sie einen neuen Ordner im 'SubSystems' Ordner erstellen, der sich innerhalb des PureBasic Installationsverzeichnisses befindet. In diesem neuen Ordner müssen Sie noch einen Unterordner mit Namen 'PureLibraries' erstellen, und in diesen kopieren Sie ihre Ersatzbibliotheken. Die Verzeichnisstruktur sieht etwa so aus:

```
..\PureBasic\SubSystems\IhrSubSystem\PureLibraries\IhreNeueBibliothek
```

Damit PureBasic Ihr neues Subsystem verwendet müssen Sie im Compiler ein Subsystem Flag setzen, dann benutzt er die Bibliotheken aus Ihrem Subsystem und nicht die aus dem Standard 'PureLibraries' Ordner. Um das zu erledigen müssen Sie das 'Compiler-Optionen' Fenster öffnen (Menü: Compiler → Compiler-Optionen...) und den Namen des Subsystem Ordners (denjenigen, den Sie im 'SubSystems' Ordner erstellt haben) in das 'Library Subsystem:' Feld eintragen und dann auf 'OK' klicken. Wenn Sie Ihr Programm das nächste mal kompilieren, sucht der Compiler im 'SubSystems' Ordner und im 'PureLibraries' Ordner nach Bibliotheken. Wenn er in diesen beiden Ordnern Bibliotheken mit dem gleichen Namen findet, wird der Compiler die aus dem spezifizierten Subsystem gegenüber den Originalen bevorzugen, deshalb der Begriff ersetzen.

Sie finden bereits ein fertiges Subsystem Beispiel im 'SubSystems' Ordner. Wenn Sie in diesen hineinschauen, finden Sie einen Ordner mit dem Namen 'OpenGL', in dem sich wiederum ein 'PureLibraries' Ordner befindet, und in diesem befinden sich die Ersatzbibliotheken für 'Palette', 'Screen', 'Sprite' und 'Sprite3D'. Mit diesen Bibliotheken werden die Originalen ersetzt, wenn dieses Subsystem ausgewählt wird. Wenn dieses Subsystem mittels Compiler Flag ausgewählt wurde, wird es für 2D Grafik 'OpenGL' anstelle von 'DirectX' verwenden.

Wie bereits zuvor erwähnt, werde ich Ihnen nun die 'Subsystem()' Compiler Funktion erklären.

Während der Kompilierung gibt die 'Subsystem()' Compiler Funktion darüber Auskunft, ob mittels Compiler Flag ein bestimmtes Subsystem aktiviert wurde. Die Syntax ist einfach genug zu verstehen. Wenn ich zum Beispiel das 'OpenGL' Subsystem aktiviert habe, wird folgende Debug Meldung angezeigt:

```
If Subsystem("OpenGL")
  Debug "Das OpenGL Subsystem wird verwendet."
EndIf
```

Die 'Subsystem()' Compiler Funktion hat einen Parameter, es ist der wörtliche String des Subsystems auf das Sie prüfen möchten (Variablen können an dieser Stelle nicht verwendet werden). Wenn das entsprechende Subsystem aktiviert ist und zum kompilieren des Programms verwendet wird, dann gibt die 'Subsystem()' Compiler Funktion True zurück.

Die 'Defined()' Compiler Funktion

Beim erstellen von Software müssen Sie manchmal prüfen, ob ein bestimmtes Datenobjekt definiert wurde. Dies ist durch die Verwendung der 'Defined()' Compiler Funktion möglich. Diese Funktion benötigt zwei Parameter, der Erste ist das Datenobjekt das Sie überprüfen möchten und der zweite ist der Datentyp des Objektes. Wenn dieses bestimmte Objekt definiert ist, dann gibt die 'Defined()' Compiler Funktion True zurück. Der Wert des geprüften Objekts interessiert nicht, es ist nur eine Information dass es definiert ist. Mit dieser Funktion können folgende Datenobjekte geprüft werden: Konstanten, Variablen, Arrays, Listen, Strukturen und Interfaces.

Wenn Sie die 'Defined()' Compiler Funktion verwenden, darf der erste Parameter nur den Namen des zu prüfenden Objekts enthalten - keine der üblichen Präfixe oder Suffixe die normalerweise mit dem Objekt verwendet werden. Die Namen von Konstanten dürfen zum Beispiel kein '#' Zeichen enthalten, und Array- und Listen Namen dürfen keine Klammern enthalten. Für den zweiten Parameter, den Typ des zu prüfenden Objektes, können Sie eingebaute Konstanten verwenden, hier eine vollständige Liste der verfügbaren Konstanten:

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

Meine Erklärung klingt so, als sei es kompliziert diese Funktion zu verwenden, deshalb schaffe ich diesen Eindruck besser aus der Welt und zeige Ihnen an einem Beispiel wie einfach es in Wahrheit ist. Dieses Beispiel prüft, ob die Variable 'Name.s' bereits definiert ist, wenn das nicht der Fall ist wird sie definiert:

```
Global Name.s = "Name 1"

If Not Defined(Name, #PB_Variable)
  ;Die Variable wird nur definiert, wenn sie zuvor nicht definiert wurde
  Global Name.s = "Name 2"
EndIf

Debug Name
```

Sie können diese Beispiel testen wenn Sie die erste Zeile auskommentieren. Wenn Sie das tun, findet die Variablendefinition nun innerhalb der 'If' Anweisung statt und die Stringvariable hat nun den Wert 'Name 2'. Code wie dieser ist in großen Projekten unbezahlbar, da Sie womöglich viele Quelltext Dateien in den Hauptquelltext einbinden und vielleicht etwas wichtiges darauf überprüfen möchten, ob es bereits definiert wurde. Wenn nicht, dann definieren Sie es.

Fortgeschrittene Compiler Optionen

Wenn Sie PureBasic verwenden, müssen Sie vielleicht für ein bestimmtes Programm die Compiler Einstellungen verändern oder Sie möchten das der Compiler bestimmte Informationen ausgibt. PureBasic bietet Ihnen die Möglichkeit dazu, indem es einige Compiler Optionen zur Verfügung stellt. Sie können die meisten dieser Optionen visuell ein- oder ausschalten. Benutzen Sie dazu innerhalb der IDE das 'Compiler-Optionen' Fenster (Menü: Compiler → Compiler-Optionen...). Um Zugriff auf alle unterstützten Compiler Optionen zu erhalten, müssen Sie den Compiler aus einem Kommandozeilen Interpreter heraus aufrufen und alle Optionen als Parameter übergeben.

Was ist also ein Kommandozeilen Interpreter? Nun, es ist ein kleines Programm, das üblicherweise mit Ihrem Betriebssystem ausgeliefert wird (wird manchmal auch als 'Shell' bezeichnet). Dieses Programm erlaubt es Ihnen auf den Laufwerken des Computers zu navigieren oder Programme zu starten und Computer Funktionen durch bekannte Befehle zu steuern. Es hat keine andere Benutzerschnittstelle als die Texteingabe

Es sind viele Kommandozeilen Interpreter verfügbar, und diese scheinen einige der populärsten zu sein:

'CLI' unter AmigaOS
'CMD' unter Microsoft Windows
'BASH' unter Linux
'Terminal' unter MacOS X

Damit diese Kommandozeilen Interpreter korrekt mit PureBasic arbeiten, müssen Sie den vollen Verzeichnispfad des Compilers in Ihre 'PATH' Umgebungsvariable aufnehmen. Das ist nötig, damit der Kommandozeilen Interpreter immer genau weiß wo der Compiler zu finden ist. Das verwenden der Umgebungsvariable erspart Ihnen das Eintippen des kompletten Verzeichnispfades zum Compiler, wenn Sie diesen aus der Kommandozeile heraus verwenden. Das verwenden eines Kommandozeilen Interpreters ist nach meiner Meinung nicht sehr benutzerfreundlich, aber es ist die einzige Möglichkeit einige der fortgeschritteneren Funktionen des PureBasic Compilers zu nutzen. Ich sage das so, obwohl ein Compiler, der über die Kommandozeile gesteuert werden kann auch Vorteile hat, besonders wenn Sie ihren eigenen Quelltext Editor verwenden. In diesem Fall können Sie den Compiler als externes Werkzeug konfigurieren.

Verwendung des Compilers auf der Kommandozeile

Die Verwendung des Compilers auf der Kommandozeile ist relativ einfach, Sie müssen nur darauf achten, dass Sie alle optionalen Parameter korrekt übergeben. Der eigentliche Compiler Befehl selbst beginnt immer mit dem Namen des PureBasic Compilers, gefolgt von allen benötigten Parametern. Der einzige Pflichtparameter ist eine Datei die dem Compiler zum verarbeiten übergeben wird. Um eine ausführbare Datei in ihrer einfachsten Form mit dem Kommandozeilen Compiler zu erstellen, benötigen Sie keine optionalen Parameter, aber Sie benötigen den Namen einer '*.pb' Datei. Öffnen Sie ein Kommandozeilen Interpreter Fenster, geben folgendes ein und drücken 'Enter':

```
PBCompiler MeinProgramm.pb
```

Wenn Ihre 'PATH' Umgebungsvariable korrekt konfiguriert ist, wird hiermit der PureBasic Compiler gestartet und die Datei 'MeinProgramm.pb' übergeben. Der Compiler erstellt ein temporäres Executable im Compiler Verzeichnis und startet es auf die gleiche Weise, als würden Sie in der IDE die 'Kompilieren/Starten' Schaltfläche drücken. Das ist die einfachste Möglichkeit ein Programm auf der Kommandozeile zu kompilieren.

Wenn Sie aus Ihrem Programm ein Executable mit benutzerdefiniertem Namen erstellen wollen, das sich im gleichen Verzeichnis wie der Quelltext befindet, können Sie folgenden Befehl verwenden:

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe"
```

Diese Anweisung kompiliert die Datei 'MeinProgramm.pb' und erstellt ein ausführbares Programm aus ihr. Dieses kompilierte Executable erhält den Namen 'MeinProgramm.exe' und wird im selben Verzeichnis erstellt in dem sich die '*.pb' Datei befindet. Beachten Sie, in dieser Anweisung verwenden wir eine neue Compiler Option: '/exe'. Diese teilt dem Compiler mit, dass wir ein Executable mit sinnvollem Namen erstellen wollen. Der String nach dieser Compiler Option gibt an welchen Namen die Datei erhalten soll. Alle Kommandozeilen Compiler Optionen beginnen mit einem vorangestellten Slash '/' wie dieses. Diese beiden Beispiele sind relativ einfach zu verstehen, tatsächlich werden sie in der Praxis aber selten verwendet. Die Programmierer bevorzugen es stattdessen die PureBasic IDE zu verwenden um solche einfachen Kompilierungen durchzuführen. Trotzdem ist es nützlich diese beiden zu kennen, besonders wenn Sie einen neuen Quelltext Editor entwerfen wollen.

Wenn Sie den Compiler auf der Kommandozeile verwenden ist es auch wichtig zu verstehen, dass Sie viele Optionen gleichzeitig verwenden können. Wenn ich zum Beispiel eine '*.pb' Datei zu einem Executable kompilieren möchte und diesem ein Icon zuweisen möchte, dann würde ich eine Befehlszeile wie diese verwenden:

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /icon ".\MeinProgramm.ico"
```

Der Compiler erkennt, dass der String hinter der '/exe' Option der Name der zu erstellenden ausführbaren Datei ist und der String hinter der '/icon' Option die Icon Datei ist, die wir auf das Executable anwenden wollen. Alle Kommandozeilen Compiler Optionen müssen immer durch Leerzeichen separiert werden.

PureBasic Kommandozeilen Compiler Optionen

Was nun folgt ist eine komplette Liste der PureBasic Kommandozeilen Compiler Optionen, einschließlich einer Erklärung jedes einzelnen und ein kleines Beispiel zur Verwendung. Merken Sie sich, die meisten dieser Optionen können in der IDE aktiviert werden indem Sie eine Checkbox oder ein Drop-Down Menü im 'Compiler-Optionen' Dialog verwenden. Dieser lässt Sie das Compiler Verhalten erheblich leichter kontrollieren. All diese Kommandozeilen Parameter sind nicht 'Case Sensitive', das heißt, dass Sie sie groß oder klein schreiben können, ganz nach Ihren Vorlieben.

/?

Diese Option ist nur in der Kommandozeile verfügbar und zeigt auf einer hilfreichen Seite alle in PureBasic verfügbaren Kommandozeilen Compiler Optionen an. Diese Hilfeseite wird in der Kommandozeilen Fenster angezeigt. Diese Compiler Option setzt alle anderen mit übergebenen Optionen außer Kraft.

```
PBCompiler /?
```

/Commented

Dies ist eine weitere Option, die nur auf der Kommandozeile verfügbar ist und erstellt eine kommentierte Assembler Datei mit dem Namen 'PureBasic.asm' im 'PureBasic\Compilers' Ordner, zusammen mit dem erstellten Executable. Diese Datei enthält die Ausgabe des PureBasic Compilers in roher Assembler Form, nachdem er die '*.pb' Datei kompiliert hat. Diese Datei kann erneut kompiliert werden, wenn Sie zum Beispiel Veränderungen vorgenommen haben. Verwenden Sie einfach die '/reasm' Compiler Option um aus der Assembler Datei ein Executable zu erstellen.

```
PBCompiler /commented
```

/Console

Diese Compiler Option ermöglicht Ihnen die Erstellung einer reinen Konsolen Anwendung. Hier ein einfaches Beispiel:

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /console
```

Um mehr Informationen über Konsolen Programme zu erhalten springen Sie zu Kapitel 9 (Konsolen Programme).

/Constant (Name=Wert)

Diese Option ist ebenfalls nur auf der Kommandozeile verfügbar und ermöglicht es Ihnen auf der Kommandozeile eine Konstante dynamisch zu definieren. Das kann nützlich sein, wenn Sie ein Programm erstellen das seine Funktionalität in Abhängigkeit vom Wert einer Konstanten ändert. Zum Beispiel: 'DEMO=1' oder 'DEMO=0'.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /constant MEIN_STRING="Hallo Welt"
```

Wie Sie in diesem Beispiel sehen können, dürfen Sie bei der Definition der Konstante kein '#' Zeichen als Präfix verwenden und neben dem Gleichheitszeichen ('=') dürfen sich keine Leerzeichen befinden. Wenn Sie eine String Konstante definieren, müssen Sie Diesen in Anführungszeichen einschließen, so wie ich es in meinem Beispiel getan habe. Das ist nicht hundertprozentig nötig, da der Compiler bei fehlenden Anführungszeichen automatisch versucht den Wert in einen String zu konvertieren, aber es können Fehler entstehen wenn der String Leerzeichen enthält. Um auf Nummer sicher zu gehen verwenden Sie Anführungszeichen für Strings.

/Debugger

Wenn Sie nur diese Compiler Option verwenden, dann wird der eigenständige (GUI) Debugger für das erzeugte temporäre Executable eingeschaltet. Wenn diese Option in Kombination mit der '/exe' Option verwendet wird, dann wird der Konsolen Debugger in das Executable mit eingebunden. Der Konsolen Debugger zeigt dann ein Konsolenfenster an während das Programm läuft. Dies erlaubt das Debuggen des Programms auf der Kommandozeile. Hier zwei Beispiele:

```
PBCompiler MeinProgramm.pb /debugger
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /debugger
```

/DLL
Diese Compiler Option ermöglicht Ihnen aus Ihrem PureBasic Quelltext eine DLL (Dynamic Linked Library) zu erstellen. Wenn Sie diese Option bei der Kompilierung verwenden, wird die erstellte DLL im 'PureBasic\Compilers' Verzeichnis unter dem Namen 'PureBasic.dll' abgelegt. Diese Datei können Sie dann beliebig umbenennen. Als ein Nebenprodukt bei der DLL Erstellung werden noch eine Statische Bibliothek mit Namen 'PureBasic.lib' und eine exportierte Datei mit Namen 'PureBasic.exp' im gleichen Ordner erstellt.

```
PBCompiler MeinProgramm.pb /dll
```

PureBasic Quelltext Dateien die zur Erstellung von DLL's verwendet werden bestehen üblicherweise nur aus Prozeduren. Manche dieser Prozeduren müssen speziell benannt werden um die Funktionalität der Bibliothek nach außen zu geben. Etwas weiter hinten in diesem Kapitel finden Sie mehr Informationen über DLL's.

/DynamicCpu
Diese Kommandozeilen Option ist äquivalent zur gleichzeitigen Verwendung der Optionen '/mmx', '/3dnow', '/sse' und '/sse2' zusammen. Wenn ihr Programm Quelltext Assembler Routinen enthält, die alle diese Prozessor Erweiterungen unterstützen und Sie verwenden die '/DynamicCpu' Option, dann wird das resultierende Executable den kompletten Prozessorspezifischen Code enthalten. Was passiert, wenn dieses Executable gestartet wird? Das Programm schaut auf den Prozessortyp und wählt aus den kompilierten Routinen die für diese Prozessorarchitektur am besten optimierte aus. Das heißt, das Programm verwendet für jede Prozessorarchitektur dynamisch anderen im Programm enthaltenen Code. Das Executable wird durch dieses Verfahren größer als sonst üblich, da es viele verschiedene Versionen der gleichen Routinen enthält, dafür kann es Ihr Programm aber erheblich beschleunigen, da für jeden Prozessor eine optimierte Fassung der Routinen verwendet wird.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /ddynamiccpu
```

/Exe "Dateiname"
Diese Compiler Option ist einfach und geradlinig. Der String nach dem '/exe' Teil ist der Name, den das Executable nach der Kompilierung erhält. Der String mit dem Dateinamen muss immer dem '/exe' Teil folgen und muss immer in Anführungszeichen eingeschlossen werden, anderenfalls können unerwartete Ergebnisse auftreten. Wenn Sie mit dieser Option ein Executable erstellen, wird dieses im gleichen Verzeichnis erstellt in dem sich auch der Quelltext zum Programm befindet.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe"
```

/Icon "IconName"
Diese Compiler Option lässt Sie ein Icon auf Ihrer Festplatte Spezifizieren, das Ihr Programm verwenden soll. Der String hinter dem '/Icon' Teil muss definiert und in Anführungszeichen eingeschlossen werden. Er gibt an, wo sich das ausgewählte Icon auf Ihrer Festplatte befindet. Der String kann auch einen relativen Pfad enthalten, wie hier im Beispiel zu sehen ist:

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /icon ".\icons\MeinIcon.ico"
```

/IgnoreResident "Dateiname"
Diese Option ist nur auf der Kommandozeile verfügbar und ermöglicht es Ihnen das Laden einer Resident Datei während der Kompilierung zu unterbinden. Das ist praktisch wenn Sie eine Resident Datei neu kompilieren möchten die sich bereits im 'PureBasic\Residents' Ordner befindet. Wenn die installierte Resident Datei beim kompilieren nicht ignoriert wird erhalten Sie Fehlermeldungen, dass bestimmte Teile aus Ihrer Datei bereits definiert sind.

```
PBCompiler MeinProg.pb /resident "MeinResid.res" /ignoreresident "MeinResid.res"
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /ignoreresident "MeinResid.res"
```

Die erste Zeile zeigt ein Beispiel, in dem eine bereits zuvor installierte Version einer Resident Datei ignoriert wird. Die Zweite Zeile zeigt wie Sie vorgehen müssen wenn Sie Strukturen oder Konstanten in Ihrem Programm neu definieren müssen, die bereits in einer installierten Resident Datei vorhanden sind.

/InlineAsm
Diese Option aktiviert die Inline Assembler Unterstützung in Ihrer '*.pb' Datei. Mit anderen Worten, wenn Sie irgendwelche Assembler Befehle in Ihrem Programm verwendet haben, sollten Sie diese Option einschalten damit die Assembler Befehle korrekt vom PureBasic Compiler verarbeitet werden.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /inlineasm
```

/LineNumbering
Diese Compiler Option fügt Ihrem Executable die Unterstützung für die interne Zeilennummerierung von Befehlen hinzu. Das ist sehr nützlich für Debugger Werkzeuge von Drittanbietern oder wenn Sie die 'OnError' Bibliothek verwenden.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /linenumbering
```

Beachten Sie, dass die Verwendung der Zeilennummerierung die Ausführungsgeschwindigkeit Ihres Executable negativ beeinflussen kann, da Sie dem Programm mehr 'Overhead' als üblich hinzufügen.

/Linker "Dateiname"
Diese Option ist nur auf der Kommandozeile verfügbar und ermöglicht es Ihnen eine Datei mit Linker Befehlen direkt an den Linker zu übergeben.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /linker "MeineLinkerBefehle.txt"
```

/MMX, /3DNow, /SSE und /SSE2
Diese vier Compiler Optionen erstellen Prozessor spezifische Executable, abhängig davon welche Option benutzt wird. Es kann immer nur eine der Optionen verwendet werden, da sie in das Executable Routinen einbinden die nur auf Prozessoren laufen die diese speziellen Erweiterungen unterstützen. Wenn Sie mehrere Assembler Routinen erstellt haben die alle Prozessorerweiterungen unterstützen, dann werden nur die zur Compiler Option passenden Routinen in das Executable eingebunden. Wenn Sie zum Beispiel ein Executable mit der '/mmx' Compiler Option erstellen, dann werden nur die MMX Routinen des Quelltextes im fertigen Programm verwendet und dieses wird nur auf Prozessoren laufen, die MMX unterstützen.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /mmx
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /3dnow
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /sse
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /sse2
```

/Quiet
Diese Option ist nur auf der Kommandozeile verfügbar und unterdrückt alle unnötigen Textausgaben des Kommandozeilen Compilers. Das ist praktisch wenn Sie zum Beispiel den Compiler als externes Werkzeug von einem anderen Quelltext Editor heraus verwenden wollen

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /quiet
```

/ReAsm
Diese Option ist nur auf der Kommandozeile verfügbar und ermöglicht es Ihnen eine zuvor mittels dem Parameter '/Commented' ausgegebene '*.asm' Datei zu reassemblieren und kompilieren.

```
PBCompiler PureBasic.asm /reasm
```

Wenn Sie diesen Befehl verwenden müssen Sie darauf achten das sich die '*.asm' im 'PureBasic\Compilers' Ordner befindet, da anderenfalls ein Compiler Fehler auftritt.

/Resident "DateiName"
Diese Option ist nur auf der Kommandozeile verfügbar und ermöglicht es Ihnen aus einer Standard '*.pb' Datei eine Resident Datei zu erstellen. Resident Dateien werden üblicherweise zur Vordefinition von Konstanten und Strukturen verwendet.

```
PBCompiler MeinProgramm.pb /resident "MeineResident.res"
```

Nach dem Parameter muss ein in Anführungszeichen eingeschlossener String angegeben werden. Dieser definiert den Dateinamen der neuen Resident Datei. Wenn dieser String fehlt oder nicht in Anführungszeichen eingeschlossen ist, tritt ein Compiler Fehler auf. Die Verwendung dieser Compiler Option erstellt eine Resident Datei im Verzeichnis des zugrundeliegenden Quelltextes.

/Resource "DateiName"
Diese Option ist nur unter Microsoft Windows verfügbar und bindet eine Resource Datei in die kompilierte DLL oder das Executable ein. Der '/resource' Option muss ein in Anführungszeichen eingeschlossener String folgen, der die einzubindende Resource Datei spezifiziert. Es kann nur eine Resource Datei eingebunden werden, diese kann bei Bedarf allerdings Referenzen auf andere Resource Dateien enthalten. An Stelle einer kompilierten Resource kann auch eine ASCII Datei mit Verweisen übergeben werden.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /resource "MeineResource.rc"
```

/Standby
Diese Option wird von der IDE verwendet um den Compiler in den Speicher zu laden, wo er auf Befehle wartet die ihm übergeben werden. Diese Option wird von normalen Anwendern üblicherweise nicht verwendet, da die Schnittstelle zum in den Speicher geladenen Compiler sehr dürftig dokumentiert ist.

/Subsystem "SubsystemName"
Diese Compiler Option ermöglicht es Ihnen ein Subsystem für die Kompilierung Ihres Programms zu spezifizieren. Eine Beschreibung zu den Subsystemen finden Sie ein wenig weiter vorn in diesem Kapitel, weshalb ich mich hier nicht selbst zitiere. Standardmäßig sind schon einige Subsysteme in PureBasic eingebaut, die Sie bei der Kompilierung auswählen können. Das sind folgende:

Windows:

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /subsystem "DirectX7"
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /subsystem "NT4"
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /subsystem "OpenGL"
```

Linux:

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /subsystem "GTK2"
```

MacOS X:

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /subsystem "GTK"
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /subsystem "GTK2"
```

Wie Sie sehen können, müssen Sie nach dem '/subsystem' Teil der Compiler Option einen in Anführungszeichen eingeschlossenen Subsystem Namen spezifizieren.

/Thread
Diese Compiler Option schaltet den 'Threadsafe' Modus für das kompilierte Executable an. Sie sollten diese Option für Programme aktivieren die Gebrauch von 'Threads' machen. Sie sollten beachten das Programme die den 'Threadsafe' Modus verwenden ein wenig langsamer laufen als Programme die diese Option nicht verwenden. Das liegt am höheren internen Verwaltungsaufwand der für die sichere Verwendung von Threads notwendig ist.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /thread
```

Mehr Informationen über Threads erhalten Sie später in diesem Kapitel.

/Unicode
Diese Compiler Option schaltet die Unicode Unterstützung für das kompilierte Executable ein. Unicode ist ein zeichencodierungs Schema das 16 Bit (2 Byte) pro Zeichen verwendet. Dadurch ist es möglich alle Zeichen der wichtigen Weltsprachen (lebende und tote) in einem einzelnen Zeichensatz abzulegen. Die Verwendung dieser Compiler Option sorgt dafür, dass alle verwendeten PureBasic Befehle sowie das interne String Management voll Unicode Kompatibel sind.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /unicode
```

/Version
Diese Compiler Option ist nur auf der Kommandozeile verfügbar. Die einzige Funktion besteht darin, die Version des Compilers im Kommandozeilen Fenster anzuzeigen. Diese Option überschreibt alle anderen Optionen in der selben Kommandozeilen Anweisung.

```
PBCompiler /version
```

/XP
Diese Compiler Option ist nur unter Microsoft Windows verfügbar. Sie schaltet die Windows XP Skin Unterstützung für Ihr Executable ein. Dadurch werden alle von Ihnen verwendeten Gadgets im Stil des Windows XP Theme angezeigt, wenn dieses beim Anwender aktiviert ist.

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe" /xp
```

Auswertung von Kommandozeilen Parametern

Vielleicht möchten Sie mit PureBasic einmal ein Kommandozeilen Werkzeug erstellen, dem Sie beim Aufruf verschiedene Parameter übergeben können. Der PureBasic Compiler arbeitet auf die gleiche Weise. Das Programm wird gestartet und beim Aufruf werden ihm Parameter übergeben. Die Werte dieser Parameter bestimmen welche Funktion das Programm ausführt. In diesem Abschnitt zeige ich Ihnen wie Sie die an Ihr Programm übergebenen Parameter auswerten und weiterverarbeiten.

Was sind Parameter?

Wenn Sie Parameter (manchmal auch Argumente genannt) an ein Kommandozeilen Programm übergeben, müssen Sie darauf achten wie Sie diese übergeben und wie sie eingesetzt werden. Wenn wir zum Beispiel dieses Compiler Beispiel aus dem vorhergehenden Kapitel betrachten:

```
PBCompiler MeinProgramm.pb /exe "MeinProgramm.exe"
```

Dieser Kommandozeilen Befehl weist das Betriebssystem an das Programm 'PBCompiler' zu starten und übergibt diesem drei Parameter ('MeinProgramm.pb', '/exe' und '"MeinProgramm.exe"'). Auch wenn wir nur eine Compiler Option nach dem '*.pb' Dateinamen (der erste Parameter) verwenden, enthält diese Option trotzdem zwei Parameter.

Wenn wir auf diese Art Parameter an ein Programm übergeben, dann trifft das Programm die Entscheidung über die Anzahl der Parameter anhand der auftretenden Leerzeichen im Parameter String. Wenn Sie Leerzeichen in einem einzelnen Parameter übergeben wollen, dann müssen Sie diesen Parameter in Anführungszeichen einschließen. Zum Beispiel würden all diese Kommandozeilen Befehle zwei Parameter an das Programm 'MeinProgramm' übergeben:

```
MeinProgramm /a 105
MeinProgramm /spiele /jetzt
MeinProgramm /spiele "Ich und mein Schatten"
MeinProgramm "Stelle es in den" "Hintergrund"
```

Wie Sie an diesen Beispielen erkennen können, dürfen Parameter aus Zahlen oder aus Strings bestehen. Wenn Sie an das Programm übergeben werden, werden sie allerdings automatisch in Strings konvertiert. Wenn Sie Parameter in Anführungszeichen übergeben (besonders bei Parametern die Leerzeichen enthalten), dann werden diese Anführungszeichen vor der Übergabe abgeschnitten. Es wird also nur der dazwischen befindliche String übergeben, nicht die Anführungszeichen.

Lesen von übergebenen Parametern in Ihrem Programm

Wenn Parameter von einem Kommandozeilen Aufruf an Ihr Programm übergeben werden, dann werden diese intern für Sie gespeichert, bereit zum Abruf wenn Sie sie benötigen. Durch Aufruf des Befehls 'ProgramParameter()' können Sie jeweils einen Parameter aus dieser internen Liste abrufen. Wenn Sie alle Parameter mit diesem Befehl abgerufen haben, gibt dieser einen leeren String zurück. Hier ein Beispielprogramm das alle an das Programm übergebenen Parameter auflistet:

```
AnzahlParameter.i = CountProgramParameters()
Text.s            = "Liste der übergebenen Parameter:" + #LF$ + #LF$

If AnzahlParameter > 0
  For x.i = 1 To AnzahlParameter
    Text.s + ProgramParameter() + #LF$
  Next x
Else
  Text.s + "Es wurde kein Parameter übergeben!"
EndIf

MessageRequester("Parameter Information", Text)
```

Die erste Zeile in diesem Beispiel verwendet den 'CountProgramParameters()' Befehl. Dieser gibt die exakte Anzahl der gültigen übergebenen Parameter zurück. Nach Aufruf dieses Befehls verwende ich eine 'For' Schleife in der ich den 'ProgramParameter()' Befehl bei jedem Schleifendurchlauf aufrufe. Somit erhalte ich alle gültigen Parameter. Wenn ich dieses Beispiel zu einem Executable mit Namen 'ListeParameter.exe' kompiliere, dann kann ich diesem Programm auf der Kommandozeile Parameter übergeben, wie hier:

```
ListeParameter /eins /zwei /drei
```

Diese Parameter habe ich nur als Beispiel angegeben, Sie können natürlich alle beliebigen Parameter angeben. Ich habe einen Slash ('/') als Präfix für meine Parameter verwendet, da dies schon fast eine Norm ist und von vielen Konsolen Programmen so erwartet wird. Wenn die Befehlszeile so eingegeben wurde, dann enthält die Variable 'AnzahlParameter' den Wert '3' und jedes Mal wenn das Programm den 'ProgramParameter()' Befehl aufruft wird jeder dieser Parameter in der Sequenz abgerufen. Beim ersten Aufruf von 'ProgramParameter()' wird '/eins' zurückgegeben, beim zweiten Aufruf wird '/zwei' zurückgegeben und der dritte Aufruf gibt '/drei' zurück. Wenn ich den 'ProgramParameter()' Befehl ein viertes mal aufrufen würde, wäre der Rückgabewert ein leerer String.

Testen von Kommandozeilen Parametern innerhalb der IDE

Wenn Sie eine Anwendung in der IDE entwickeln die Kommandozeilen Parameter akzeptiert, dann möchten Sie bestimmt nicht jedes Mal die Anwendung kompilieren und von der Kommandozeile aufrufen um diese zu testen. Die PureBasic IDE nimmt Ihnen diese lange Prozedur ab, indem Sie ein Eingabefeld zur Verfügung stellt in das Sie ihre Kommandozeilen Parameter eingeben können die zur Kompilierzeit mit übergeben werden sollen. Dieses Feld finden Sie im 'Compiler-Optionen' Dialog, den Sie über das Menü 'Compiler' erreichen. Wenn Sie in diesem Dialog den Reiter 'Kompilieren/Starten' anwählen finden Sie in der Mitte ein Eingabefeld mit der Beschriftung 'Executable-Kommandozeile:'. In dieses Feld geben Sie die Parameter ein die Sie beim Starten des Programms mit dem 'Kompilieren/Starten' Befehl übergeben möchten. Geben Sie hier ein paar Parameter ein, damit Sie die Beispiele in diesem Abschnitt testen können.

Treffen von Entscheidungen basierend auf übergebenen Parametern

Wenn Sie Kommandozeilen Parameter verwenden um die Funktionalität oder die Ausgabe zu steuern, dann müssen Sie in Ihrem Programm verschiedene Aktionen durchführen, abhängig von den Werten der übergebenen Parameter. Vielleicht soll ihr Programm bei der Übergabe eines bestimmten Parameters etwas völlig anderes tun. Das treffen von Entscheidungen aufgrund der übergebenen Parameter ist nicht so komplex wie Sie jetzt vielleicht vermuten. Sie verwenden einfach 'If' und 'Select' Anweisungen um die Parameter Werte zu prüfen.

Das nächste Beispiel demonstriert diesen Vorgang. Es prüft ob als Parameter die Werte 'Maus', 'Katze' oder 'Hund' übergeben wurden. Übergeben Sie einen oder mehrere dieser Parameter und betrachten Sie die Programmausgabe:

```
;Zähle Parameter
AnzahlParameter.i = CountProgramParameters()

;Add all command line parameters to a linked list
Global NewList Parameter.s()
If AnzahlParameter > 0
  For x.i = 1 To AnzahlParameter
    AddElement(Parameter())
    Parameter() = UCase(ProgramParameter())
  Next x
EndIf

;Prüfen ob ein bestimmter Parameter übergeben wurde
Procedure ParameterPassed(Parameter.s)
  ForEach Parameter()
    If Parameter = Parameter()
      ProcedureReturn #True
    EndIf
  Next
  ProcedureReturn #False
EndProcedure

;Prüfen ob der Parameter 'Maus' übergeben wurde
If ParameterPassed("MAUS")
  MessageRequester("Information", "'Maus' wurde als Parameter übergeben.")
EndIf
;Prüfen ob der Parameter 'Katze' übergeben wurde
If ParameterPassed("KATZE")
  MessageRequester("Information", "'Katze' wurde als Parameter übergeben.")
EndIf
```

```
;Prüfen ob der Parameter 'Hund' übergeben wurde
If ParameterPassed("HUND")
  MessageRequester("Information", "'Hund' wurde als Parameter übergeben.")
EndIf
```

Wenn Sie das Beispiel genauer betrachten können Sie sehen das ich den 'UCase()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → String → UCase) verwendet habe, um alle übergebenen Parameter in Großbuchstaben zu wandeln. Dadurch kann ich String Vergleiche durchführen ohne auf die Schreibweise zu achten. Wenn Sie das in Ihrem Programm so machen, dann können Sie einfach Großbuchstaben String Vergleiche durchführen, wie in den 'If' Anweisungen zu sehen ist. Das ist eine gute Vorgehensweise, da es keinen Unterschied macht wie der Anwender die Parameter eingibt (kleingeschrieben, großgeschrieben oder gemischt), alle Parameter werden vor dem Vergleich in der 'If' Anweisung in Großbuchstaben gewandelt.

Wenn Sie ihr Programm mit Unterstützung für Kommandozeilen Parameter ausstatten, dann scheint das recht altmodisch zu sein und wirkt auch ein wenig bieder. Sie geben allerdings dem Benutzer die Möglichkeit Ihr Programm von der Kommandozeile zu nutzen und erreichen auf diese Weise vielleicht mehr Anwender. Die Unterstützung von Parametern erhöht in jedem Fall die Flexibilität, das sollten Sie auf jeden Fall berücksichtigen wenn Sie ein Programm entwickeln.

Ein genauerer Blick auf die Numerischen Datentypen

Numerische Datentypen erlauben es Ihnen Zahlen in vielen verschiedenen Formaten zu speichern und jeder Typ benötigt eine unterschiedlich große Menge an Speicher um diese Zahlen zu speichern. Das hört sich nicht nach einer sensationellen Nachricht an, aber der Speicherverbrauch eines numerischen Typs entscheidet letztendlich über sein numerisches Limit. Wenn Sie in Kapitel 2 zurückblättern und sich Abb. 2 anschauen, sehen Sie eine Tabelle in der alle Grenzen der numerischen Datentypen von PureBasic angegeben sind. Diese Grenzen ergeben sich durch die begrenzte Anzahl der Bits die für den entsprechenden Datentyp im Speicher reserviert werden. Wenn Sie zum Beispiel auf den Datentyp Byte (.b) schauen, dann sehen Sie dass dieser Datentyp ein Byte im Speicher belegt, was acht binären Stellen oder 'Bits' entspricht. Diese Bits können nur eine endliche Zahl von Kombinationen und damit auch nur eine begrenzte Menge an Zahlen repräsentieren. Genau gesagt können acht Bits maximal zweihundertsechsundfünfzig (256) einmalige Kombinationen annehmen in der jedes Bit entweder '1' oder '0' sein kann. Das bedeutet dass maximal zweihundertsechsundfünfzig einmalige Zahlen von einer binären acht Bit Zahl (1 Byte) dargestellt werden können.

In diesem Abschnitt werde ich Ihnen erklären wie Zahlen im Speicher abgelegt werden und wie binäre Zahlen eigentlich berechnet werden. Ich hoffe das wird Ihnen ein wenig mehr Einblick darüber geben wie Computer im inneren funktionieren, was nicht nur für PureBasic interessant ist.

Wenn Sie in einer Programmiersprache mit Zahlen hantieren, müssen Sie einen ganz klaren Unterschied zwischen Ganzzahlen (Integer) und Fließkommazahlen (Float) machen. Integer sind Zahlen die positiv oder negativ sein können, sie enthalten aber keinen Dezimalpunkt. Float Zahlen können positiv oder negativ sein und enthalten immer einen Dezimalpunkt. Das wurde bereits früher in diesem Buch schon erklärt aber ich denke ich werde ihr Gedächtnis an dieser Stelle etwas auffrischen.

Was ist Binär?

Binäre Notation, oder Basis-Zwei wie sie auch manchmal genannt wird, ist ein numerisches System in dem die einzelnen Stellen nur die Werte '0' oder '1' annehmen können. Ähnlich dem Dezimal System (manchmal auch als Basis-Zehn bezeichnet) repräsentiert jede binäre Stelle eine andere Potenz der Basis.

Lassen Sie mich dieses Konzept ein wenig genauer erklären. Hier ein Beispiel für eine Dezimalzahl:

```
542
```

In diesem Beispiel ist die Ziffer '2' mit dem Potenzwert eins (10^0) verknüpft, die '4' mit dem Wert zehn (10^1) und die '5' mit dem Wert einhundert (10^2). Jede Potenz ist zehnmal größer als die rechts von ihr. Um zu ermitteln welchen Wert die Zahl repräsentiert müssen Sie einfach alle Ziffern mit ihren entsprechenden Potenzen multiplizieren und die Ergebnisse anschließend addieren, z.B. ('5' x hundert) + ('4' x zehn) + ('2' x eins) = '542' (fünfhundertzweiundvierzig).

Genau auf die gleiche Weise arbeiten Binärzahlen (Basis-Zwei) wenn Sie mit Integer Zahlen arbeiten. Die Position jeder binären Ziffer ist ebenfalls mit einer bestimmten Potenz der Basis verknüpft, mit dem Unterschied dass hier die Basis zwei ist. Das heißt, dass jede Ziffer den doppelten Wert im Vergleich zur

Ziffer an der rechten Seite aufweist. Auch wenn das sehr kompliziert klingt ist es doch überraschend einfach zu verstehen. Schauen Sie auf Abb. 44, dadurch wird Ihnen einiges klarer.

Vorzeichenloses Byte

Binärer Ziffern (Bits)	0	1	0	0	1	1	0	1	= 77
Zugeordnete Werte	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**	

8 Bit Zahl
(1 Byte)

Numerischer Bereich: '0' bis '255'

Abb. 44

Um den Wert zu ermitteln den diese binären Ziffern repräsentieren folgen wir der gleichen Prozedur wie im Dezimalsystem (Basis-Zehn). Wir multiplizieren den Wert jeder Ziffer mit der zugeordneten Potenz und addieren dann die Ergebnisse. Nehmen wir zum Beispiel den Wert '01001101' der in Abb. 44 zu sehen ist und den Wert '77' repräsentiert. Die Multiplikation und Addition der Ergebnisse habe ich bereits im Dezimal Beispiel demonstriert. Dieser Vorgang beschränkt sich hier auf die Addition da die die binäre Notation nur die Ziffern '0' und '1' verwendet. Wenn Sie auf Abb. 44 schauen und die Werte unterhalb der binären Einsen Addieren, dann erhalten Sie '64 + 8 + 4 + 1 = 77'.

Aus Abb. 44 können Sie jetzt entnehmen dass nur eine begrenzte Menge an Zahlen mit diesen acht Bit dargestellt werden kann. Bestimmte Typen haben bestimmte Begrenzungen, irgendwann reichen die Bits nicht mehr aus um größere Zahlen darzustellen. Größere Datentypen arbeiten auf die gleiche Weise, ihnen stehen nur mehr Bits zur Verfügung.

Vorzeichenbehaftete- und Vorzeichenlose Integer Zahlen

In Abb. 2 können Sie sehen, dass viele Integer Typen in PureBasic einen positiven und negativen Wertebereich haben. Man nennt diese Typen auch Vorzeichenbehaftete Numerische Typen, da ihr Wert potentiell ein negatives Vorzeichen haben kann. Vorzeichenlose Typen haben dagegen einen Wertebereich der von Null bis zu einer großen positiven Zahl reicht. Der PureBasic Typ 'Ascii' ist ein Beispiel für einen vorzeichenlosen Integer Typ, dieser ist ebenfalls in Abb. 2 zu sehen. Da durch diese Typen nur eine begrenzte Anzahl Bits reserviert wird benötigen wir zwei Methoden um die binären Werte zu interpretieren, abhängig davon ob sie vorzeichenbehaftet oder vorzeichenlos sind.

Wenn Sie auf Abb. 45 schauen sehen Sie, dass jeder Integer Datentyp auf zwei Arten gelesen werden kann. Das Byte zum Beispiel kann vorzeichenbehaftet und vorzeichenlos sein, und beide können Werte in einem Bereich von zweihundertsechsundfünfzig möglichen Zahlen enthalten, inklusive Null ... ja, auch Null ist eine Zahl!

Datentyp	Numerischer Wertebereich
vorzeichenbehaftetes Byte	-128 bis 127
vorzeichenloses Byte	0 bis 255
vorzeichenbehaftetes Word	-32768 bis 32767
vorzeichenloses Word	0 bis 65535
vorzeichenbehaftetes Long	-2147483648 bis 2147483647
vorzeichenloses Long *	0 bis 4294967295

* Dieser Datentyp ist nicht in PureBasic verfügbar (vielleicht in einer zukünftigen Version?)

Abb. 45

Lesen von Vorzeichenlosen Integer Zahlen unter Verwendung der binären Notation

Wie Sie in Abb. 44 gesehen haben ist eine vorzeichenlose Zahl einfach in binärer Notation darzustellen. Sie setzen zuerst die entsprechenden Bits auf '1' und addieren dann die zugeordneten Werte wo eine '1' auftritt. Zum Beispiel ist der Maximalwert eines vorzeichenlosen Bytes in binärer Darstellung '11111111' ('255'), also alle Bits auf '1' gesetzt. Schön und einfach.

Lesen von Vorzeichenbehafteten Integer Zahlen unter Verwendung der binären Notation

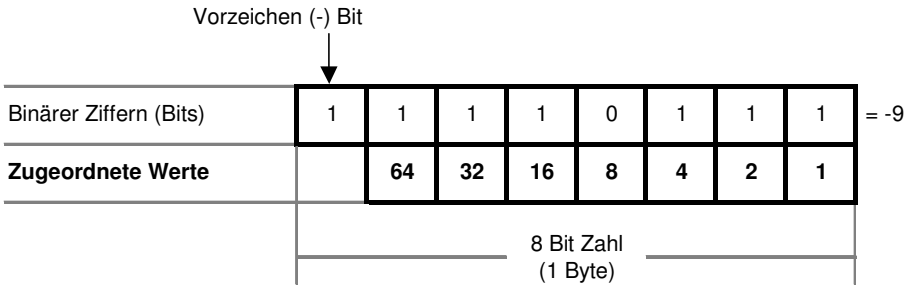
Ein vorzeichenbehafteter Integer Wert folgt ein wenig anderen Regeln bei der binären Darstellung. PureBasic verwendet ein System mit dem Namen Zweierkomplement um vorzeichenbehaftete Integer Werte darzustellen. In der Zweierkomplement Form gibt das linke und somit werthöchste Bit (manchmal auch 'Most Significant Bit' [MSB]) der vorzeichenbehafteten Binärzahl Auskunft darüber ob ein Vorzeichen (-) vorhanden ist oder nicht. Wenn das 'MSB' den Wert '0' hat handelt es sich um eine positive Integer Zahl, die auf die gleiche Art gelesen werden kann wie ich es im Absatz zuvor beschrieben habe. Wenn das 'MSB' den Wert '1' hat handelt es sich um eine negative Integer Zahl. Diese muss dann unter Verwendung der Zweierkomplement Form gelesen werden, die weiter unten erkläre.

Da das 'MSB' für das Vorzeichen reserviert ist stehen Ihnen nur noch die restlichen Bits zu Darstellung eines Zahlenwertes zur Verfügung. Lassen Sie uns wieder ein Byte als Beispiel nehmen. Wenn wir das Vorzeichen Bit auf '0' setzen um eine positive Zahl darzustellen, dann verbleiben uns nur noch sieben Bit für einen Zahlenwert. Diese sieben Bit setze ich nun alle auf '1', wie hier: '01111111'. Dieser Wert repräsentiert den höchsten positiven Wert den ein vorzeichenbehaftetes Byte enthalten kann, was in diesem Fall dem Wert '127' entspricht.

Wenn wir einen negativen Integer Wert in binärer Notation darstellen wollen müssen wir etwas anders vorgehen. Lassen Sie uns erst die vorzeichenlose Darstellung der Zahl '9' betrachten, was '00001001' entspricht. Da das 'MSB' den Wert '0' hat wissen wir definitiv dass es sich um eine positive Zahl handelt. Um diese Zahl in Zweierkomplement Form zu bringen und sie als negative Zahl darzustellen, müssen wir zuerst alle Bits Invertieren und dann zu dieser invertierten Zahl '1' addieren. Somit sieht die Zahl '-9' folgendermaßen aus: '11110111'. Mit Invertieren der Bits meine ich, dass Sie ein Bit mit dem Wert '0' in ein Bit mit dem Wert '1' umwandeln und ein Bit mit dem Wert '1' in ein Bit mit dem Wert '0'. Mit dieser Vorgehensweise bearbeiten Sie alle Bits des entsprechenden Typs. Wie Sie sehen hat das 'MSB' nun den Wert '1', was signalisiert dass es sich bei dieser Binärzahl um eine negative Zahl handelt. Das Invertieren der Bits und die Addition von '1' zu einer positiven Integer Zahl ist die Zweierkomplement Methode zum wandeln des Vorzeichens bei dieser Zahl.

Um diese Zahl wieder in eine positive Zahl zu wandeln folgen Sie exakt der gleichen Prozedur. Nach der Invertierung von '-9' ('11110111') sieht die Zahl folgendermaßen aus: '00001000'. Wir addieren dann Eins ('1') zu dieser Zahl und erhalten: '00001001', was der positiven '9' entspricht wenn wir die Zahl nach der normalen vorzeichenlosen Methode lesen. Abb. 46 zeigt ein vorzeichenbehaftetes Byte das den Wert '-9' visuell darstellt.

Vorzeichenbehaftetes Byte



Numerischer Bereich: '-128' bis '127'

Abb. 46

Auch wenn ich in den letzten paar Beispielen nur den Byte Typ verwendet habe um Ihnen zu erklären wie binäre Zahlen gelesen werden, gelten diese Methoden universell für alle numerischen Integer Typen, egal welcher Größe. In allen Integer Werten wachsen alle den Bits zugeordneten Werte um eine Potenz von '2', wenn Sie in der Binärzahl nach links wandern. In vorzeichenbehafteten Zahlen ist das ganz linke Bit immer das Vorzeichen Bit.

Fließkommazahlen und binäre Notation

Eine Fließkommazahl ist eine Zahl die einen Dezimalpunkt enthält und sowohl positiv oder auch negativ sein kann. Zahlen wie diese werden auf eine Art gespeichert in Der der Dezimalpunkt in der Zahl wandern kann, so dass es möglich ist sehr große oder sehr kleine Zahlen zu speichern. Durch diese Speichermethode ist es möglich riesige Zahlen zu speichern, allerdings geht das nur zu Lasten der numerischen Genauigkeit. Auch wenn Fließkommazahlen extrem flexibel sind, müssen Sie zur Kenntnis nehmen dass Sie einiges an Präzision verlieren wenn Sie viele Dezimalstellen abspeichern.

PureBasic unterstützt zur Zeit zwei Arten von Fließkommazahlen, das ist der normale 'Float' Typ sowie der 'Double' Typ. Der Unterschied zwischen den beiden Typen liegt in der Anzahl der Bits die diese verwenden um die Fließkommazahl abzuspeichern. Ein Float verwendet 32 Bit (4 Byte) und ein Double verwendet 64 Bit (8 Byte), das erklärt warum Doubles erheblich präziser beim speichern von großen Zahlen sind. Warum also nicht immer Doubles an Stelle von Floats verwenden? Nun, zum einen ist es die Geschwindigkeit. Floats können erheblich schneller aus dem Speicher gelesen werden, besonders wenn Sie große Float Arrays verwenden. Das liegt daran, dass Sie im Vergleich zu Doubles nur die Hälfte der Datenmenge aus dem Speicher lesen müssen. Üblicherweise tendieren Spiele zur Verwendung von Floats, während Programme die eine höhere Präzision verlangen eher zu Doubles tendieren.

Speichern von Fließkommazahlen in binärer Notation

Aufgrund der speziellen Natur von Fließkommazahlen, die einen Dezimalpunkt zusammen mit einer großen Zahl links, rechts oder auf beiden Seiten des Dezimalpunktes speichern müssen, wird eine andere Form der Speicherung in binärer Codierung verwendet. Diese Methode ist recht kompliziert und wird im 'IEEE 754' Standard für die binäre Codierung von Fließkommazahlen beschrieben. Ich versuche das ganze so einfach wie möglich zu beschreiben.

Zuerst einmal werden alle Fließkommazahlen in drei binären Teilen gespeichert. Bei diesen Teilen handelt es sich um das Vorzeichen, den Exponenten und die Mantisse. Diese drei Teile sind alle innerhalb der für den Typ verfügbaren Anzahl Bits untergebracht, egal ob Sie Float oder Double verwenden. Schauen Sie auf Abb. 47.

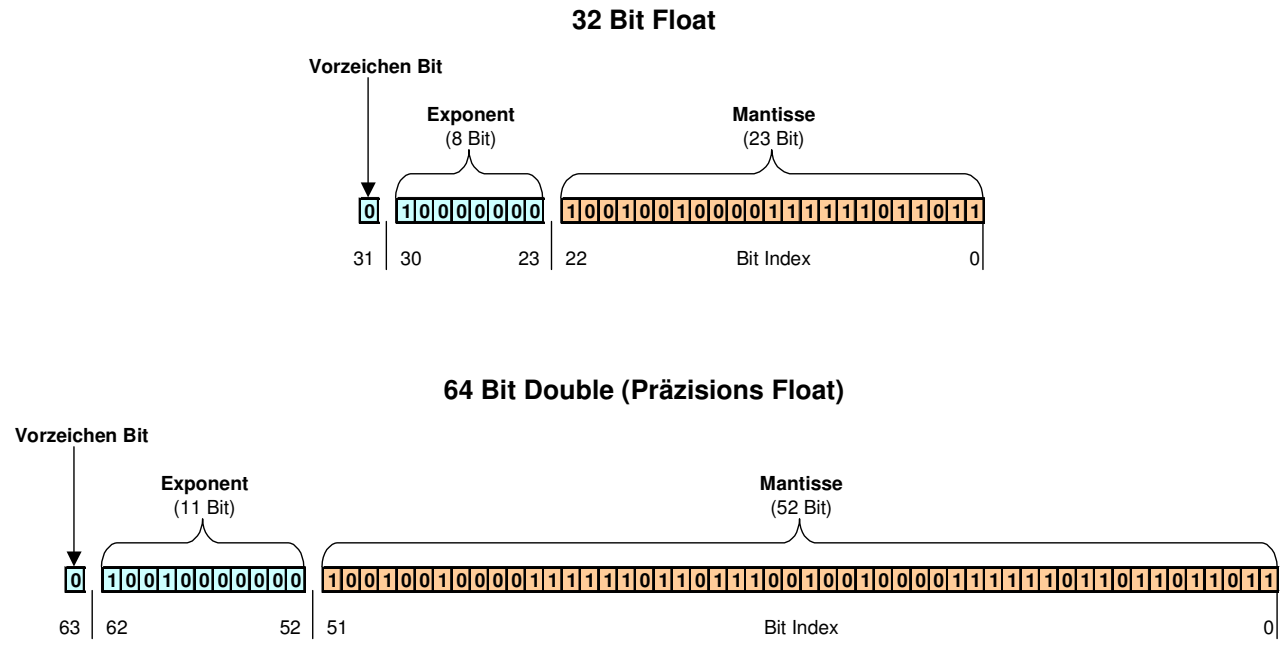


Abb. 47

Hier können Sie sehen wie der Typ intern aufgeteilt ist um alle drei Bereiche unterzubringen. Nun müssen wir nur noch verstehen wie eine Fließkommazahl codiert wird damit sie in diese Bereiche hineinpasst. Das Vorzeichen Bit ist wahrscheinlich der am einfachsten zu erklärende Bereich. Wenn die zu codierende Zahl positiv ist, dann hat das Vorzeichen Bit den Wert '0', bei einem negativen Wert hat es den Wert '1'. Lassen Sie uns mit einer Beispiel Zahl beginnen und ich zeige Ihnen wie diese für ein 32 Bit Float codiert wird. Lassen Sie uns die Zahl '10,625' nehmen.

Schritt Eins, Codieren der Zahl in Binärer Notation

Um die Zahl so zu Codieren, dass Sie in die binären Bereiche des Float Typs passt, müssen wir verschiedene Schritte ausführen. Der Erste ist die Konvertierung von beiden Seiten des Dezimalpunktes in binäre Notation, wie hier:

```
10,625 = 1010.101
```

Das sieht ein wenig kurios aus, da Sie wahrscheinlich noch nie einen Dezimalpunkt (oder genauer gesagt einen Radix Punkt) innerhalb einer Binärzahl gesehen haben. Keine Angst, diese Schreibweise dient nur dazu die Zahl in diesem Stadium zu visualisieren. Die binäre Codierung einer Zahl wie Dieser unterscheidet sich ein wenig von dem was ich zuvor erklärt habe, da wir mit einer Ganzzahl auf der einen Seite und einem

Bruch auf der anderen Seite des Radix Punktes hantieren. Der Ganzzahl Teil ist schnell konvertiert, da er der Standard Codierung für binäre Zahlen folgt die ich in Abb. 44 erklärt habe, zum Beispiel entspricht Dezimal '10' = Binär '1010'. Der Bruch Teil auf der anderen Seite verwendet eine ähnliche Methode, allerdings repräsentieren die einzelnen Bits in der Binärschreibweise andere Werte, es handelt sich hierbei selbst um Brüche statt um ganze Zahlen. Außerdem füllen wir diese Zahlen nicht mit unnötigen Nullen auf.

In der Standard vorzeichenlosen Binärcodierung beginnen die den Bits zugeordneten Werte mit '1' und wachsen nach links gehend um den Faktor '2'. Bei der Bruch Darstellung in Binärschreibweise beginnen die zugeordneten Werte mit '0,5' und werden dann um den Faktor '2' kleiner wenn Sie sich nach rechts bewegen. Schauen Sie auf Abb. 48.

Codierung eines Dezimalbruchs in Binärer Notation

Original Zahl: **10,625**

Die Ganzzahl wird folgendermaßen codiert:

Binärer Ziffern (Bits)	1	0	1	0	(8+2) = **10**
Zugeordnete Werte	**8**	**4**	**2**	**1**	

Radix Punkt

Der Bruch wird folgendermaßen codiert:

Binärer Ziffern (Bits)	1	0	1	(0.5 + 0.125) = **.625**
Zugeordnete Werte	**0.5**	**0.25**	**0.125**	

Radix Punkt

Abb. 48

Hier können Sie sehen, dass die zugeordneten Werte des Binärbruchs mit 0,5 beginnen und mit jeder weiteren Stelle nach rechts um den Faktor '2' kleiner werden, weshalb sie so berechnet sind: '0.5', '0.25', '0.125', usw. Je weiter Sie nach rechts gehen desto präziser kann eine Bruchzahl werden. Da unsere Zahl den Bruchteil '.625' enthält kann dieser einfach durch '0.5' + '0.125' dargestellt werden. An diesen Stellen tauchen dann auch die binären Einsen auf.

Das Wissen darüber wie eine Bruchzahl in Binärer Form dargestellt wird hilft Ihnen nicht notwendigerweise bei der Codierung dieser Zahl, besonders wenn Sie mit riesigen Dezimalbrüchen arbeiten. Glücklicherweise gibt es eine einfache Methode diese Zahlen zu Codieren: 'Der Verdopplungs-Trick'.

Zuerst nehmen wir unseren Ausgangsbruch '.625' und verdoppeln diesen. Dann überprüfen wir ob das Ergebnis größer oder gleich '1' ist. Wenn das der Fall ist notieren wir eine binäre '1'. Danach verwenden wir den Bruchteil der letzten Berechnung, verdoppeln diesen wieder und prüfen auch hier wieder das Ergebnis. Wenn an einer bestimmten Stelle das Ergebnis kleiner als '1' ist notieren wir eine binäre '0'. Hier ein Beispiel wie das in der Praxis funktioniert:

.625 * 2 = 1.25 - Das Ergebnis ist größer oder gleich '1', weshalb wir eine binäre '1' notieren
.25 * 2 = 0.5 - Das Ergebnis ist kleiner als '1', weshalb wir eine binäre '0' notieren
.5 * 2 = 1.0 - Das Ergebnis ist größer oder gleich '1', weshalb wir eine binäre '1' notieren

Wenn keine Bruchteile mehr übrig sind oder die verfügbaren Bit Stellen sind aufgebraucht, dann beenden Sie die Berechnung. Wenn Sie nun auf das binäre Ergebnis dieser Berechnung von oben nach unten schauen, dann sehen Sie die Binärkombination '101', genau das was wir für '.625' erwartet haben.

Jetzt wissen wir, wie wir einen Dezimalbruch in Binärform bringen und ich kann mit der Erklärung fortfahren, wie diese Zahlen in einem 32 Bit Float Typ codiert werden. Damit ist der erste Schritt beendet, wir haben unsere Zahl in Binärform gebracht:

```
10.625 = 1010.101
```

Schritt Zwei, Bewegen des Radix Punktes und Berechnung des Exponenten

In diesem Schritt müssen wir den Radix Punkt bewegen, da dieser nicht in der fertigen Binärzahl enthalten ist. Da der Radix Punkt definiert wo sich der Dezimalpunkt in der Dezimalversion dieser Zahl befindet, müssen wir uns eigentlich nur merken an welcher Stelle er sich befand und können ihn dann beim decodieren wieder an die Original Position zurücksetzen. Um das zu erreichen schieben wir den Radix Punkt in der Binärzahl so weit dass sich links von diesem Punkt nur noch eine '1' befindet. Weitere Nullen links von der letzten Eins können ignoriert werden, allerdings muss ihre Position gezählt werden wenn der Radix Punkt sie passiert. Schauen Sie auf dieses Beispiel:

```
1010.101      (Original)
1.010101      (Radix Punkt um 3 Stellen nach links verschoben)
```

Wenn Sie den Radix Punkt bewegen, müssen Sie sich merken um wie viele Stellen er verschoben wurde. Hier habe ich ihn '3' Stellen nach links verschoben um eine führende '1' auf seiner linken Seite zu behalten, die Zahl der Bewegungen ist also eine positive Zahl. Wenn die Original Zahl beispielsweise den Wert '0.0001101' hätte, dann müsste ich den Radix Punkt so lange nach rechts verschieben bis sich links von ihm eine führende '1' befindet (die Nullen werden ignoriert). Dadurch ergibt sich ein negativer Radix Bewegungswert von '-4'. (Den Grund für die führende 1 erkläre ich Ihnen in Kürze).

Die Anzahl der Stellen um Die Sie den Radix Punkt verschieben müssen, bis sich links von ihm nur noch eine '1' befindet, wird Exponent genannt. Dieser Exponent wird dann zu einer vordefinierten Zahl addiert (abhängig davon welche Größe von Fließkomma-Typ Sie verwenden) um sicherzustellen dass die codierte Zahl vorzeichenlos ist. Wenn Sie zum Beispiel einen 32 Bit Float Typ verwenden, müssen Sie den Wert '127' addieren, bei einem 64 Bit Double Typ müssen Sie '1023' addieren - diese Werte werden als Exponenten Bias bezeichnet.

In unserem Original Beispiel haben wir den Radix Punkt um '3' Stellen nach links verschoben (was einer positiven 3 entspricht), weshalb wir zu dieser '3' den Wert '127' addieren und als Ergebnis '130' erhalten. Diese Zahl wird dann nach der vorzeichenlosen Standardmethode in die Binärschreibweise umgewandelt (siehe Abb. 44), was '10000010' ergibt. Dieser Wert wird dann im Exponenten Bereich innerhalb des 32 Bit Float Wertes abgespeichert (siehe Abb. 47). Da wir die Anzahl der Radix Punkt Verschiebungen erfassen bedeutet das, dass der Exponenten Bereich eigentlich eine Begrenzung aufweist wie weit wir den Radix Punkt verschieben können. Diese Begrenzung liegt bei '127' Stellen nach links und '(-)126' Stellen nach rechts. Liegt der Wert über '127' wird die Zahl als 'unendlich' interpretiert, liegt der Wert unterhalb '-126' wird die Zahl als 'denormalisierte' Zahl bezeichnet. Dieser Spezialfall tritt auf wenn der Exponent auf '-126' gehalten wird und die führende '1' in eine '0' geändert wird. Im unteren Grenzfall wird die Zahl als '0' interpretiert.

Schritt Drei, Berechnung der Mantisse

Die Mantisse ist der Teil der Float Zahl, die den signifikanten Teil der Zahl enthält. In diesem Bereich wird die eigentliche Zahl gespeichert, der Exponent gibt nur an wo sich der Radix Punkt relativ zur Mantisse befindet. Nach dem verschieben des Radix Punktes in Schritt Zwei sollte unsere Zahl nun folgendermaßen aussehen: '1.010101'.

Um die Mantisse davon zu berechnen ignorieren wir zunächst die führende '1' der binären Zahl. Diese '1' muss nicht codiert werden, da sie immer vorhanden ist. Jede Zahl die wir für einen Fließkomma Typ Codieren hat immer diese '1'. Deshalb ist es absolut in Ordnung wenn wir diese Zahl beim codieren weglassen, wir müssen nur beim Decodieren daran denken sie wieder einzufügen. Das Weglassen dieser Zahl bringt uns einen Zugewinn von einem Bit in der Mantisse und hilft uns somit die Original Dezimalzahl etwas genauer abzulegen. Erinnern Sie sich, je mehr Bits wir zur Verfügung haben, desto präziser kann die Originalzahl abgespeichert werden.

Nach dem weglassen der '1' am Anfang der Zahl verliert auch der Radix Punkt seine Bedeutung und wir sind Diesen damit auch los. Jetzt sieht die Binärzahl folgendermaßen aus: '010101'. Das ist nun unsere Mantisse. Wenn die Mantisse kleiner als 23 Bit ist (was in unserem Beispiel der Fall ist), dann müssen wir die Mantisse bis auf eine Länge von 23 Bit mit Nullen auffüllen. Die Nullen müssen sich auf der rechten Seite befinden, wie hier: '01010100000000000000000'. Das ist nun die vollständige Mantisse die im Mantissen Bereich der 32 Bit Float Zahl abgelegt wird (siehe Abb. 47).

Somit sieht die Zahl '10,625' in Binärschreibweise folgendermaßen aus:

'0 10000010 01010100000000000000000'

Grenzen von Fließkommazahlen

Auch wenn Fließkommazahlen sehr flexibel sind, haben sie doch den Nachteil dass ihnen der Makel der Ungenauigkeit anhaftet. Das hängt mit der Art wie sie binär gespeichert werden zusammen. Wundern Sie sich nicht wenn Sie einen Wert aus einer Float Variable lesen und dieser vom übergebenen Wert abweicht. Maximal sieben Dezimalstellen können in einer Fließkommazahl mit einfacher Genauigkeit (Float) korrekt gespeichert werden, mit doppelter Genauigkeit (Double) sind es sechzehn. Die Präzision kann auch bei Berechnungen mit der Fließkommazahl beeinträchtigt werden, wobei sich die Fehler mit jeder weiteren Berechnung weiter vergrößern. Daran sollten Sie immer denken wenn Sie mit Floats und Doubles arbeiten.

Direkte Verwendung von Binärzahlen in PureBasic

Sie können Variablen direkt Binärwerte zuweisen, indem Sie das '%' Präfix verwenden. Wenn Sie das Prozentzeichen auf diese Art verwenden, sollten Sie es nicht mit dem Modulo Operator aus Kapitel 3 (Operatoren) verwechseln, auch wenn sie das gleiche Zeichen verwenden. PureBasic erkennt die binäre Zuweisung, da sich links kein Ausdruck und rechts kein Leerzeichen befindet. Hier ein Beispiel:

```
BinaerVariable.b = %01101101
Debug BinaerVariable
```

Allen numerischen Typen von PureBasic können Sie auf diese binäre Weise Werte zuweisen.

Zeiger

Ich denke es ist fair Ihnen zu sagen, dass Zeiger in Computer Programmen eine Menge Leute verrückt machen. Ich kann mir nicht erklären warum, da Zeiger eine immense Macht verleihen können wenn Sie auf verschiedene Programmierprobleme treffen. Vielleicht ist es die Angst vor dem Unbekannten das die Menschen abschreckt? Ich weiß es nicht, aber ich weiß dass die meisten Menschen sich wundern warum sie sich so lange vor den Zeigern gefürchtet haben, nachdem ihnen die Thematik verständlich erklärt wurde. Dieser nächste Abschnitt handelt von Zeigern, und davon wie PureBasic diese erstellt und verwendet, und welche Vorteile sie für uns Programmierer bringen.

Was ist also ein Zeiger? Nun, ganz einfach, ein Zeiger in PureBasic ist eine Variable die eine numerische Adresse eines bestimmten Speicherbereiches enthält. So einfach ist das! Was also sind Speicheradressen? Denken Sie an all die Bytes im Speicher, jedes einzelne ist mit einer einmaligen Zahl verknüpft, als wären wir in einem riesigen Array. Jedes Byte im Speicher kann über seinen zugeordneten Wert (Adresse) betrachtet werden. Da in modernen Computern Milliarden von Bytes vorhanden sind, bestehen diese numerischen Adressen üblicherweise aus recht langen Zahlen.

Ermitteln einer Speicheradresse

Nun wissen wir, dass ein Zeiger nur eine Variable ist die eine Speicheradresse enthält. Jetzt müssen wir nur noch wissen, wie wir die Speicheradresse von etwas nützlichem ermitteln. Dafür verwenden wir spezielle Ein-Zeichen Funktionen, die als Präfix vor dem entsprechenden Datenobjekt Namen die Adresse dieses Objektes zurückgeben.

Speicher Adressen Funktionen

Datenobjekt	Funktion	Anwendungsbeispiel
Variablen	@	@VariablenName
Arrays	@	@ArrayName()
Prozeduren	@	@ProzedurName()
Sprungmarken	?	?SprungmarkenName

Abb. 49

In Abb. 49 können Sie die beiden speziellen Speicheradressen Funktionen sehen, und mit welchen Datenobjekten sie verwendbar sind. Die '@' Funktion ist die flexiblere der beiden, da sie mit Variablen, Arrays und Prozeduren funktioniert. Die '?' Funktion ist für Sprungmarken reserviert. Hier ein einfaches Beispiel wie Sie die Speicheradresse einer Byte Variable ermitteln:

```
MeineByteVariable.b = 100
Debug MeineByteVariable
Debug @MeineByteVariable
```

In diesem Beispiel habe ich eine Byte Variable mit dem Namen 'MeineByteVariable' erstellt ,der ich den Wert '100' zugewiesen habe, was bis dahin alles klar sein sollte. In der letzten Zeile des Beispiels ermittle ich die Speicheradresse dieser Variable indem ich das '@' vor den Variablen Namen setze. Wenn ich das '@' auf diese Weise verwende erhalte ich als Rückgabewert die Speicheradresse von 'MeineByteVariable', die ich dann im Debug Ausgabefenster anzeige. Wenn Sie das obige Beispiel starten und auf das Debug Ausgabefenster schauen, dann sehen Sie dass der Originalwert '100' und darunter die Adresse der Variable angezeigt wird. Die Adresse ist in der Regel eine große Zahl, üblicherweise sieben oder acht Stellen lang, und die Adresse kann ihren Wert ändern, abhängig vom Computer auf dem Sie das Programm starten. Das hängt mit der teilweise unterschiedlichen Art der Speicher Reservierung für die Variable zusammen.

Wenn ich das Programm auf meinem Computer starte, erhalte ich folgende Anzeige im Debug Ausgabefenster:

```
100
4389540
```

Das sagt mir, dass die Byte Variable die ich zum speichern des Wertes '100' verwendet habe an der Speicheradresse '4389540' zu finden ist.

Mit Hilfe dieser speziellen Speicheradressen Funktionen ist es möglich die Adressen von allen Variablen Typen, Arrays, Prozeduren und Sprungmarken zu ermitteln, was sehr nützlich sein kann. Hier ein Beispiel das in der normalen Welt recht nutzlos wäre, aber es demonstriert die Ermittlung der Speicheradressen von all diesen Datenobjekten und gibt diese im Debug Ausgabefenster aus:

```
ByteVariable.b   = 1
WordVariable.w   = 2
LongVariable.l   = 3
QuadVariable.q   = 4
FloatVariable.f  = 5
DoubleVariable.d = 6
StringVariable.s = "Sieben"
Dim LongArray.l(8)

Procedure Prozedur(Test.i)
  Debug "Teste meine Prozedur."
EndProcedure

Debug "Adresse der Byte Variable: "   + Str(@ByteVariable)
Debug "Adresse der Word Variable: "   + Str(@WordVariable)
Debug "Adresse der Long Variable: "   + Str(@LongVariable)
Debug "Adresse der Quad Variable: "   + Str(@QuadVariable)
Debug "Adresse der Float Variable: "  + Str(@FloatVariable)
Debug "Adresse der Double Variable: " + Str(@DoubleVariable)
Debug "Adresse der String Variable: " + Str(@StringVariable)
Debug "Adresse des Arrays: "          + Str(@LongArray())
Debug "Adresse der Prozedur: "        + Str(@Prozedur())
Debug "Adresse der Sprungmarke: "     + Str(?Sprungmarke)

DataSection
  Sprungmarke:
  Data.s "Test"
EndDataSection
```

Beachten Sie, dass bei der Ermittlung der Adressen von Arrays, Prozeduren und Sprungmarken kein "schmückendes Beiwerk", wie zum Beispiel Typ Identifikatoren, Array Dimensionen, Parameter, usw., angehängt werden muss, auch wenn das Array in obigem Beispiel so definiert wurde:

```
...
Dim LongArray.l(8)
...
```

Um seine Speicheradresse zu ermitteln müssen Sie nur seinen Namen und die Klammern am Ende verwenden, wie hier:

```
@LongArray()
```

Auf die gleiche Weise ermitteln Sie die Startadresse einer Prozedur, Sie benötigen nur den Namen und die Klammern am Ende:

```
@Prozedur()
```

Es werden keine Typen oder Parameter benötigt. Das Selbe gilt für Sprungmarken. Auch wenn diese mit einem Doppelpunkt am Ende definiert werden, so wird dieser Doppelpunkt bei der Verwendung einer Speicheradressen Funktion ignoriert, wie hier:

```
?Sprungmarke
```

Die Ermittlung der Speicheradressen all dieser Datenobjekte ist einfach, da die Speicheradressen Funktionen auf eine sehr einfache Art arbeiten. Gleich zu Beginn, damit wir uns richtig verstehen, ich empfehle Ihnen alle ermittelten Speicheradressen in Variablen zu speichern um den Code sauber und übersichtlich zu halten, und denken Sie immer daran diesen Variablen aussagekräftige Namen zu geben.

Erstellen und Benennen von Zeigern

Wie ich bereits zu Beginn erwähnt habe ist ein Zeiger eine Variable, die die numerische Adresse eines Speicherortes enthält. Ich habe allerdings nicht erwähnt, dass ein Zeiger eine spezielle Art von Variable ist, die ein paar bestimmten Regeln folgt. Erstens beginnen die Namen von Zeigern in PureBasic mit einem Asterisk Zeichen '*', wie hier

```
*Zeiger
```

Zweitens können Zeiger unterschiedliche Größen haben, abhängig davon auf welcher Computer Architektur das Programm kompiliert wurde. Auf einem 32 Bit System belegt ein Zeiger zum Beispiel 4 Byte im Arbeitsspeicher, während ein Zeiger auf einem 64 Bit System 8 Byte belegt um eine Adresse zu speichern.

Weiterhin sind Zeiger nicht mit einem bestimmten Typ verknüpft. Das bedeutet, wenn Sie einen Zeiger mit einem Asterisk zu Beginn und einem numerischen eingebauten Typ erstellen, dann wird der eingebaute Typ ignoriert um sicherzustellen dass der Zeiger die richtige Größe für diese bestimmte Architektur hat. Wenn Sie zum Beispiel einen Zeiger als Byte Typ definieren, dann wird diese Typen Zuordnung ignoriert, wie hier:

```
*Zeiger.b = @Variable
```

Der Byte Typ wird ignoriert und der Zeiger verwendet 4 Byte auf einem 32 Bit System und 8 Byte auf einem 64 Bit System.

Das Asterisk wird als Teil des Zeiger Namen verwendet, ist also ein permanentes Zeichen. Wenn Sie erst einmal einen Zeiger mit dem Asterisk definiert haben, dann müssen Sie es als Teil des Namens immer mit angeben wenn Sie den Zeiger verwenden. Diese beiden Variablen haben zum Beispiel keinerlei Beziehung zueinander, es handelt sich um zwei völlig verschiedene Variablen:

```
Variable.i
*Variable.i
```

Auch wenn der einzige Unterschied zwischen diesen beiden Variablen das Asterisk ist, die erste Variable ist vom Typ Integer, während die zweite Variable ein Zeiger ist (definiert durch das Asterisk). In anderen Sprachen wird ein Asterisk vor dem Variablennamen verwendet um die Speicheradresse zu ermitteln und ohne Asterisk wird der Wert der Variable empfangen, es handelt sich also um die gleiche Variable. Es ist wichtig zu verstehen, dass PureBasic nicht diese erweiterte Funktionalität besitzt, hier handelt es sich um zwei unterschiedliche Variablen.

Die Verwendung des Asterisk um eine Zeigervariable zu erzeugen ist eine gute Möglichkeit diese von anderen Typen in PureBasic zu differenzieren, und hilft Ihnen beim lesen des Codes schneller zu erkennen was der eigentliche Inhalt der Variable ist. In der Hilfe Datei sind alle Beispiele die mit Zeigern arbeiten auf diese Weise gestaltet, um dem Programmierer schnell ersichtlich zu machen wo ein Zeiger benötigt wird. Schauen Sie sich ein paar Syntax Beispiele der Speicher Befehle in der Hilfe an (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Memory), dort finden Sie überall Asterisken.

Wenn wir diese Konvention verwenden, können wir das erste Beispiel so neu schreiben:

```
MeineByteVariable.b = 100
*SpeicherAdresse    = @MeineByteVariable
Debug MeineByteVariable
Debug *SpeicherAdresse
```

Erkennen Sie das Asterisk vor dem Zeiger Namen? Auf diese Weise können Sie Zeiger leicht von anderen Variablen unterscheiden.

Zugriff auf den Speicher in einer strukturierten Weise mittels eines Zeigers

In diesem nächsten Beispiel zeige ich Ihnen wie Sie einen Zeiger dazu verwenden können als strukturierte Variable zu agieren. Dieser Zeiger hat die gleiche Größe wie jeder andere, allerdings können wir auf den Speicherbereich auf den er zeigt in einer etwas strukturierteren Weise zugreifen.

```
Structure DETAILS
  Alter.l
  Groesse.l
EndStructure

Meine.DETAILS
Meine\Alter   = 37
Meine\Groesse = 178

*Zeiger.DETAILS = @Meine
Debug *Zeiger\Alter
Debug *Zeiger\Groesse
```

Hier können Sie sehen dass ich eine Struktur mit dem Namen 'DETAILS' erstellt habe die zwei Felder vom Typ Long enthält. Unmittelbar danach habe ich eine Variable mit Namen 'Meine' erstellt, die diese Struktur als Typ verwendet und habe den Feldern jeweils einen Wert zugewiesen. Das ist nun die Stelle an der die Verwendung des Asterisk hilfreich bei der Zeiger Erstellung sein kann. Wenn wir einen Zeiger auf eine strukturierte Variable erstellen weisen wir diesem die Speicheradresse folgendermaßen zu:

```
*Zeiger.DETAILS = @Meine
```

Dadurch wird ein Zeiger mit Namen '*Zeiger' erstellt und ihm wird die Speicheradresse der 'Meine' strukturierten Variable zugewiesen. Da wir eine Struktur als Typ bei der Erstellung der '*Zeiger' Variable verwendet haben, können wir auf die Speicheradresse ebenfalls in einer strukturierten Art zugreifen.

Deshalb geben diese beiden Zeilen Daten im Debug Ausgabefenster aus:

```
Debug *Zeiger\Alter
Debug *Zeiger\Groesse
```

die eigentlich in der 'Meine' Strukturvariable liegen. Wir erhalten diese Daten über unseren neu erstellten Zeiger. Noch einmal, der hier erstellte Zeiger ist nur eine Variable die eine numerische Speicheradresse enthält, es werden keine Daten dupliziert. Abb. 50 verdeutlicht wie dieser Vorgang im Speicher funktioniert.

Die 'Meine.DETAILS' strukturierte Variable im Speicher

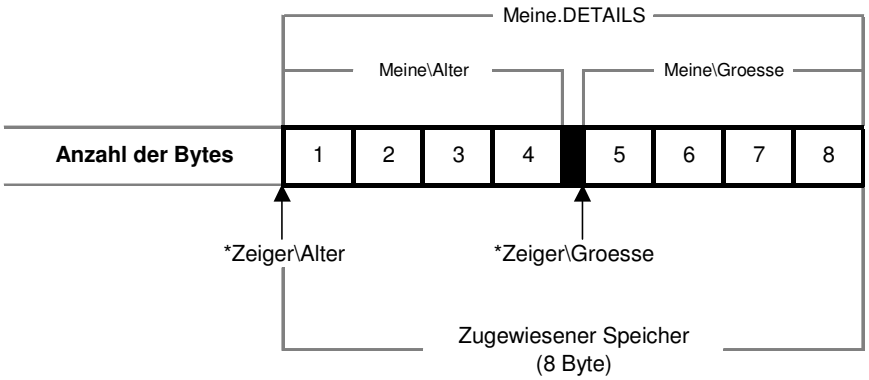


Abb. 50

Für was ist das nun brauchbar? Es kann zum Beispiel verwendet werden, um mehr als einen Wert von einer Prozedur zurückzugeben. Sie können zum Beispiel eine 'Static' strukturierte Variable in Ihrer Prozedur erstellen und Dieser Werte zuweisen. Dann geben Sie als Rückgabewert der Prozedur die Speicheradresse zu dieser Variable zurück. Mit dieser Adresse können Sie einen Zeiger auf eine Strukturierte Variable erstellen, der Ihnen den Zugriff auf die in der Prozedur zugewiesenen Werte ermöglicht, wie z.B. hier:

```
Structure MAHLZEITEN
  Fruehstueck.s
  Mittagessen.s
  Abendessen.s
EndStructure

Procedure.i HoleMahlzeiten()
  Static Gary.MAHLZEITEN
  Gary\Fruehstueck = "Cornflakes"
  Gary\Mittagessen = "Käse Sandwiches"
  Gary\Abendessen  = "Spaghetti Bolognese"
  ProcedureReturn @Gary
EndProcedure

*Gary.MAHLZEITEN = HoleMahlzeiten()

Debug *Gary\Fruehstueck
Debug *Gary\Mittagessen
Debug *Gary\Abendessen
```

Wir verwenden das 'Static' Schlüsselwort um die Daten in der 'Gary' strukturierten Variable zu erhalten wenn die Prozedur verlassen wird, anderenfalls werden die Daten zerstört. Die von der Prozedur zurückgegebene Speicheradresse weisen wir dann einem strukturierten Variablen Zeiger mit Namen '*Gary' zu (beachten Sie das Asterisk). Wir können dann auf diesen Speicherbereich unter Verwendung der 'MAHLZEITEN' Struktur zugreifen. Wie Sie sehen ist das eigentlich ganz praktisch.

Um die Flexibilität noch zu erhöhen, Sie können die Adresse jedes beliebigen Variablen Typs einem strukturierten Variable Zeiger zuweisen, hierzu kann jeder in PureBasic eingebaute Typ verwendet werden. Hier ein Beispiel:

```
Structure SCREENKOORDINATEN
  x.w
  y.w
EndStructure

Koordinaten.l             = %00000011000000000000010000000000
*Screen.SCREENKOORDINATEN = @Koordinaten

Debug *Screen\x
Debug *Screen\y
```

Hier habe ich eine 32 Bit Variable vom Typ Long mit dem Namen 'Koordinaten' definiert und ihr einen binären Wert zugewiesen. Dann habe ich die Speicheradresse dieser Variable einem neu erstellten strukturierten Variable Zeiger mit Namen '*Screen' zugewiesen. Da die für den Zeiger verwendete Struktur zwei 16 Bit Word Typen enthält, passt der 32 Bit Wert auf den der Zeiger verweist nicht in ein Feld der Struktur, wenn wir ihn aber in der Mitte teilen überlappt er beide Felder. Das erlaubt es mir zwei Werte aus dieser einen Variable auszulesen und ich erhalte die zwei 'x' und 'y' Werte.

Übergabe einer Strukturierten Variable in eine Prozedur mittels eines Zeigers

Jede strukturierte Variable kann als Zeiger an eine Prozedur übergeben werden, was es der Prozedur ermöglicht diese Variable zu verändern. Das ist in PureBasic momentan die einzige Möglichkeit eine strukturierte Variable an eine Prozedur zu übergeben. Wenn Sie eine strukturierte Variable auf diese Art übergeben, dann wird die Variable eigentlich 'Per Referenz' übergeben, sehr ähnlich der Übergabe von Arrays und Listen. Das bedeutet, dass eigentlich keine Werte in einen Parameter 'kopiert' werden. Stattdessen wird ein Zeiger an die Prozedur übergeben und Diese verändert dann die Original strukturierte Variable. Sie müssen auf jeden Fall eine Sache beachten wenn Sie mit dieser Methode arbeiten. Wenn Sie eine strukturierte Variable an eine Prozedur übergeben, dann hat der Parameter der Prozedur vielleicht einen anderen Namen als ihre strukturierte Variable die Sie übergeben, aber egal welchen Namen Sie übergeben, es wird immer die Original übergebene Strukturvariable verändert. Hier ein Beispiel:

```
Structure KOORDINATEN
  x.l
  y.l
EndStructure
Punkt.KOORDINATEN
```

```
Procedure ErhoeheWerte(*Variable.KOORDINATEN)
  *Variable\x + 10
  *Variable\y + 10
EndProcedure

Punkt\x = 100
Punkt\y = 100

ErhoeheWerte(@Punkt)
Debug Punkt\x
Debug Punkt\y
```

Hier habe ich eine Struktur mit dem Namen 'KOORDINATEN' definiert und dann mit ihr eine strukturierte Variable mit Namen 'Punkt' erstellt. Nachdem ich allen Feldern dieser Variable Werte zugewiesen habe, übergebe ich die Speicheradresse dieser Variable an die 'ErhoeheWerte()' Prozedur.

In der Definition der 'ErhoeheWerte()' Prozedur habe ich einen Parameter in folgender Form definiert: '*Variable.KOORDINATEN'. Wenn ein Parameter auf diese Weise definiert wird, erwartet die Prozedur einen Zeiger von einer strukturierten Variable, die mit der Struktur 'KOORDINATEN' als Typ erstellt wurde. Der Name '*Variable' kann beliebig abgeändert werden (so lange Sie das Asterisk als Präfix definiert lassen), diese Variable wird lediglich dazu verwendet den übergebenen Zeiger zu bearbeiten. Wenn ich in der obigen Prozedur folgendes mache:

```
...
*Variable\x + 10
*Variable\y + 10
...
```

dann tue ich eigentlich das:

```
Punkt\x + 10
Punkt\y + 10
```

da die Prozedur die Daten an der Original Speicheradresse verändert.

Die Verwendung dieser Methode zur Übergabe von strukturierten Variablen an Prozeduren kann sehr praktisch sein, wenn Sie viele gruppierte Informationen übergeben wollen, zum Beispiel Adress-Details, Konfigurations-Details oder irgendwelche anderen gruppierten Einstellungen oder Werte. Wenn Sie diese Methode verwenden können Sie unter Umständen auch die Anzahl der an eine Prozedur zu übergebenden Parameter drastisch reduzieren, da Sie alle Parameter in einer strukturierten Variable übergeben. Als Beispiel aus dem echten Leben, ich habe vor ein paar Jahren eine E-Mail Prozedur geschrieben an die ich eine strukturierte Variable übergeben konnte, die die Details zum Mail Account zusammen mit Empfänger, Betreff und Nachricht, usw. enthielt ... dadurch werden einige Sachen ein wenig übersichtlicher und leichter handhabbarer.

Lesen und schreiben von Werten mittels Zeigern

Wenn Sie eine Speicheradresse ermitteln haben Sie die Möglichkeit diese auf vielfältige Art und Weise zu nutzen. Ohne Frage ist der übliche Weg das Lesen von Werten aus dem entsprechenden Speicherbereich oder das Speichern von Daten an dieser Stelle. Dieser Vorgang wird 'Peeken' und 'Poken' genannt. Sie können einen 'Peek' Befehl verwenden um Daten von einer Speicherstelle zu lesen, und einen 'Poke' Befehl um Daten an eine Speicherstelle zu schreiben.

In PureBasic gibt es elf 'Peek' Befehle und elf 'Poke' Befehle, um alle eingebauten Datentypen zu unterstützen. Alle Befehle haben annähernd die gleichen Namen, sie unterscheiden sich nur im letzten Buchstaben des Befehls. Bei diesem Buchstaben handelt es sich immer um die jeweilige Typenkennzeichnung die Sie auch bei der Typenvergabe von Variablen verwenden. Wenn Sie zum Beispiel ein Byte von einer bestimmten Speicheradresse auslesen möchten, dann müssen Sie den 'PeekB()' Befehl verwenden. Wie das 'B' im Namen suggeriert, wird von dieser Speicheradresse ein Byte gelesen. Auf die gleiche Art würden Sie einen String an eine bestimmte Speicheradresse schreiben, in diesem Fall würden Sie den 'PokeS()' Befehl verwenden. Mehr Informationen zu diesen zweiundzwanzig Befehlen finden Sie in der PureBasic Hilfe (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Memory).

'Peeken' von Werten die an bestimmten Speicherstellen liegen

Die Verwendung eines 'Peek' Befehls ist relativ selbsterklärend, Sie übergeben ihm eine Speicheradresse (zum Beispiel aus einem Zeiger) und der Peek Befehl gibt die entsprechenden Daten von dieser Stelle zurück. Um einen bestimmten Datentyp zu lesen müssen Sie den damit verknüpften Peek Befehl

verwenden. Hier ein Beispiel, das ein Byte von einer bestimmten Speicherstelle mittels eines 'PeekB()' Befehls ausliest:

```
Gewicht.b  = 30
*Gewicht   = @Gewicht
LeseWert.b = PeekB(*Gewicht)
Debug LeseWert
```

Hier weise ich einer Byte Variable mit dem Namen 'Gewicht' den Wert '30' zu. In der nächsten Zeile weise ich die Speicheradresse dieser Variable einem Zeiger mit dem Namen '*Gewicht' zu (beachten Sie das Asterisk). Die nächste Zeile enthält den 'PeekB()' Befehl, der den eigentlichen Byte Wert von der Speicheradresse aus dem Zeiger ausliest und zurückgibt. Sie sind nicht gezwungen einen Zeiger für den Adress-Parameter einzugeben (auch wenn es aus Gründen der Übersichtlichkeit empfehlenswert ist), wir können das Beispiel auch folgendermaßen schreiben:

```
Gewicht.b  = 30
LeseWert.b = PeekB(@Gewicht)
Debug LeseWert
```

Hier verwende ich die Byte Variable direkt mit der Speicheradressen Funktion, komplett ohne Zeiger. Ich denke es hängt an Ihnen welche Vorgehensweise Sie bevorzugen. Am Ende des Tages benötigen alle 'Peek' Befehle eine Speicheradresse als Parameter, diese kann als definierter Zeiger oder als Rückgabewert einer Speicherfunktion zugeführt werden.

'Poken' von Werten in bestimmte Speicherbereiche

Das 'Poken' von Werten in Speicherbereiche ist genauso einfach wie die Arbeit mit dem 'Peek' Befehl. Zum schreiben einer Speicherstelle müssen Sie einen neuen Wert und die Speicheradresse übergeben, an die der Wert geschrieben werden soll. Hier ein Beispiel zum 'Poken' eines neuen Long Wertes in den Speicherbereich einer bestehenden Long Variable:

```
Gewicht.l = 1024
*Gewicht  = @Gewicht
PokeL(*Gewicht, 333)
Debug Gewicht
```

Hier beginnen wir mit dem Original Startwert '1024' den wir der Variable 'Gewicht' zugewiesen haben. Dann habe ich einen Zeiger mit Namen '*Gewicht' erstellt, der die Speicheradresse von 'Gewicht' enthält. Danach habe ich den 'PokeL()' Befehl verwendet um den Wert '333' an die Speicherstelle zu schreiben, die im ersten Parameter spezifiziert wurde. Der neue Wert von 'Gewicht' wird dann im Debug Ausgabefenster ausgegeben.

Ein Wort der Warnung wenn Sie neue Werte 'Poken'

Seien Sie sehr vorsichtig wo Sie ihre neuen Werte hinschreiben. Die Verwendung der PureBasic 'Memory' Befehle kann sehr gefährlich sein wenn Sie nicht wissen was Sie tun. Die Poke Befehle führen keine interne Prüfung durch die Ihnen mitteilt ob Sie an diese Speicherstelle gefahrlos einen Wert schreiben können. Sie können eine zufällige Zahl als Speicheradresse an einen Poke Befehl übergeben und er wird fröhlich einen Wert an diese Stelle im Speicher schreiben. Es ist ein Zweischneidiges Schwert, da dieser Befehl sehr hilfreich bei der Neuorganisierung von Werten im Speicher sein kann, auf der anderen Seite kann eine fehlerhafte Handhabung aber auch immense Schäden anrichten. Wenn Sie einen Fehler machen überschreiben Sie vielleicht eine Speicherstelle von einem wichtigen Programm und dieses liest einen falschen benötigten Wert. Das Resultat ist ein Programmabsturz oder in gravierenden Fällen sogar ein Systemabsturz.

Um Werte sicher in den Speicher zu Poken sollten Sie als Ziel eine bestehende Variable oder einen Array Speicherbereich auswählen oder sie verwenden reservierten Speicher den, Sie mit dem 'AllocateMemory()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Memory → AllocateMemory) anfordern. Stellen Sie sicher dass alle Werte die Sie Poken nicht mehr Speicher benötigen als ihnen zur Verfügung steht. Es wäre zum Beispiel schlechte Praxis einen Long Wert in den Speicherbereich einer Byte Variable zu Poken, da Sie 3 Byte mehr in diesen Bereich schreiben als Sie eigentlich dürfen. Das wird im ersten Moment funktionieren, da keine Fehlerprüfung stattfindet, aber Sie überschreiben 3 Byte potentiell wichtiger Daten oder was auch immer sich an dieser Stelle befand.

Verwendung einer Speicheradresse als Start Index

Wenn Sie aus fortlaufenden Speicherblöcken Werte lesen möchten oder Werte in diese hineinschreiben möchten, so ist es möglich die Adresse des Start Bytes als Start Index für den Rest des Blockes zu

verwenden. Das ist besonders nützlich wenn Sie einen Zeiger auf ein Array verwenden. Schauen Sie auf dieses Beispiel:

```
Dim Zahlen.l(2)

Zahlen(0) = 100
Zahlen(1) = 200
Zahlen(2) = 300

*ArrayZeiger = @Zahlen()

Debug PeekL(*ArrayZeiger)
Debug PeekL(*ArrayZeiger + 4)
Debug PeekL(*ArrayZeiger + 8)
```

Nach der Erstellung des Arrays 'Zahlen' mit Variablen vom Typ Long weise ich jedem Indize einen numerischen Wert im Bereich '100' bis '300' zu. Danach erstelle ich einen Array Zeiger mit dem Namen '*ArrayZeiger' der die Speicheradresse des Arrays enthält.

Wenn Sie ein Array auf diese Weise erstellen, dann werden alle Indizes in einem fortlaufenden Speicherblock abgelegt, so dass alle Daten Einer hinter dem Anderen abgespeichert werden. Die zurückgegebene Adresse der Speicheradressen Funktion ist die Adresse des ersten Index. Effektiv enthält der Zeiger mit dem Namen '*ArrayZeiger' die Speicheradresse des Array Index 'Zahlen(0)'.

Wenn wir dann folgende Code Zeile verwenden

```
...
Debug PeekL(*ArrayZeiger)
...
```

dann lesen wir den Wert vom Speicherbereich des ersten Index. Sie können am ersten angezeigten Wert im Debug Ausgabefenster erkennen, dass das stimmt. Um andere Werte aus dem Array zu lesen müssen wir den Zeiger als Startwert verwenden und zu diesem Wert die Anzahl der Bytes addieren, um die wir uns durch den Speicher bewegen möchten. Wir wissen dass der Long Typ vier Bytes im Speicher belegt, deshalb können wir den Zeigerwert um '4' erhöhen um den nächsten Array Wert aus dem Speicher zu lesen. Die folgenden Codezeilen aus dem obigen Beispiel machen genau das, sie lesen weitere Werte aus dem Array indem sie den Array Zeiger als Startpunkt verwenden.

```
...
Debug PeekL(*ArrayZeiger + 4) ;Zeigt den zweiten Long Wert an: '200'
Debug PeekL(*ArrayZeiger + 8) ;Zeigt den dritten Long Wert an: '300'
```

Ich habe diesen Vorgang in Abb. 51 etwas visualisiert. Wenn ich ein Array aus Bytes verwende, muss ich den Zeiger natürlich nur um '1' erhöhen um das zweite Byte zu lesen und um zwei um das dritte Byte zu lesen, usw. Das gleiche gilt für Word Typen, hier müsste ich den Zeiger jedes Mal um '2' erhöhen.

Das 'Zahlen' Array im Speicher

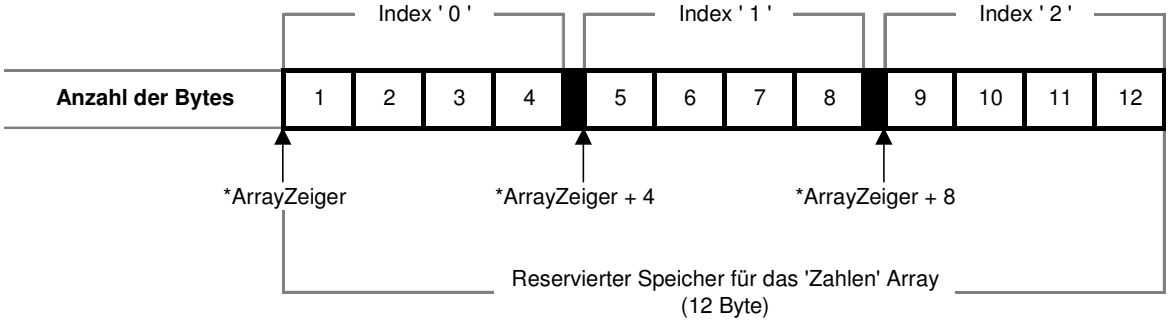


Abb. 51

Threads

Ein Thread ist eine Sequenz von Anweisungen die in einem separaten Task (eigenständiger Prozess) ausgeführt werden aber trotzdem zum Ursprungsprogramm dazugehören. Jedes Programm kann mehrere Threads parallel starten und jeder bearbeitet eine andere Aufgabe, wie zum Beispiel das Warten auf Ereignisse von einer Zeitaufwendigen Aufgabe. Threads sind Ableger des Hauptprogramms, die unabhängig von diesem Anweisungen abarbeiten können. Wenn das Hauptprogramm geschlossen wird, werden alle Threads ebenfalls angehalten und beendet. Welches Programm verwendet dann Threads? Nun, jedes Programm kann sie potentiell nutzen, aber hauptsächlich werden sie in Programmen verwendet die mehr als eine Sache gleichzeitig erledigen müssen. Ein Beispiel wäre ein Suchprogramm, das Dateien auf Ihrer Festplatte sucht. Den Code zum Suchen würden Sie dann so schreiben, dass er in einem Thread ausgeführt wird. Dadurch würden das Neuzeichnen der Oberfläche oder die Ereignis Behandlung in der Hauptschleife nicht blockiert werden. Das Suchen könnte unabhängig durchgeführt werden, ohne den Nebeneffekt, dass das Hauptprogramm nicht mehr auf Eingaben reagiert. Ein weiteres Beispiel wäre ein Programm das große Dateien bearbeitet. Da die Bearbeitung dieser Dateien in der Regel etwas länger dauert, würden Sie diese Bearbeitung in einen Thread auslagern, während das Hauptprogramm zwischenzeitlich andere Dinge erledigen kann. Das sind zwei Beispiele die verdeutlichen wo Threads bei der Lösung von Programmierproblemen hilfreich sein können.

Die Verwendung von Threads

Bevor ich weiter fortfahre Ihnen zu zeigen wie Sie Threads verwenden, zeige ich Ihnen ein einfaches Programm ohne Threads, damit Sie einen Vergleich haben. Dieses Programm ruft eine Prozedur zweimal auf um Werte in zwei verschiedene ListViewGadgets einzufügen. Beachten Sie, dass Sie nach dem drücken des Start Knopfes weder das Fenster bewegen können, noch können Sie den aktuell ins Gadget eingetragenen Text betrachten, bis beide Prozeduren vollständig abgearbeitet sind.

```
Enumeration
  #FENSTER_HAUPT
  #LISTE_EINS
  #LISTE_ZWEI
  #KNOPF_TEST
EndEnumeration

Procedure TextEinfuegen(Gadget.i)
  For x.i = 1 To 25
    AddGadgetItem(Gadget, -1, Str(x))
    Delay(100)
  Next x
EndProcedure

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER_HAUPT, 0, 0, 290, 200, "Thread Test", #FLAGS)
  ListViewGadget(#LISTE_EINS, 10, 10, 130, 150)
  ListViewGadget(#LISTE_ZWEI, 150, 10, 130, 150)
  ButtonGadget(#KNOPF_TEST, 95, 170, 100, 20, "Starte Test")
  Repeat
    Ereignis.i = WaitWindowEvent()
    Select Ereignis
      Case #PB_Event_Gadget
        Select EventGadget()
          Case #KNOPF_TEST
            TextEinfuegen(#LISTE_EINS)
            TextEinfuegen(#LISTE_ZWEI)
        EndSelect
    EndSelect
  Until Ereignis = #PB_Event_CloseWindow
EndIf
End
```

Sie werden ebenfalls feststellen dass die beiden Prozeduraufrufe von 'TextEinfuegen()' in der Hauptschleife nacheinander ausgeführt werden. Das heißt, dass die erste Prozedur beendet sein muss bevor die zweite Prozedur beginnen kann. Genau so funktionieren Prozedurale Programme. In diesem Fall betrifft uns das nicht weiter, da wir ohnehin keine Aktualisierung in den ListViewGadgets sehen, was an der 'Delay()' Anweisung in der 'TextEinfuegen()' Prozedur liegt.

Kompilieren von Threadsicheren Programmen

PureBasic hat eine Compiler Einstellung zum erstellen von Threadsicheren Executables, deren Verwendung Pflicht ist, wenn Sie stressfrei mit Threads programmieren wollen. Wenn Sie den Compiler auf der Kommandozeile verwenden, können Sie die '/Thread' Compiler Option verwenden oder Sie schalten den Threadsicheren Modus im 'Compiler-Optionen' Dialog der IDE an (Menü: Compiler → Compiler-Optionen... → Compiler-Optionen → Threadsicheres Executable erstellen).

Wenn dieser Modus nicht eingeschaltet ist, und Sie kompilieren und starten Ihr Programm, dann ist Ihr fertiges Executable nicht hundertprozentig Threadsicher. Da alle Threads in Ihrem Programm auf gemeinsame Daten innerhalb Ihres Executables zugreifen, kann es passieren dass in diesen Daten durch gleichzeitige Zugriffe Fehler auftreten. Dieses Problem tritt besonders häufig bei der Verwendung von Strings auf. Ohne die Threadsichere Compiler Option werden Strings in Ihrem Programm unter Umständen überschrieben und Sie erhalten unerwartete Ergebnisse zurück.

Die Verwendung der Threadsicheren Compiler Option hat allerdings einen kleinen Preis. Aufgrund des zusätzlichen, automatisch in Ihr Executable eingebundenen Codes wird die Ausführungsgeschwindigkeit Ihres Programms gegenüber dem nicht threadsicheren Modus ein wenig reduziert. Die Entscheidung Threads zu verwenden muss gut durchdacht sein.

Allen Anfängern in PureBasic würde ich generell empfehlen die 'Threadsicheres Executable erstellen' Compiler Option zu verwenden wenn sie Programme kompilieren die mit Threads arbeiten.

Diese 'Delay()' Anweisung blockiert das Programm und hindert es daran Interne Nachrichten, wie zum Beispiel das Neuzeichnen der Benutzeroberfläche oder die Bewegung des Fensters durchzuführen. Wir klicken auf die Schaltfläche und müssen warten bis die Prozeduren abgearbeitet sind, bevor wir wieder mit dem Programm arbeiten können. Nun stellen Sie sich vor es handelt sich um ein Bildverarbeitungsprogramm und die beiden Prozeduren würden jeweils dreißig Minuten für die Fertigstellung brauchen, egal was sie tun. Das ist eine lange Zeit die Sie warten müssen, bevor Sie das Fenster bewegen oder einen Fortschritt auf der Benutzeroberfläche betrachten können. Wir wären auch nicht in der Lage eine Fortschrittsanzeige in unser Programm zu integrieren.

Wenn wir diese beiden Prozeduren gleichzeitig und völlig unabhängig vom Hauptprogramm laufen lassen möchten, dann können wir zwei Threads erstellen die gleichzeitig neben dem Hauptprogramm arbeiten. Das könnte so aussehen:

```
Enumeration
  #FENSTER_HAUPT
  #LISTE_EINS
  #LISTE_ZWEI
  #KNOPF_TEST
EndEnumeration

;Einfügen von Text in das spezifizierte ListGadget
Procedure TextEinfuegen(Gadget.i)
  For x.i = 1 To 25
    AddGadgetItem(Gadget, -1, Str(x))
    Delay(100)
  Next x
EndProcedure

#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER_HAUPT, 0, 0, 290, 200, "Thread Test", #FLAGS)
  ListViewGadget(#LISTE_EINS, 10, 10, 130, 150)
  ListViewGadget(#LISTE_ZWEI, 150, 10, 130, 150)
  ButtonGadget(#KNOPF_TEST, 95, 170, 100, 20, "Starte Test")
  Repeat
    Ereignis.i = WaitWindowEvent()
    Select Ereignis
      Case #PB_Event_Gadget
        Select EventGadget()
          Case #KNOPF_TEST
            Thread1.i = CreateThread(@TextEinfuegen(), #LISTE_EINS)
            Thread2.i = CreateThread(@TextEinfuegen(), #LISTE_ZWEI)
        EndSelect
```

```
    EndSelect
  Until Ereignis = #PB_Event_CloseWindow
EndIf
End
```

Wenn Sie etwas genauer auf die beiden Beispiele schauen, dann werden Sie feststellen dass der einzige Unterschied im Aufruf der Prozeduren liegt. Jeder neue Thread in PureBasic beginnt sein Leben als Standard Prozedur, die auf die übliche Weise im Quelltext definiert wird und immer genau einen Parameter enthält. Um daraus einen Thread zu erstellen verwenden Sie den 'CreateThread()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Thread → CreateThread), der selbst zwei Parameter benötigt. Der erste Parameter ist die Speicheradresse der Prozedur die Sie als Thread starten möchten. Der Zweite Parameter ist ein Pflichtparameter und kann von jedem Typ sein. Sie müssen diesen Parameter immer an die Prozedur übergeben, auch wenn Sie ihn in der Prozedur nicht benötigen. Ich habe ihn hier verwendet um das zu füllende Gadget zu spezifizieren.

Wenn der 'CreateThread()' beim Erstellen eines Threads auf diese Weise einen Wert größer als Null zurückgibt, dann wurde der Thread erfolgreich erstellt und wird augenblicklich gestartet. Der Rückgabewert (wenn erfolgreich) ist ein Integer Wert (Thread Identifikator) und kann mit den anderen Befehlen aus der Thread Bibliothek (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Thread) verwendet werden.

Lassen Sie uns noch einmal die Regeln durchgehen die wir beim erstellen eines Threads aus einer Prozedur einhalten müssen:

1). Die Thread Prozedur muss einen Parameter haben, und darf nicht mehr als Einen haben.
2). Die Thread Prozedur kann keinen Wert zurückgeben. Wenn sie einen Wert zurückgibt, dann geht dieser Wert verloren.

Wenn Sie das Beispiel starten, dann können Sie beobachten, dass die Werte gleichzeitig und in Echtzeit in die beide Gadgets eingetragen werden. Während die Werte eingetragen werden, können Sie das Fenster über den Bildschirm bewegen und Sie werden feststellen dass es nun ganz normal aktualisiert wird. Die beiden neu erstellten Threads sind zu unabhängigen Teilen des Hauptprogramms geworden, sie laufen parallel zu diesem. Wenn das Hauptprogramm zu irgendeinem Zeitpunkt beendet wird, werden auch automatisch die beiden Threads beendet.

Wenn Sie in einem der Gadgets entstellten Text bemerken, dann müssen Sie sich vergewissern dass die Compiler Option 'Threadsicheres Executable erstellen angewählt ist, wie ich es in der Infobox 'Kompilieren von Threadsicheren Programmen' beschrieben habe.

Warten auf das Ende von Threads

Die Verwendung von Threads kann manchmal auch verzwickt sein, und kann neue Probleme aufwerfen während andere behoben werden. Wenn Sie in Ihrem Hauptprogramm Threads verwenden, dann sind diese immer an die 'Lebenserwartung' des Hauptprogramms gebunden. Wenn das Hauptprogramm endet, dann enden auch alle damit verknüpften Threads.

In PureBasic können Sie dieses Problem mit dem 'WaitThread()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Thread → WaitThread) umgehen. Dieser Befehl vergewissert sich ob ein Thread seine Arbeit abgeschlossen hat, bevor er mit dem Hauptprogramm fortfährt. Hier ein Beispiel:

```
Procedure SchreibeZahlen(NichtVerwendet.l)
  For x.i = 1 To 20
    PrintN(Str(x))
    Delay(75)
  Next x
EndProcedure

If OpenConsole()
  ThreadID.i = CreateThread(@SchreibeZahlen(), 0)
  WaitThread(ThreadID)
  Print("Drücken Sie Enter zum beenden.") : Input()
  CloseConsole()
EndIf
End
```

Nach der Zeile in der ich den Thread erstellt habe, habe ich einen 'WaitThread()' Befehl verwendet. Dieser benötigt einen Parameter, es handelt sich um den Thread Identifikator des zuvor erstellten Thread. Dieser Befehl hält die normale Programmausführung an, bis der Thread der im Thread Identifikator übergeben

wurde beendet ist, oder in anderen Worten, bis die Prozedur die zum erstellen des Thread verwendet wurde ihre Arbeit beendet hat. Sie können dem 'WaitThread()' Befehl bei Bedarf einen optionalen zweiten Parameter übergeben, er definiert einen Timeout Wert in Millisekunden. Wenn diese Zeit abgelaufen ist wird die Programmausführung fortgesetzt, unabhängig vom Status des Thread. Ich habe diesen optionalen zweiten Parameter in meinem Beispiel nicht verwendet, weil ich sicherstellen wollte das der Thread beendet ist bevor ich Ihnen die Möglichkeit gebe das Programm zu beenden. Wenn Sie die 'WaitThread' Zeile auskommentieren, werden Sie feststellen, dass Sie das Programm jederzeit mit der Enter Taste beenden können und damit den Thread während der Ausführung 'abschneiden'. Mit dem 'WaitThread()' Befehl, an welcher Stelle auch immer, müssen Sie warten bis der Thread seine Arbeit beendet hat bevor ein Druck auf die Enter Taste ausgewertet wird.

Sichere Verwendung von gemeinsam genutzten Daten

Wenn sie viele Threads in einem Programm verwenden, müssen Sie deren Aktionen sehr vorsichtig programmieren, besonders wenn Sie den Threads erlauben auf gemeinsam genutzte Date zuzugreifen. Wenn Sie viele Threads haben, die gleichzeitig versuchen Lese- und Schreiboperationen in der gleichen Datei oder im gleichen Speicherbereich auszuführen, endet das in der Regel mit zerstörten Daten und einer Menge Tränen. Um das zu vermeiden stellt PureBasic einen einfachen Mechanismus zur Verfügung, der es nur einem Thread gleichzeitig erlaubt Zugriff auf diese gemeinsamen Daten zu nehmen. Bei diesem Mechanismus handelt es sich um einen 'Mutex' (gegenseitiger Ausschluss).

Ein Mutex ist ein Datenobjekt das als Sperre verwendet werden kann um einen Thread an der Verwendung von Daten zu hindern, die bereits von einem anderen Thread verwendet werden. Wenn Sie einen Mutex zum Schutz gemeinsam genutzter Daten verwenden, dann verhindert dieser den Zugriff eines Threads auf die Daten, bis dieser Thread den Mutex gesperrt hat. Nachdem ein Thread dieses Mutex Objekt gesperrt hat, hat dieser freien Zugriff auf die Daten. Wenn ein anderer Thread zur gleichen Zeit versucht diesen Mutex zu sperren, dann wird der zweite Thread angehalten bis der Mutex durch den ersten Thread wieder freigegeben wird, dieser also den Zugriff auf die gemeinsam genutzten Daten beendet. Hier ein Beispiel in dem ich einen Mutex verwende um den schreibenden Zugriff auf die Konsole zu kontrollieren:

```
Global KonsolenZugriff.i = CreateMutex()

Procedure SchreibeZahlen(ThreadNummer.l)
  LockMutex(KonsolenZugriff)
  ConsoleLocate(ThreadNummer * 20, 0)
  ConsoleColor((ThreadNummer + 1) * 3, 0)
  Print("Thread " + Str(ThreadNummer) + " gesichert.")
  For x.i = 1 To 20
    ConsoleLocate(ThreadNummer * 20, x + 1)
    Print(Str(x))
    Delay(75)
  Next x
  UnlockMutex(KonsolenZugriff)
EndProcedure

If OpenConsole()
  EnableGraphicalConsole(#True)
  Thread0.i = CreateThread(@SchreibeZahlen(), 0)
  Thread1.i = CreateThread(@SchreibeZahlen(), 1)
  Thread2.i = CreateThread(@SchreibeZahlen(), 2)
  WaitThread(Thread0)
  WaitThread(Thread1)
  WaitThread(Thread2)
  ConsoleLocate(0, 23)
  Print("Drücken Sie Enter zum beenden.") : Input()
  CloseConsole()
EndIf
End
```

Wenn Sie dieses Programm starten dann werden von jedem Thread Zahlen auf die Konsole geschrieben, allerdings habe ich einen 'Mutex' in der 'SchreibeZahlen()' Prozedur verwendet. Nur ein Thread kann den Mutex zur gleichen Zeit belegen und Zahlen auf die Konsole schreiben.

Die erste Zeile im Code erstellt ein neues Mutex Objekt mit dem 'CreateMutex()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Thread → CreateMutex) erzeugt. Dieser Befehl benötigt keinen Parameter, gibt aber beim Aufruf den Identifikator des neu erstellten Mutex Objektes zurück. Der

zurückgegebene Wert ist ein Integer Wert, weshalb ich ihn in einer Integer Variable mit dem Namen 'KonsolenZugriff' abspeichere.

Nachdem das Mutex Objekt erstellt wurde muss ich meine Prozeduren so gestalten, dass sie den Status gesperrt oder nicht gesperrt einhalten. Um dies zu erreichen muss ich allen Prozeduren die auf die gemeinsam genutzten Daten zugreifen Sperr- und Entsperr Befehle hinzufügen. In meinem Beispiel habe ich nur eine Thread Prozedur, weshalb ich nur hier diese Funktionalität einbauen muss. Wenn ich noch mehr Thread Prozeduren hinzufüge die auf die Konsole schreiben, dann muss ich diese ebenfalls so bearbeiten, dass Sie den Mutex Status beachten.

Um einem Thread die Möglichkeit zu geben einen Mutex zu sperren verwenden wir den 'LockMutex()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Thread → LockMutex). Dieser Befehl benötigt einen Parameter, es handelt sich hierbei um den Identifikator des Mutex Objektes das wir sperren möchten. Wenn der Mutex zum Zeitpunkt des Befehlsaufrufes frei ist, dann sperrt der 'LockMutex()' Befehl den Mutex und der Thread fährt normal mit seiner Arbeit fort. Wenn der Mutex aber durch einen anderen Thread gesperrt ist, dann hält der 'LockMutex()' Befehl den aufrufenden Thread so lange an, bis dieser die Möglichkeit hat den Mutex selbst zu sperren, was der Fall ist wenn der andere Thread den Mutex wieder freigegeben hat.

Sie können diesen Vorgang in meinem Mutex Beispiel beobachten, ich sperre den Mutex bevor ich auf die Konsole schreibe und gebe ihn wieder frei wenn ich mit meiner Arbeit fertig bin. Dadurch ermögliche ich es anderen Threads den Mutex erneut zu sperren und auf die Konsole zu schreiben. Das Entsperren des Mutex erfolgt mit dem 'UnlockMutex()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Thread → UnlockMutex), dem wir den Identifikator des zu entsperrenden Mutex als Parameter übergeben.

Fortführen Ihres Threads bei gesperrtem Mutex

Manchmal ist es sinnvoll einen Thread nicht zu pausieren wenn er auf ein gesperrtes Mutex Objekt trifft. Sie wollen Ihren Thread in dieser Zeit vielleicht etwas anderes erledigen lassen oder einfach nur eine Rückmeldung über den Status des Mutex geben lassen. Wenn das der Fall ist können Sie den 'TryLockMutex()' Befehl (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Thread → TryLockMutex) verwenden. Der 'TryLockMutex()' Befehl macht was er sagt, er versucht den im ersten Parameter übergebenen Mutex zu sperren. Wenn dies fehlschlägt, weil ein anderer Thread bereits diesen Mutex gesperrt hat, dann gibt dieser Befehl den Wert '0' (Null) zurück. Wenn der Sperrvorgang erfolgreich war, gibt dieser Befehl einen nicht Null Wert zurück (ein numerischer Wert der nicht den Wert Null hat).

Hier ein Beispiel das drei Threads verwendet. Alle versuchen das Mutex Objekt mit dem 'TryLockMutex()' Befehl zu sperren. Wenn dies einem Thread nicht gelingt wartet er nicht bis der Mutex freigegeben wird, sondern er zeigt die Wartezeit auf den Mutex in einem StringGadget an.

```
Enumeration
  #FENSTER_HAUPT
  #TEXT_EINS
  #TEXT_ZWEI
  #TEXT_DREI
  #KNOPF_START
EndEnumeration

Global Sperren.i = CreateMutex()

Procedure Aktualisieren(Gadget.i)
  StartZeit.i = ElapsedMilliseconds()
  Repeat
    If TryLockMutex(Sperren)
      For x.i = 1 To 20
        SetGadgetText(Gadget, Str(Gadget) + " hat den Mutex gesperrt. " + Str(x))
        Delay(250)
      Next x
      UnlockMutex(Sperren)
      Break
    Else
      Zeit.s = "(" + Str((ElapsedMilliseconds() - StartZeit) / 1000) + " Sek.)"
      SetGadgetText(Gadget, Str(Gadget) + " wartet auf den Mutex. " + Zeit)
      Delay(1)
    EndIf
  ForEver
  SetGadgetText(Gadget, Str(Gadget) + " ist fertig.")
EndProcedure
```

```
#FLAGS = #PB_Window_SystemMenu | #PB_Window_ScreenCentered
If OpenWindow(#FENSTER_HAUPT, 0, 0, 290, 130, "'TryLockMutex()' Test", #FLAGS)
  StringGadget(#TEXT_EINS, 10, 10, 270, 20, "")
  StringGadget(#TEXT_ZWEI, 10, 40, 270, 20, "")
  StringGadget(#TEXT_DREI, 10, 70, 270, 20, "")
  ButtonGadget(#KNOPF_START, 95, 100, 100, 20, "Start")
  Repeat
    Ereignis.i = WaitWindowEvent()
    Select Ereignis
      Case #PB_Event_Gadget
        Select EventGadget()
          Case #KNOPF_START
            Thread1.i = CreateThread(@Aktualisieren(), #TEXT_EINS)
            Thread2.i = CreateThread(@Aktualisieren(), #TEXT_ZWEI)
            Thread2.i = CreateThread(@Aktualisieren(), #TEXT_DREI)
        EndSelect
    EndSelect
  Until Ereignis = #PB_Event_CloseWindow
EndIf
End
```

Hier können Sie sehen, dass ich innerhalb der Thread Prozedur eine 'Repeat' Schleife verwendet habe um den Thread am laufen zu halten, während er den Mutex nicht sperren kann. Wenn der Mutex gesperrt werden konnte, dann zählt der Thread bis Zwanzig und aktualisiert das String Gadget. Nachdem der Zählvorgang abgeschlossen ist wird der Mutex wieder freigegeben und das 'Break' Schlüsselwort wird aufgerufen um die Schleife zu verlassen, und damit den Thread zu beenden. Eine 'If' Anweisung prüft den Rückgabewert des 'TryLockMutex()' Befehls und entscheidet dann welche Aktion ausgeführt wird.

Zusammenfassung

Threads sind ein weiteres nützliches Werkzeug in der Werkzeugkiste des Programmierers, und wie jedes Werkzeug muss es richtig eingesetzt werden. Wenn Sie ein Programm erstellen das Threads verwendet, dann würde ich Ihnen empfehlen immer die 'Threadsicheres Executable erstellen' Compiler Option zu aktivieren und so wenige Threads wie möglich zu verwenden.

Ich empfehle Ihnen ebenfalls den kompletten 'Thread' Bereich in der PureBasic Hilfe durchzulesen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Thread), da hier noch viele weitere hilfreiche Thread Befehle beschrieben werden, die ich hier nicht alle anführen konnte. Bevor Sie mit Threads arbeiten, sollten Sie vollständig mit ihrem Nutzen und ihren Einschränkungen vertraut sein.

Dynamic Link Libraries

Dynamic Link Libraries, auch Dynamically Linked Libraries genannt oder üblicherweise abgekürzt einfach 'DLL' sind eine Microsoft Implementierung von gemeinsam benutzbaren Code Bibliotheken für das Microsoft Windows Betriebssystem, und sind als solche nur auf einem Microsoft Betriebssystem verwendbar. DLLs ermöglichen es dem Programmierer, Code in eine gemeinsam verwendbare Bibliothek zu kompilieren und ermöglichen es anderen Programmierern die darin enthaltene Funktionalität in ihren Programmen zu nutzen. Programme können sich zur Laufzeit dynamisch mit der DLL verbinden und die darin enthaltenen Befehle nutzen. Wenn die Funktion in der DLL abgearbeitet ist kann das Programm die Bibliothek wieder entladen und den ihr zugewiesenen Speicher wieder freigeben.

Das DLL Konzept war einst dazu gedacht die Programmgröße und den Speicherverbrauch zu reduzieren, indem man die Bibliotheken auf der Festplatte ablegt und nur bei Bedarf nachlädt. Die Realität sieht aber oft anders aus, und besonders beim Microsoft Windows Betriebssystem ist diese 'Krankheit' weit verbreitet. Hier finden sich oft die gleichen DLLs in verschiedenen Versionen, da diese von verschiedenen Programmen aus Kompatibilitätsgründen benötigt werden. Das heißt, dass manchmal viele Versionen der gleichen DLL (allerdings in verschiedenen Versionen) benötigt werden. Dieser Umstand hat Windows den Spitznamen 'DLL Hell' gegeben, um darauf hinzuweisen welche Probleme daraus entstehen können.

Schieben wir einmal die Betriebssystem Probleme zur Seite, DLLs können sehr nützlich sein wenn Sie ein Programm in Modularer Art erstellen möchten. Wenn Sie einen bestimmten Code als nützlich für zukünftige Programme erachten, dann können Sie diesen in eine DLL kompilieren und er steht somit für jedes Programm zur Verfügung.

PureBasic macht die Verwendung von DLLs durch die eingebaute 'Library' Bibliothek (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Library) sehr einfach. Sie können auf einfache Weise DLLs laden und

verwenden, während der PureBasic Compiler eine Option hat, mit der Sie eigenen Programmcode zu einer DLL kompilieren können. Das macht PureBasic zur perfekten Wahl bei der Verwendung von DLLs.

Erstellen einer DLL

Wenn Sie Ihre eigene DLL erstellen, müssen Sie natürlich ein paar Dinge beachten, aber keine Angst, das ganze ist recht schmerzlos. Zuerst einmal müssen Sie daran denken dass nahezu der komplette Code innerhalb von Prozeduren untergebracht werden muss. Das liegt daran, dass wenn ein Programm Ihre DLL lädt, dann führt es den darin enthaltenen Code über Aufrufe der kompilierten Prozedurnamen aus. Das öffnen einer Bibliothek startet nicht automatisch die DLL wie ein Programm, stattdessen müssen Sie einzelne Prozeduren in ihr direkt aufrufen. Globale Variablen- und Struktur Definitionen finden nicht innerhalb von Prozeduren in der DLL statt, das ist aber auch das einzige was nicht innerhalb einer Prozedur liegt.

Wenn Sie Code für eine DLL schreiben müssen Sie darauf achten, dass es vier speziell reservierte Prozedurnamen gibt, die automatisch von Windows aufgerufen werden nachdem die DLL geladen ist. Diese vier Prozeduren werden wie jede andere DLL Prozedur definiert, aber wenn sie richtig benannt sind ruft Windows jede einzelne auf wenn eine bestimmte Bedingung in der umliegenden DLL auftritt. Ich persönlich definiere diese vier Funktionen in allen DLLs die ich erstelle, auch wenn ich sie nicht verwende. Das hat folgenden Grund, wenn ich irgendwann einmal diese bestimmte DLL erweitere, dann sind die Funktionen im Bedarfsfall vorhanden und bereit zur Verwendung. Hier ist eine Liste der vier reservierten Prozedurnamen und welche Bedingungen ihre automatische Ausführung auslösen:

'AttachProcess(Instanz.i)'
Diese Prozedur wird einmal aufgerufen wenn ein Programm die DLL lädt. Diese Prozedur ist die einzige Stelle in der DLL, an der Sie Arrays und Listen definieren können. Ich verwende diese Prozedur auch zur Definition von Variablen und Strukturen. Damit halte ich alles an einer Stelle zusammen und vermeide es Definitionen außerhalb von Prozeduren vorzunehmen. Die einzige Einschränkung besteht darin, dass in die Prozedur keine DirectX Initialisierungs Routinen geschrieben werden dürfen.

'DetachProcess(Instanz.i)'
Diese Prozedur wird aufgerufen wenn das Programm die DLL schließt. Jeglicher Code zum aufräumen sollte sich in dieser Prozedur befinden, wie zum Beispiel die Freigabe von Ressourcen die von der DLL verwendet werden.

'AttachThread(Instanz.i)'
Diese Prozedur wird jedes Mal aufgerufen wenn ein Thread im Programm erstellt wird der diese DLL lädt.

'DetachThread(Instanz.i)'
Diese Prozedur wird jedes Mal aufgerufen wenn ein Thread im Programm beendet wird der diese DLL geladen hat.

Wie Sie in der Auflistung sehen können, sind diese Prozeduren nicht schwer zu verstehen und sie haben einfache Namen. Wenn Sie diese automatisch aufgerufenen Prozeduren in Ihrer DLL verwenden möchten, dann müssen Sie nur die Prozedurnamen wie oben beschrieben definieren. Der 'Instanz' Parameter den alle Prozeduren verwenden ist ein Instanz Handle (manchmal auch Modul Handle genannt). Dieses kann bei der etwas mehr fortgeschrittenen Programmierung sehr praktisch sein. Dieser Parameter muss immer definiert werden wenn Sie diese Prozeduren verwenden, auch wenn Sie nicht beabsichtigen den Parameter in der Prozedur zu verarbeiten.

Nun wissen Sie wie eine DLL programmiert werden sollte (alles muss sich innerhalb von Prozeduren befinden) und Sie kennen nun die Namen der speziellen reservierten Prozeduren. Lassen Sie mich Ihnen ein Beispiel für eine einfache DLL zeigen:

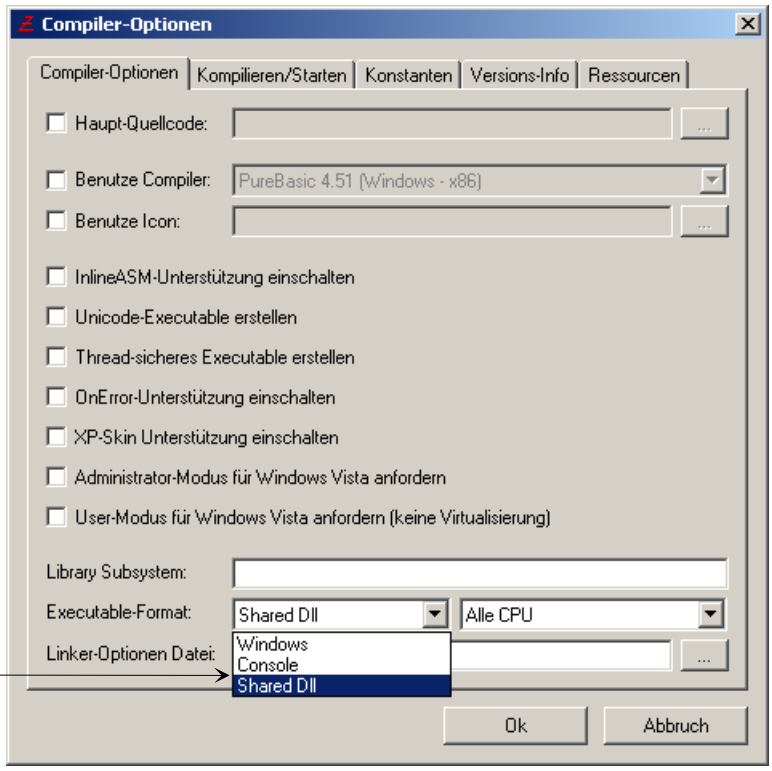
```
ProcedureDLL AttachProcess(Instanz.i)
  Global Nachricht.s = "Das ist meine DLL."
EndProcedure

ProcedureDLL ZeigeAlarm()
  MessageRequester("Alarm", Nachricht)
EndProcedure
```

Hier habe ich um die Sache kurz zu halten nicht alle Speziellen reservierten Prozeduren verwendet. Die einzige die ich in dem Beispiel verwendet habe ist die 'AttachProcess()' Prozedur, in der ich eine Globale Stringvariable mit dem Namen 'Nachricht' definiert habe. Die zweite Prozedur in dem Beispiel hat den Namen 'ZeigeAlarm()'. Diese Prozedur werde ich aufrufen wenn ich die DLL mit einem anderen Programm

öffne. Wie Sie sehen können verwendet der Message Requester in der 'ZeigeAlarm()' Prozedur den String, den ich in der 'AttachProcess()' Prozedur definiert habe.

Sie werden ebenfalls bemerkt haben, dass ich ein anderes Schlüsselwort zum definieren der Prozeduren verwendet habe, an Stelle von 'Procedure' habe ich 'ProcedureDLL' verwendet. Das ist essentiell wenn Sie spezielle reservierte Prozeduren, oder für alle Programme verfügbare Prozeduren erstellen möchten. Sie können das Standard 'Procedure' Schlüsselwort verwenden um Prozeduren zu definieren, allerdings sind diese dann von anderen Programmen die Ihre DLL laden nicht erreichbar. Normale Prozeduren werden in DLLs hauptsächlich für interne Berechnungen, usw. verwendet.

Nachdem wir unsere DLL in obigem Stil programmiert haben müssen wir sie testen. Also lassen Sie uns zuerst das Beispiel in eine DLL kompilieren. Da Dynamic Link Libraries ein wenig von normalen Programmen abweichen, müssen wir dem Compiler mitteilen dass wir eine DLL an Stelle eines normalen Programms erstellen möchten. Wenn Sie den Compiler auf der Kommandozeile verwenden, dann können Sie dies mit der Compiler Option '/DLL' erledigen. Innerhalb der IDE können Sie den 'Compiler-Optionen' Dialog verwenden (Menü: Compiler → Compiler-Optionen... → Compiler-Optionen → Executable-Format → Shared DLL). Abb. 52 zeigt wo Sie 'Shared DLL' im Compiler Optionen Dialog auswählen müssen. Nachdem Sie dies erledigt haben können Sie den Befehl 'Executable erstellen...' im 'Compiler' Menü auswählen um die eigentliche DLL zu erzeugen. Hier haben Sie dann die Möglichkeit einen Dateinamen und den Ablageort für Ihre DLL zu definieren.



Der 'Compiler-Optionen' Dialog, wie er unter Microsoft Windows erscheint Abb. 52

Nachdem Sie Ihre DLL erstellt haben werden Sie feststellen, dass der Compiler drei Dateien erstellt hat. Wenn wir unsere DLL zum Beispiel 'Demo.dll' genannt hätten, dann würden wir folgende drei Dateien vorfinden:

'Demo.dll' 'Demo.exp' 'Demo.lib'

Die erste Datei ist natürlich die kompilierte DLL, die Zweite ist eine Export Datei und die Dritte ist eine statische 'Library' Datei. PureBasic benötigt nur die DLL um die darin befindlichen Befehle zu nutzen, die anderen Dateien können aber für Programmierer anderer Sprachen nützlich sein. Wenn Sie eine DLL veröffentlichen wollen, empfehle ich Ihnen alle drei Dateien weiterzugeben. Wenn Sie die DLL allerdings nur mit PureBasic verwenden, dann benötigen Sie nur die DLL Datei, die anderen beiden Dateien können Sie ignorieren.

Verwenden einer DLL in Ihrem Programm

Die Verwendung einer DLL in Ihrem Programm ist einfach, Sie öffnen einfach die Bibliothek und rufen die Prozedur auf die Sie starten möchten. Dieser Vorgang wird in PureBasic mit den Befehlen aus der 'Library' Bibliothek (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Library) ausgelöst. Hier ein Beispiel wie wir die 'ZeigeAlarm()' Prozedur aus der 'Demo.dll' aufrufen würden. Stellen Sie sicher, dass sich die 'Demo.dll' im gleichen Verzeichnis wie Ihr Quelltext mit diesem Code befindet:

```
#LIBRARY_DEMO = 0
If OpenLibrary(#LIBRARY_DEMO, "Demo.dll")
  CallFunction(#LIBRARY_DEMO, "ZeigeAlarm")
  CloseLibrary(#LIBRARY_DEMO)
EndIf
```

Zuerst öffnen Sie die Bibliothek mit dem 'OpenLibrary()' Befehl. Dieser benötigt zwei Parameter. Der Erste ist die PB Nummer, die Sie mit der neu geöffneten Bibliothek verknüpfen möchten und der Zweite ist der Name der DLL die Sie öffnen möchten. Um eine Funktion in der geöffneten DLL aufzurufen verwenden Sie den 'CallFunction()' Befehl der zwei Pflichtparameter benötigt. Der erste Parameter ist die PB Nummer der Bibliothek aus der Sie eine Funktion ausführen möchten und der zweite Parameter ist der Name der Prozedur die Sie ausführen möchten. Beachten Sie, dass Sie bei der Eingabe des Prozedurnamen keine Klammern am Ende einfügen. Nach diesen zwei Pflichtparametern kann der 'CallFunction()' Befehl noch bis zu zwanzig optionale Parameter haben, die an die Prozedur in der DLL übergeben werden. Das ist nützlich wenn die Prozedur in der DLL Parameter benötigt. Der letzte Befehl in meinem Beispiel ist der 'CloseLibrary()' Befehl, der die mit der übergebenen PB Nummer verknüpfte DLL wieder schließt.

Die Erstellung und Verwendung von DLLs kann etwas komplizierter werden als hier gezeigt, aber die hier verwendeten Beispiele zeigen den Umriss des ganzen Prozesses. Alle DLLs bestehen aus Prozeduren, und um diese zu verwenden müssen Sie die Bibliothek öffnen und die Prozeduren aufrufen (egal welche Methode Sie verwenden), das ist einfach.

Wenn Sie eine DLL auf diese Weise verwenden, werden Sie vielleicht verführt die DLL zu öffnen, eine Prozedur zu verwenden und die Bibliothek unmittelbar wieder zu schließen. Das ist in Ordnung wenn Sie gelegentlich ein oder zwei Prozeduren aufrufen, aber Sie bekommen Geschwindigkeitsprobleme wenn Ihr Programm regen Gebrauch von der DLL macht. Ein besserer Weg ist die Bibliothek beim Programmstart zu öffnen und dann die Befehle der DLL in Ihrem Programm zu nutzen. Wenn Sie dann ihr Programm beenden, schließen Sie vorher noch die DLL Bibliothek. Das stellt sicher, dass die DLL nur einmal geöffnet und auch nur einmal geschlossen wird, außerdem fallen die Geschwindigkeitseinbußen bei den Dateioperationen weg.

Rückgabe von Werten aus DLL Prozeduren

Wenn Sie Prozeduren in DLLs verwenden, kann es manchmal nötig sein einen Wert von einer Prozedur zurückzugeben. Das ist mit dem 'ProcedureReturn' Schlüsselwort möglich, wie in einer ganz normalen Prozedur. Hier ein einfaches Beispiel einer DLL Prozedur die einen Wert zurückgibt:

```
ProcedureDLL.l MultipliziereWerte(x.l, y.l)
  ProcedureReturn x * y
EndProcedure
```

Wie Sie sehen ist die Sache sehr geradlinig und abgesehen vom 'ProcedureDLL' Schlüsselwort sieht das Ganze wie eine normale Prozedur aus. Kompilieren Sie den Code in eine DLL und rufen ihn dann mit diesem Code auf:

```
#LIBRARY = 0
If OpenLibrary(#LIBRARY, "Demo.dll")
  Ergebnis.l = CallFunction(#LIBRARY, "MultipliziereWerte", 22, 33)
  Debug Ergebnis
  CloseLibrary(#LIBRARY)
EndIf
```

Hier verwende ich wieder den 'CallFunction()' Befehl, aber diesmal verwende ich zwei der optionalen Parameter um Werte an die 'MultipliziereWerte()' Prozedur in der DLL zu übergeben. Ich übergebe als Parameter die Werte '22' und '33'. Wenn diese an die 'MultipliziereWerte()' Prozedur in der DLL übergeben werden, dann verwendet diese sie als Parameter und bearbeitet sie dementsprechend. In diesem Fall werden die beiden Werte multipliziert und das Ergebnis zurückgegeben. Der zurückgegebene Wert wird dann an den 'CallFunction()' Befehl übergeben und dieser leitet ihn weiter in unser Programm. Ich habe eine Variable vom Typ Long mit dem Namen 'Ergebnis' verwendet um den zurückgegebenen Wert aus der DLL zu speichern. Diesen gebe ich dann im Debug Ausgabefenster aus.

Während diese Möglichkeit Werte zurückzugeben für die meisten eingebauten Integer Typen gut geeignet ist (siehe 'Einschränkungen beim Arbeiten mit DLLs' weiter unten), funktioniert das Ganze mit Strings ein wenig anders. Zu Beginn: Jeder String den Sie aus einer DLL zurückgeben möchten muss Global definiert sein. Dadurch wird der String erhalten wenn die DLL Prozedur beendet wird. Weiterhin gilt, wenn ein String aus einer DLL Prozedur entgegengenommen wird, dann ist der Rückgabewert ein Zeiger auf diesen String, auch wenn der Rückgabetyp der Prozedur als String definiert wurde.

Hier ein Beispiel zur Demonstration:

```
ProcedureDLL AttachProcess(Instanz.i)
  Global MeinString.s = "Ich möchte in das Hauptprogramm."
EndProcedure

ProcedureDLL.s HoleString()
  ProcedureReturn MeinString
EndProcedure
```

In diesem Beispiel habe ich die 'HoleString()' Prozedur definiert um einen String zurückzugeben, und ich habe mich vergewissert dass die 'MeinString' Variable Global definiert ist. Kompilieren Sie das Beispiel zu einer DLL und rufen Sie es mit diesem Code auf:

```
#LIBRARY = 0
If OpenLibrary(#LIBRARY, "Demo.dll")
  *Ergebnis = CallFunction(#LIBRARY, "HoleString")
  Debug PeekS(*Ergebnis)
  CloseLibrary(#LIBRARY)
EndIf
```

Sie können sehen, dass der 'CallFunction()' Befehl einen Zeiger zurückgibt, weshalb wir eine Zeigervariable verwenden müssen um den Wert zu speichern. Deshalb habe ich einen Zeiger mit dem Namen '*Ergebnis' erstellt um die zurückgegebene Adresse zu speichern. Mit dem 'PeekS()' Befehl hole ich dann den String von dieser Adresse ab.

Einschränkungen beim Arbeiten mit DLLs

Die einzigen Einschränkungen beim arbeiten mit DLLs gibt es beim übergeben und zurückgeben von Werten zur und von der DLL. Bytes, Words und Longs können relativ problemlos verwendet werden, da alle DLL Befehle sie unterstützen, aber die anderen Typen können ein wenig Kopfzerbrechen bereiten. Sie haben bereits die eigentümliche Art der String Rückgabe kennengelernt, der als Speicheradresse an Stelle eines Strings seinen Weg in das Hauptprogramm gefunden hat, Float Werte werden bei der Rückgabe aus der DLL scheinbar völlig verstümmelt. Quads und Doubles werden von den Standard 'Library' Befehlen überhaupt nicht unterstützt, können aber mit den PureBasic Prototypen verarbeitet werden. Dieses Thema geht aber über den Bereich des Buches hinaus.

Um einen Float Wert aus einer DLL Prozedur zurückzugeben verwenden Sie am besten auch wieder einen Zeiger und lesen dann im Hauptprogramm den eigentlichen Wert mit dem 'PeekF()' Befehl. Hier ein kleines Beispiel wie Sie erfolgreich einen Float Wert aus einer DLL Prozedur zurückgeben. Zuerst das DLL Beispiel:

```
ProcedureDLL AttachProcess(Instanz.i)
  Global MeinFloat.f = 3.1415927
EndProcedure

ProcedureDLL.l HolePi()
  ProcedureReturn @MeinFloat
EndProcedure
```

Anstatt den eigentlichen Float Wert zurückzugeben ist die DLL Prozedur 'HolePi()' so definiert, dass Sie einen Long Wert zurückgibt. Dadurch kann ich die Speicheradresse der Float Variable übergeben. Nach dem Kompilieren in eine DLL kann ich die Funktion mit folgendem Code aufrufen:

```
#LIBRARY = 0
If OpenLibrary(#LIBRARY, "Demo.dll")
  *Ergebnis = CallFunction(#LIBRARY, "HolePi")
  Debug PeekF(*Ergebnis)
  CloseLibrary(#LIBRARY)
EndIf
```

Nachdem die Speicheradresse zurückgegeben wurde, wird sie dem '*Ergebnis' Zeiger zugewiesen. Anschließend verwende ich den 'PeekF()' Befehl um den Float Wert von der Speicherstelle zu lesen, auf die der Zeiger verweist. Genau genommen kann dieser Trick mit dem Zeiger auch für Quads oder Doubles verwendet werden, so dass man effektiv sagen kann PureBasic unterstützt die Rückgabe dieser Typen.

Zur Zeit gibt es keine Möglichkeit mit den 'Library' Befehlen einen Quad- oder Double Wert als Parameter an eine DLL Prozedur zu übergeben.

Die Verwendung von DLLs ist sehr einfach, wenn Sie bei der Übergabe und Rückgabe von Werten mit dem Kopf bei der Sache sind. Sie machen den Code wiederverwendbar, indem sie ihn in eine sauber verpackte Bibliothek stecken. Es gibt einige Einschränkungen bei der Verwendung von einigen eingebauten Typen, aber diese können mit einigen fortschrittlicheren Techniken umgangen werden, üblicherweise durch die Verwendung eines Zeigers auf die benötigten Daten. Die etwas komplizierteren Ansätze, wie die Prototypen, sind zum momentanen Zeitpunkt noch etwas zu fortgeschritten, aber Sie können jederzeit mehr darüber in der PureBasic Hilfe erfahren (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Prototype) und sich selbst damit vertraut machen. Seien Sie nicht schockiert, wenn Sie diese nicht beim ersten Anlauf verstehen. Weiterhin empfehle ich Ihnen den 'Library' Bereich in der PureBasic Hilfe durchzulesen (Hilfe Datei: Referenz-Handbuch → Allgemeine Libraries → Library), um mehr über die Befehle zu erfahren die mit den DLLs verwendet werden können. Das sollte Ihnen ein gutes Verständnis über die verfügbaren Möglichkeiten vermitteln bevor Sie sich entscheiden Ihre erste richtige DLL zu programmieren.

DLL Aufruf Konventionen

Wenn Sie PureBasic verwenden, dann gibt es zwei Aufruf Konventionen die mit DLLs verwendet werden können. Als Standard geht PureBasic davon aus, dass die DLL die 'stdcall' Aufruf Konvention verwendet und dass alle DLLs die mit PureBasic mittels des 'ProcedureDLL' Schlüsselwortes erstellt werden ebenfalls zur Verwendung der 'stdcall' Aufruf Konvention vorbedacht sind. Diese Konvention wird von PureBasic als Standard verwendet, da dies auch die Standard Aufruf Konvention von Microsoft Windows ist.

Bei Bedarf kann auch die 'cdecl' Aufruf Konvention verwendet werden. Hierzu werden die komplementären 'C' Funktionen der PureBasic 'Library' Befehle verwendet. An Stelle des 'ProcedureDLL' Schlüsselwortes, zum erstellen einer 'stdcall' DLL Prozedur, verwenden Sie zum Beispiel 'ProcedureCDLL' um eine Prozedur nach der 'cdecl' Aufruf Konvention zu erstellen. Das gleiche gilt beim Aufruf von Prozeduren der DLL. An Stelle des Aufrufs einer 'stdcall' Prozedur mit dem 'CallFunction()' Befehl rufen Sie eine 'cdecl' Prozedur mit dem 'CallCFunction()' Befehl auf.

Es existieren noch mehr dieser komplementären 'C' Funktionen in PureBasic. In der Hilfe Datei im 'Library' Bereich finden Sie eine komplette Auflistung der Befehle.

Das Windows Application Programming Interface

Sie werden sich vielleicht fragen, warum ich einen Bereich über Windows Programmierung in ein Buch einfüge das eigentlich eine plattformübergreifende Programmiersprache beschreibt. Ich habe mich dazu entschlossen, weil die Windows Version von PureBasic die besonders nützliche Eigenschaft hat, das Windows Application Programming Interface direkt nutzen zu können. Lassen Sie mich das etwas ausführlicher erklären.

Das Windows Application Programming Interface (üblicherweise abgekürzt als Win32 API) befindet sich in allen neuen Versionen des Microsoft Windows Betriebssystems, es handelt sich hierbei um eine weit verbreitete Programmierschnittstelle, die von allen Computer Programmen genutzt werden kann. Das Win32 API ist im Grunde eine große Befehlsbibliothek, die sich auf viele verschiedene DLLs verteilt und automatisch während der normalen Windows Installation auf die Festplatte gelangt. Das API selbst ist Teil des Betriebssystems und kann von jedem Computer Programm dazu verwendet werden, das meiste auszuführen wozu das Betriebssystem fähig ist, wie z.B. öffnen von Fenstern, manipulieren von Strings, zeichnen von Formen, usw.

Verwenden von Win32 API Befehlen

Um einen Win32 API Befehl zu verwenden kann der Benutzer einfach die entsprechende DLL laden, die den benötigten Befehl enthält und diesen dann bei Bedarf mit den entsprechenden Parametern aufrufen. Das kann in PureBasic einfach mit den 'Library' Befehlen erledigt werden. Da PureBasic für Windows das Win32 API allerdings direkt unterstützt können Sie die API Befehle so einfach wie ganz normale PureBasic Befehle

verwenden. Andere, schwerfälligere Programmiersprachen müssen zum Beispiel solche Sachen bewerkstelligen um einen einfachen Message Requester unter Zuhilfenahme des Win32 API anzuzeigen:

```
#LIBRARY = 0
If OpenLibrary(#LIBRARY, "User32.dll")
  Stil           = 0
  Beschreibung.s = "Test"
  TextString.s   = "Das ist ein Win32 API Test"
  CallFunction(#LIBRARY, "MessageBoxA", #Null, @TextString, @Beschreibung, Stil)
  CloseLibrary(#LIBRARY)
EndIf
```

Sie können sehen, dass Sie zuerst die DLL mit dem benötigten Befehl öffnen müssen. Hier ist es der Befehl 'MessageBoxA' der sich in der 'User32.dll' befindet. Nachdem diese geöffnet wurde, können Sie den benötigten Befehl aufrufen und ihm die benötigten Parameter übergeben. Das ist alles gut und schön aber auch ein wenig kompliziert und langwierig. In PureBasic können Sie den gleichen Befehl folgendermaßen aufrufen:

```
MessageBox_(#Null, "Das ist ein Win32 API Test", "Test", #MB_OK)
```

Haben Sie den Unterstrich ('_') nach dem Befehlsnamen bemerkt? Dieser verdeutlicht, dass es sich hierbei um einen API Befehl handelt und ruft ihn automatisch aus der entsprechenden DLL heraus auf. Ich denke wir sind uns einig, dass diese Vorgehensweise erheblich einfacher und schneller ist. Wir verwenden API Befehle als wären es Standard PureBasic Befehle.

Der Unterstrich kann verwendet werden um automatisch einen Befehl aus dem Win32 API einzufügen. Im Win32 API gibt es zum Beispiel einen Befehl mit dem Namen 'CharUpper', der einen String in Großbuchstaben umwandelt. Durch die Verwendung des Unterstrich vor den Klammern können wir diesen Befehl wie einen normalen PureBasic Befehl verwenden, wie hier:

```
TextString.s = "das ist ein kleingeschriebener text."
CharUpper_(@TextString)
Debug TextString
```

Dieser Befehl benötigt gemäß der Win32 API Dokumentation die Speicheradresse eines String als Parameter, weshalb ich diese übergebe. Wenn Sie dieses Beispiel starten und auf das Debug Ausgabefenster schauen, werden Sie feststellen dass der String in Großbuchstaben umgewandelt wurde.

Die Verwendung des Win32 API auf diese Art entschärft den Ärger mit dem Öffnen der benötigten DLLs und erlaubt es Ihnen API Befehle in Ihrem Programm zu nutzen, wann immer sie diese benötigen. Wenn Sie die Windows Version von PureBasic verwenden können Sie in Ihrem Code jederzeit die normalen Befehle mit den API Befehlen mischen, ganz wie Sie wollen.

Verwenden von Win32 API Konstanten und Strukturen

Um die einfache Verwendung der Win32 API Befehle zu vervollständigen sind in der Windows Version von PureBasic bereits alle verknüpften Konstanten und Strukturen vordefiniert. Das bedeutet, dass wenn Sie irgendwann einmal eine Konstante oder eine Struktur für die Verwendung mit einem Win32 API Befehl benötigen, dann sind diese bereit intern definiert, bereit zur sofortigen Verwendung.

Das können Sie in der PureBasic Version des 'MessageBox' Beispiels sehen, in dem ich die Konstante '#MB_OK' verwendet habe, ohne diese vorher zu definieren. Das funktioniert weil die Win32 API Konstante '#MB_OK' bereits intern definiert wurde. Um eine Win32 API Konstante in PureBasic zu verwenden, müssen Sie nur das Raute Zeichen ('#') vor den Namen der Konstante setzen und diese sollte dann so verwendbar sein, als wäre sie direkt in Ihrem Quelltext definiert worden.

Win32 API Konstante: 'MB_OK'
PureBasic Version: '#MB_OK' (muss nicht definiert werden, da dies bereit intern geschehen ist)

Das gleiche gilt für die Win32 API Strukturen, immer wenn Sie eine benötigen, dann verwenden Sie sie einfach. Sie müssen sie nicht definieren. Schauen Sie auf dieses Beispiel:

```
DesktopInfo.RECT
DesktopHandle.i = GetDesktopWindow_()
GetClientRect_(DesktopHandle, @DesktopInfo)
```

```
DesktopGroesse.s = "Ihr Desktop hat folgende Größe: "
DesktopGroesse + Str(DesktopInfo\right) + " x "
DesktopGroesse + Str(DesktopInfo\bottom)
Debug DesktopGroesse
```

Hier verwende ich die Win32 API Struktur mit dem Namen 'RECT', ohne sie vorher definiert zu haben und danach verwende ich die Win32 API Befehle 'GetDesktopWindow' und 'GetClientRect', als wären es eingebaute Befehle (beachten Sie die Unterstriche). Wie Sie an der 'RECT' Struktur erkennen können, wurden alle Win32 API Strukturen unter Verwendung von Großbuchstaben im Namen definiert. Das ist die Standard Form wenn Sie unter Verwendung des Win32 API programmieren, und meine bevorzugte Schreibweise für Strukturen, wie in Kapitel 8 (Meine Code Gestaltung) beschrieben.

Diese Namens Konventionen entsprechen exakt der Art, in der Strukturen in der Win32 API Referenz definiert wurden. Dadurch vermeide ich Verwirrungen wenn die Zeit kommt diese zu verwenden.

Wenn zum Beispiel die Win32 API Dokumentation sagt, dass ein bestimmter Befehl eine Variable vom Typ 'RECT' benötigt, dann können Sie 'RECT' direkt in Ihrem PureBasic Code als Struktur verwenden.

Betrachten Sie diesen Beispiel Text aus der Win32 API Dokumentation, die den 'BeginPaint' API Befehl beschreibt.

```
Die BeginPaint Funktion präpariert das spezifizierte Fenster zum zeichnen und
füllt eine PAINTSTRUCT Struktur mit Informationen zum zeichnen.

HDC BeginPaint(
       HWND hwnd,          // Handle zum Fenster
       PAINTSTRUCT lpPaint // Zeiger zur Struktur mit Informationen zum zeichnen);

Parameter

  hwnd
    Identifikator des Fensters das neu gezeichnet werden soll.

  lpPaint
    Zeiger zur PAINTSTRUCT Struktur die Informationen zum zeichnen empfängt.

Rückgabewerte
  Wenn die Funktion erfolgreich war, dann ist der Rückgabewert das Handle zum
  Anzeigegeräte Kontext (HDC) des spezifizierten Fensters. Wenn die Funktion
  fehlschlägt ist der Rückgabewert NULL, was anzeigt dass kein Anzeigegeräte
  Kontext verfügbar ist.
```

Sie können sehen, dass dieser Befehl als zweiten Parameter einen Zeiger (oder eine Speicheradresse) zu einer strukturierten Variable benötigt, und diese Variable muss mit der 'PAINTSTRUCT' Struktur erstellt werden. Um eine solche Variable in PureBasic zu erzeugen gehen wir so vor:

```
ZeichenVariable.PAINTSTRUCT
```

Und das war's. Die 'PAINTSTRUCT' Struktur wurde bereits intern definiert, weshalb wir sie hier ohne eigene Definition verwenden können. Ein voll funktionsfähiges Beispiel in Ihrem Hauptprogramm könnte folgendermaßen aussehen:

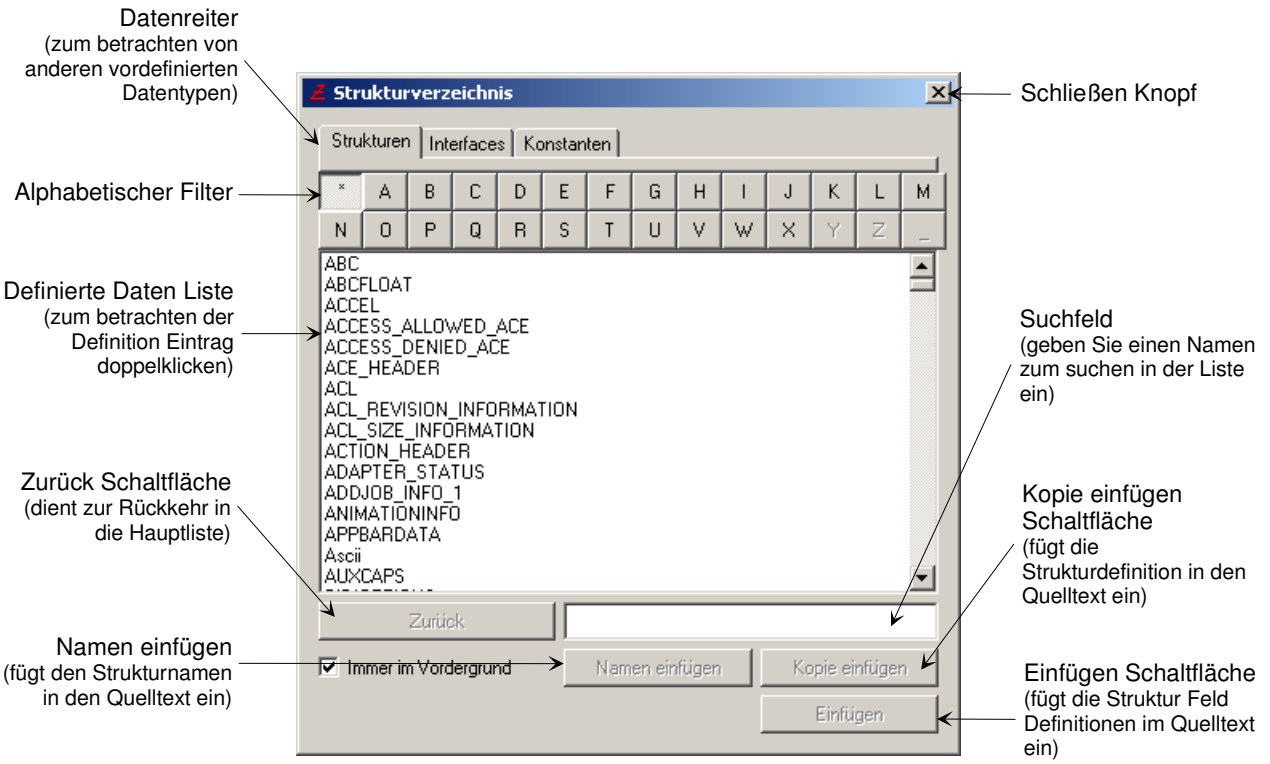
```
...
ZeichenVariable.PAINTSTRUCT
FensterHandle.i = WindowID(#MEIN_FENSTER)
HDC.i = BeginPaint_(FensterHandle, @ZeichenVariable)
...
```

Betrachten von intern definierten Konstanten und Strukturen

Nun wissen Sie, dass eine Menge praktischer Daten bereits vordefiniert sind, aber Sie wissen nicht zwangsläufig was exakt alles definiert wurde. Um hier Klarheit zu schaffen und Ihnen zu zeigen was exakt definiert wurde, stellt die PureBasic IDE ein praktisches Werkzeug zur Verfügung, mit dem Sie all diese Daten betrachten können. Dieses Werkzeug nennt sich 'Strukturverzeichnis' und befindet sich im 'Werkzeuge' Menü (Menü: Werkzeuge → Strukturverzeichnis). Abb. 53 zeigt das Strukturverzeichnis wie es auf dem Windows Betriebssystem aussieht und benennt viele der Funktionen, die es zu einem nützlichen Werkzeug machen.

Nachdem das Strukturverzeichnis geöffnet wurde, werden Sie mit einer Liste aller intern definierten Strukturen begrüßt. Diese Liste kann durchsucht und alphabetisch sortiert werden wenn Sie das entsprechende Steuerelement betätigen. Auch wenn diese Liste zu Beginn nur Strukturen anzeigt, können Sie sich durch die Anwahl des entsprechenden Reiters auch Interfaces und Konstanten anzeigen lassen. Die Interfaces haben wir in diesem Buch nicht besprochen, aber die Konstanten Liste ist sehr hilfreich für Sie, da hier alle internen Konstanten mit ihren verknüpften Werten angezeigt werden. Alle eingebauten PureBasic Konstanten sind hier ebenfalls einsehbar.

Um die eigentliche Definition einer intern definierten Struktur zu betrachten, führen Sie einfach einen Doppelklick auf dem Strukturnamen in der Liste aus. Wenn Sie das getan haben werden die Schaltflächen 'Kopie Einfügen' und 'Einfügen' aktiv. Die 'Kopie Einfügen' Schaltfläche fügt eine Kopie der Strukturdefinition in Ihren PureBasic Quelltext ein, während die 'Einfügen' Schaltfläche Code zur eigentlichen Verwendung der Struktur in Ihren Quelltext einfügt. Öffnen Sie ein neues Quelltext Fenster spielen Sie ein wenig damit, um herauszufinden wie es funktioniert.



Der 'Strukturverzeichnis' Dialog, wie er unter Microsoft Windows erscheint Abb. 53

Dokumentationen zum Win32 API

Wenn Sie das Win32 API verwenden, dann brauchen Sie eine gute Referenz Dokumentation um zu verstehen welche Befehle verfügbar sind und wofür sie verwendet werden. Der erste Hafen den Sie anlaufen ist wahrscheinlich Microsoft's Developers Network im Internet (www.msdn.com), allerdings kann das für den Anfänger etwas verwirrend sein, da die Seite sehr massiv mit Informationen über jeden denkbaren Aspekt der Windows Programmierung vollgepackt ist, so dass die Navigation zur Qual werden kann. Für mich persönlich ist die beste Lösung eine kleinere, mehr fokussierte Referenz zu verwenden, und dann bei Bedarf auf die MSDN Seite zu wechseln, falls ich noch mehr Informationen über einen bestimmten Befehl oder eine Syntax benötige.

Die kleinere Referenz die ich verwende ist die 'Win32.hlp' Datei, eine Windows Hilfe Datei die eine Referenz zu den Win32 API Befehlen enthält. Diese Hilfe Datei ist kein Bestandteil des PureBasic Paketes, weshalb Sie diese selbst herunterladen und installieren müssen. Diese Datei finden Sie unter anderem auf 'PureArea', einen Link hierzu finden Sie im Anhang A (Nützliche Internet Links). Nachdem Sie die 'Win32.hlp' Datei geladen haben sollten Sie diese in einen Ordner mit dem Namen 'Help' im PureBasic Verzeichnis kopieren. Wenn sich in Ihrem PureBasic Verzeichnis noch kein 'Help' Ordner befindet, müssen Sie diesen erstellen und dann die 'Win32.hlp' Datei(en) in diesen Ordner kopieren. Manchmal wird die Hilfe Datei von weiteren Dateien begleitet, wie z.B. 'Win32.cnt' und 'Win32.kwf', in diesem Fall kopieren Sie alle Dateien zusammen in diesen Ordner. Nachdem Sie das erledigt haben, können Sie den Vorteil der Integration dieser Datei in die IDE genießen.

Nachdem die Datei korrekt installiert ist, können Sie in der IDE jeden Win32 API anwählen, indem Sie den Cursor auf den Befehl bewegen, und dann 'F1' drücken. Nun erscheint an Stelle der PureBasic Hilfe die 'Win32.hlp' Datei und zeigt auf der richtigen Seite die Beschreibung des angewählten Win32 API Befehls an. Dieses Verhalten beendet nicht die Funktion der Standard PureBasic Hilfe, es wird nur eine andere Hilfe Datei angezeigt, abhängig vom in der IDE angewählten Befehl.

Probieren Sie es selbst aus, nehmen Sie diesen Code von einem vorhergehenden Beispiel, bewegen Sie den blinkenden Cursor in den Befehl 'GetClientRect' und drücken Sie 'F1'.

```
DesktopInfo.RECT
DesktopHandle.i = GetDesktopWindow_()
GetClientRect_(DesktopHandle, @DesktopInfo)

DesktopGroesse.s = "Ihr Desktop hat folgende Größe: "
DesktopGroesse + Str(DesktopInfo\right) + " x "
DesktopGroesse + Str(DesktopInfo\bottom)
Debug DesktopGroesse
```

Sie sollten nun sehen, dass sich die 'Win32.hlp' Datei öffnet und die Seite des 'GetClientRect' Befehls anzeigt. Denken Sie daran, die 'Win32.hlp' Datei ist eine Standard Windows Hilfe Datei und als solche einfach navigierbar und voll durchsuchbar. Diese Datei ist vielleicht einen Blick wert, wenn Sie planen intensiver in die Win32 API Programmierung einzusteigen.

Nachteile der Win32 API Verwendung in PureBasic

Einer der größten Nachteile der Verwendung des Win32 API in PureBasic Programmen liegt darin, dass Sie diese Befehle nur auf einem Microsoft Windows Betriebssystem verwenden können. Das bedeutet, dass jedes Programm das Sie mit diesen Befehlen schreiben nur unter PureBasic für Windows kompilierbar ist.

Ein weiterer Nachteil liegt darin, dass das Win32 API original für die Verwendung von C und C++ als Programmiersprache entwickelt wurde, und als solches jede Menge Referenzen zu fortgeschrittenen Datentypen enthält die in PureBasic einfach nicht existieren. Für die Meisten dieser Typen können Sie aber einen PureBasic Typ als Ersatz für die C/C++ Typen verwenden. Abb. 54 zeigt eine Tabelle, in der Sie sehen welchen PureBasic Typ Sie an Stelle des aufgelisteten C/C++ Typs verwenden können. Wenn Sie zum Beispiel in der Win32 API Beschreibung eines bestimmten Befehls sehen, dass dieser eine Variable vom Typ 'DWORD32' benötigt, dann können Sie Abb. 54 entnehmen, dass Sie stattdessen den PureBasic Typ Integer verwenden können.

Sie werden bemerkt haben, dass einige Typen in Abb. 54 mit einem '†' Symbol markiert sind. Damit soll kenntlich gemacht werden, dass diese Typen nicht genau in ihre PureBasic Gegenstücke passen. Der Grund für dieses Durcheinander ist der, dass die markierten Win32 API Typen vorzeichenlose Typen sind (außer 'TBYTE'), während die PureBasic Ersatztypen alle vorzeichenbehaftet sind. Deswegen können einige Werte überlaufen wenn sie so in ihren PureBasic Gegenstücken genutzt werden. Die einzige Ausnahme ist der API Typ: 'TBYTE', der vorzeichenbehaftet ist, während sein PureBasic Gegenstück vorzeichenlos ist. Das wirft jetzt aber keine großen Probleme in Ihrem Programm auf, denn mit ein wenig Programmierarbeit können Sie einfach die vorzeichenbehafteten Werte in vorzeichenlose Werte wandeln, und auch wieder zurück. Hier ein paar Prozeduren die Sie verwenden können um all diese Problemtypen zu konvertieren:

```
;Gibt den korrekten vorzeichenlosen Wert von einem UCHAR zurück.
Procedure.w UCHAR(UCHAR.b)
  ProcedureReturn UCHAR & $FF
EndProcedure

;Gibt den korrekten vorzeichenlosen Wert von einem TBYTE zurück.
CompilerIf #PB_Compiler_Unicode
  Procedure.w TBYTE(TBYTE.c)
CompilerElse
  Procedure.b TBYTE(TBYTE.c)
  CompilerEndIf
  ProcedureReturn TBYTE
EndProcedure

;Gibt den korrekten vorzeichenlosen Wert von einem USHORT zurück.
Procedure.l USHORT(USHORT.w)
  ProcedureReturn USHORT & $FFFF
EndProcedure
```

```
;Gibt den korrekten vorzeichenlosen Wert von einem UINT zurück.
Procedure.q UINT(UINT.l)
  ProcedureReturn UINT & $FFFFFFFF
EndProcedure

;Gibt den korrekten vorzeichenlosen Wert von einem UINT32 zurück.
Procedure.q UINT32(UINT32.l)
  ProcedureReturn UINT32 & $FFFFFFFF
EndProcedure

;Gibt den korrekten vorzeichenlosen Wert von einem ULONG zurück.
Procedure.q ULONG(ULONG.l)
  ProcedureReturn ULONG & $FFFFFFFF
EndProcedure

;Gibt den korrekten vorzeichenlosen Wert von einem ULONG32 zurück.
Procedure.q ULONG32(ULONG32.l)
  ProcedureReturn ULONG32 & $FFFFFFFF
EndProcedure
```

PureBasic Ersatztypen für die Win32 API Typen

Byte	Ascii	Character	Word	Unicode
BOOLEAN BYTE CHAR	UCHAR	TBYTE † TCHAR	SHORT WCHAR WORD	USHORT

Quad	*Zeiger	Float	Double
DWORD64 INT64 LONG64 LONGLONG POINTER_64	DWORD_PTR INT_PTR LONG_PTR UINT_PTR ULONG_PTR	FLOAT	DOUBLE

Integer

BOOL	HHOOK	LANGID	LPWORD	PSHORT
COLORREF	HICON	LCID	LPWSTR	PSTR
DWORD	HIMAGELIST	LCTYPE	LRESULT	PTBYTE
DWORD32	HIMC	LONG	PBOOL	PTCHAR
HACCEL	HINSTANCE	LONG32	PBOOLEAN	PTSTR
HANDLE	HKEY	LPARAM	PBYTE	PUCHAR
HBITMAP	HKL	LPBOOL	PCHAR	PUINT
HBRUSH	HLOCAL	LPBYTE	PCRITICAL_SECTION	PULONG
HCONV	HMENU	LPCOLORREF	PCSTR	PUSHORT
HCONVLIST	HMETAFILE	LPCRITICAL_SECTION	PCTSTR	PVOID
HCURSOR	HMODULE	LPCSTR	PCWCH	PWCHAR
HDC	HMONITOR	LPCTSTR	PCWSTR	PWORD
HDDEDATA	HPALETTE	LPCVOID	PDWORD	PWSTR
HDESK	HPEN	LPCWSTR	PFLOAT	SC_HANDLE
HDROP	HRGN	LPDWORD	PHANDLE	SC_LOCK
HDWP	HRSRC	LPHANDLE	PHKEY	SERVICE_STATUS_HANDLE
HENHMETAFILE	HSZ	LPINT	PINT	UINT †
HFILE	HWINSTA	LPLONG	PLCID	UINT32 †
HFONT	HWND	LPSTR	PLONG	ULONG †
HGDIOBJ	INT	LPTSTR	PLUID	ULONG32 †
HGLOBAL	INT32	LPVOID	POINTER_32	WPARAM

Die mit '†' markierten Typen müssen nach einem Befehlsaufruf korrekt gelesen werden. Diese Typen können vorzeichenbehaftet oder vorzeichenlos sein im Gegensatz zu den PureBasic Typen die Sie verwenden.

Abb. 54

Wenn Sie zum Beispiel ein PureBasic Byte verwenden (welches vorzeichenbehaftet ist) um einen vorzeichenlosen 'UCHAR' Wert zu speichern, dann können Sie diesen Wert nur unter Verwendung der 'UCHAR()' Prozedur richtig lesen, wie hier:

```
MeinUCHAR.b = 255 ;Stellt '-1' mit Vorzeichen und '255' ohne Vorzeichen dar.
Debug MeinUCHAR ;Gibt den Wert '-1' aus.
VorzeichenloserWert.w = UCHAR(MeinUCHAR)
Debug VorzeichenloserWert ;Gibt den Wert '255' aus.
```

Der einzige Nachteil bei der Verwendung dieser Prozeduren liegt darin, (auch wenn das völlig korrekt ist) dass die Werte die sie zurückgeben doppelt so viel Speicherplatz belegen wie der Typ den Sie als Parameter übergeben haben. Schauen Sie auf die 'UCHAR()' Prozedur aus dem letzten Beispiel. Diese akzeptiert ein Byte als Parameter des 'UCHAR' das Sie lesen wollen, der Rückgabewert ist allerdings vom Typ Word und enthält den richtigen vorzeichenlosen Wert. Das ist die doppelte Größe des Original Byte. Das muss so gehandhabt werden, weil nahezu alle PureBasic Typen vorzeichenbehaftet sind. Deshalb müssen wir einen Typ verwenden, der groß genug ist den vorzeichenlosen Wert korrekt auszudrücken, um Überlauf Fehler zu vermeiden.

Das Zuweisen von vorzeichenlosen Werten an vorzeichenbehaftete Typen auf der anderen Seite ist einfach. Wie im letzten Beispiel gezeigt, können Sie einen vorzeichenlosen Wert einem vorzeichenbehafteten Typ zuweisen, allerdings wir dieser Wert immer als vorzeichenbehafteter Wert gespeichert. So speichert diese Codezeile:

```
MeinUCHAR.b = 255
Debug MeinUCHAR
```

eigentlich den Wert '-1'. Das ist so in PureBasic implementiert um die Win32 API Konvertierung zu erleichtern.

Eine kurze Bemerkung zu den Win32 API String Zeiger Typen

Wenn Sie Win32 API String Zeiger Typen verwenden ist es nützlich zu wissen, wie Sie für jeden der unterschiedlichen String Typen korrekt aus dem Speicher lesen und in diesen schreiben müssen. Für einen ein Byte pro Zeichen ASCII String würde das Win32 API zum Beispiel einen Zeiger vom Typ 'LPCSTR' verwenden, während ein Zeiger vom Typ 'LPCWSTR' auf einen Unicode String mit zwei Byte pro Zeichen zeigen würde. Auch wenn alle String Zeiger in PureBasic den Typ Integer verwenden um den eigentlichen Zeiger zu repräsentieren und die Speicheradresse darin zu speichern, so müssen Sie doch beim Peeken und Poken von Jedem unterschiedliche Modi in den 'PeekS()' und 'PokeS()' Befehlen verwenden, um die Zeichencodierung des Strings zu bewahren.

Hier sind die Win32 API String Zeiger Typen, und was noch wichtiger ist, wie Sie jeden Typ von String korrekt aus dem Speicher lesen oder ihn dort hinein schreiben:

'LPCSTR'
Ein Byte pro Zeichen (ASCII) String, auch wenn das Programm im Unicode Modus kompiliert wurde.

```
PeekS(StringZeiger.i, StringLaenge, #PB_Ascii)
PokeS(StringZeiger.i, Text.s, StringLaenge, #PB_Ascii)
```

'LPCWSTR'
Zwei Byte pro Zeichen (Unicode) String, auch wenn das Programm nicht im Unicode Modus kompiliert wurde.

```
PeekS(StringZeiger.i, StringLaenge, #PB_Unicode)
PokeS(StringZeiger.i, Text.s, StringLaenge, #PB_Unicode)
```

'LPCTSTR' 'LPTSTR' 'LPSTR' 'PCTSTR' 'PTSTR'
Exakt das gleiche wie ein Standard PureBasic String, der ein Byte pro Zeichen hat, wenn das Programm nicht im Unicode Modus kompiliert wurde und zwei Byte pro Zeichen, wenn das Programm im Unicode Modus kompiliert wurde.

```
PeekS(StringZeiger.i, StringLaenge)
PokeS(StringZeiger.i, Text.s, StringLaenge)
```

Ein Tor zu einer größeren Welt

Ich hoffe diese kurze Einführung in das Win32 API wird Sie anspornen noch mehr darüber zu lesen und vielleicht verwenden Sie es auch in Ihren Windows Programmen. Wenn Sie PureBasic verwenden um Software für Microsoft Windows zu schreiben, benötigen Sie gelegentlich API Befehle für Extrafunktionen die momentan in reinem PureBasic nicht verfügbar sind.

V. Anhang

in diesem Abschnitt befinden Sich die Anhänge zum Buch. Dieses zusätzliche Material enthält verschiedene nützliche Informationen, wie nützliche Internet Links, einige Seiten mit hilfreichen Tabellen und ein komplettes Sachwortverzeichnis mit Fachbegriffen zum Computer

Diese abschließenden Seiten versorgen Sie mit einer Schnellreferenz zu einigen der nützlicheren Themen, die Sie bestimmt von Zeit zu Zeit wieder aufschlagen werden.

A. Nützliche Internet Links

Die folgenden Seiten enthalten viele nützliche Web Adressen aus dem Internet die Sie sich einmal betrachten sollten, wenn Sie mit PureBasic loslegen. Diese Webseiten enthalten viele nützliche Informationen, wie Code Beispiele, Werkzeuge und Erweiterungen, wo Sie technische Unterstützung erhalten und wo Sie Fragen stellen können.

PureBasic Beginners
http://www.pb-beginners.co.uk/

Diese Webseite gehört zu diesem Buch. Die meisten Codes, Bilder und Sounddateien die in den Beispielen in diesem Buch verwendet wurden, können von dieser Webseite heruntergeladen werden, zusammen mit weiteren nützlichen Werkzeugen, inklusive der 'Win32.hlp' Datei.

Offizielle PureBasic Webseite
http://www.purebasic.com/

Die offizielle Webseite von PureBasic. Hier können Sie die Vollversion von PureBasic erwerben und herunterladen. Hier finden Sie weiterhin Entwickler Informationen und Sie erhalten Zugriff auf technische Unterstützung.

Offizielles PureBasic Forum
http://www.purebasic.fr/english/ (englisch)
http://www.purebasic.fr/german/ (deutsch)

Diese Seiten werden hoffentlich einer Ihrer ersten Zwischenstopps sein wenn Sie PureBasic verwenden. Alles was Sie tun müssen ist sich zu registrieren, und Sie können so viele Fragen stellen wie Sie wollen. Dieses Online Forum wird auch vom PureBasic Team mit überwacht und dieses Hilft dabei Diskussionen und Fragen zu beantworten die PureBasic betreffen. Den Reichtum an Information über PureBasic auf diesen Seiten sollten Sie nicht unterschätzen. Sie haben nicht nur zugriff auf das geteilte Wissen von Hunderten PureBasic Programmierern, sondern Sie haben auch die Möglichkeit in bereits gespeicherten Nachrichten zu suchen. Das bedeutet, dass Sie sofortigen Zugriff auf jahrelang gesammeltes PureBasic Wissen haben. Vielleicht ist Ihr Problem schon von jemand anderem gelöst worden.

PureArea
http://www.purearea.net/

Diese Seite enthält eine Menge für PureBasic relevantes Material, unter anderem Tausende von PureBasic Code Beispielen im downloadbaren 'Code Archiv', downloadbare 'User Libraries' (Benutzerdefinierte Befehlsbibliotheken) sowie Links zu Hunderten von Shareware und Freeware Werkzeugen die mit PureBasic erstellt wurden. Auf dieser Seite finden Sie auch eine Online Version des PureBasic Referenz Handbuches.

Der PureBasic Visual Designer
http://www.purebasic.be/

Diese Seite enthält Neuigkeiten und aktuelle Versionen des offiziellen PureBasic Visual Designers. Alles was mit der Entwicklung des Visual Designers zu tun hat finden Sie auf dieser Seite.

Diese Seite ist leider nicht mehr erreichbar. Wie bereits in Kapitel 9 erwähnt ist die Weiterentwicklung des Visual Designers zur Zeit völlig offen.

(Anmerkung des Übersetzers)

PureProject
http://pureproject.reelmedia.org/

Mehrere PureBasic Code Schnipsel sowie Spiele, Anwendungen und viele downloadbare 'User Libraries'. Auf dieser Seite gibt es auch einen Bereich der sich mit dem Windows API befasst.

Pure Vision
http://purevision.reelmedia.org/

Ein kommerzieller Ersatz für den Standard PureBasic Visual Designer. Mit diesem professionellen Visual Designer lassen sich auch modifizierte Benutzeroberflächen erstellen, die im Aussehen vom Windows Standard abweichen (GUI Skinning).

Microsoft Windows API Dokumentation
http://msdn.microsoft.com/library/

Microsoft's Seite zum Windows API. Auf dieser Seite finden Sie detailliert alle Informationen zum Windows Application Programming Interface. Über das Suchfeld auf der Webseite finden Sie schnell die gesuchte Information.

API-Guide
http://www.allapi.net/agnet/apiguide.shtml

Ein Programm mit dem Namen 'API-Guide', das es Ihnen ermöglicht alle verfügbaren Win32 API Befehle aufzulisten. Für alle Befehle sind Details und die erforderlichen Parameter verfügbar sowie Beispiele für fast alle Befehle. Beachten Sie, dass die Beispiele für Visual Basic entwickelt wurden, allerdings lassen sich diese leicht nach PureBasic konvertieren.

Die OGRE Engine
http://www.ogre3d.org/

Die Webseite der OGRE 3D Engine. Diese wird von PureBasic für 3D Grafik verwendet.

ModPlug XMMS Plugin
http://modplug-xmms.sourceforge.net/

Dieses Plugin wird von PureBasic zum abspielen von Modulen verwendet. Mehr Informationen erhalten Sie in Kapitel 12 (Modul Dateien)

B. Hilfreiche Tabellen

Dieser Anhang enthält viele nützliche Tabellen, einige von ihnen sind bereits zuvor in diesem Buch aufgetaucht. Diese sind hier erneut abgebildet, um es für Sie einfacher zu machen, bestimmte Dinge zu finden die Sie brauchen.

PureBasic's Numerische Typen

Name	Suffix	Speicherverbrauch	Bereich
Byte	.b	1 Byte (8 Bit)	-128 bis +127
Ascii	.a	1 Byte (8 Bit)	0 bis +255
Character	.c	1 Byte (8 Bit) (Ascii)	0 bis +255
Word	.w	2 Byte (16 Bit)	-32768 bis +32767
Unicode	.u	2 Byte (16 Bit)	0 bis +65535
Character	.c	2 Byte (16 Bit) (Unicode)	0 bis +65535
Long	.l	4 Byte (32 Bit)	-2147483648 bis +2147483647
Integer	.i	4 Byte (32 Bit)*	-2147483648 bis +2147483647
Integer	.i	8 Byte (64 Bit)*	-9223372036854775808 bis +9223372036854775807
Quad	.q	8 Byte (64 Bit)	-9223372036854775808 bis +9223372036854775807
Float	.f	4 Byte (32 Bit)	unlimitiert**
Double	.d	8 Byte (64 Bit)	unlimitiert**

* Größe abhängig vom Betriebssystem (32/64 Bit), ** Erklärung in Kapitel 13 (numerische Datentypen) Abb. 2

PureBasic's String Typen

Name	Suffix	Speicherverbrauch	Bereich
String	.s	Länge des Strings + 1 Byte	unlimitiert
String	$	Länge des Strings + 1 Byte	unlimitiert
Fester String	.s{Länge}	Länge des Strings	benutzerdefiniert*
Fester String	${Länge}	Länge des Strings	benutzerdefiniert*

* der Länge Parameter definiert die maximale Länge des Strings Abb. 3

Prioritäten von Operatoren

Priorität*	Operatoren
1	( )
2	~
3	<< >> % !
4	| &
5	* /
6	+ -
7	> >= < <= = <>
8	And Or Not XOr

* die oberen Operatoren werden zuerst verarbeitet Abb. 13

Operatoren Schnellübersicht

Operator	Beschreibung
=	Gleich. Dieser bietet zwei Verwendungsmöglichkeiten. Die Erste ist den Wert des Ausdrucks RVO der Variablen LVO zuzuweisen. Die zweite Möglichkeit ist die Verwendung des Ergebnisses des Operators in einem Ausdruck, um zu Prüfen ob der Wert des Ausdrucks LVO und RVO identisch ist (wenn sie identisch sind, gibt der Gleich Operator True zurück, anderenfalls gibt er False zurück).
+	Plus. Addiert den Wert des Ausdrucks RVO zum Wert des Ausdrucks LVO und gibt das Ergebnis zurück. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert des Ausdrucks RVO direkt in die Variable LVO addiert.
-	Minus. Subtrahiert den Wert des Ausdrucks RVO vom Wert des Ausdrucks LVO. Wenn sich LVO kein Ausdruck befindet, erhält der Wert des Ausdrucks RVO ein negatives Vorzeichen. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert des Ausdrucks RVO direkt aus der Variable LVO subtrahiert. (Dieser Operator kann nicht mit Strings verwendet werden).
*	Multiplikation. Multipliziert den Wert des Ausdrucks LVO mit dem Wert des Ausdrucks RVO. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert des Ausdrucks RVO direkt in der Variable LVO multipliziert. (Dieser Operator kann nicht mit Strings verwendet werden).
/	Division. Dividiert den Wert des Ausdrucks LVO durch den Wert des Ausdrucks RVO. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert in der Variable LVO direkt durch den Wert des Ausdrucks RVO dividiert. (Dieser Operator kann nicht mit Strings verwendet werden).
&	Bitweises AND. Sie sollten mit binären Zahlen vertraut sein, wenn Sie diesen Operator verwenden. Das Ergebnis dieses Operators ist der Wert des Ausdrucks LVO UND verknüpft mit dem Wert des Ausdrucks RVO. Diese Verknüpfung erfolgt Bit für Bit. Wenn das Ergebnis des Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird das Ergebnis direkt in dieser Variable gespeichert. (Dieser Operator kann nicht mit Strings verwendet werden).
|	Bitweises OR. Sie sollten mit binären Zahlen vertraut sein, wenn Sie diesen Operator verwenden. Das Ergebnis dieses Operators ist der Wert des Ausdrucks LVO ODER verknüpft mit dem Wert des Ausdrucks RVO. Diese Verknüpfung erfolgt Bit für Bit. Wenn das Ergebnis des Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird das Ergebnis direkt in dieser Variable gespeichert. (Dieser Operator kann nicht mit Strings verwendet werden).
!	Bitweises XOR. Sie sollten mit binären Zahlen vertraut sein, wenn Sie diesen Operator verwenden. Das Ergebnis dieses Operators ist der Wert des Ausdrucks LVO EXKLUSIV ODER verknüpft mit dem Wert des Ausdrucks RVO. Diese Verknüpfung erfolgt Bit für Bit. Wenn das Ergebnis des Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird das Ergebnis direkt in dieser Variable gespeichert. (Dieser Operator kann nicht mit Strings verwendet werden).
~	Bitweises NOT. Sie sollten mit binären Zahlen vertraut sein, wenn Sie diesen Operator verwenden. Das Ergebnis dieses Operators ist der NICHT verknüpfte Wert des Ausdrucks RVO, d.h. die Bits des Ergebnisses sind im Vergleich zum Wert des Ausdrucks invertiert. (Dieser Operator kann nicht mit Strings verwendet werden).
<	Kleiner als. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO kleiner ist als der Wert des Ausdrucks RVO, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
>	Größer als. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO größer ist als der Wert des Ausdrucks RVO, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
<=	Kleiner oder gleich. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO kleiner oder gleich dem Wert des Ausdrucks RVO ist, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
>=	Größer oder gleich. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO größer oder gleich dem Wert des Ausdrucks RVO ist, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
<>	Ungleich. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO gleich dem Wert des Ausdrucks RVO ist, dann gibt der Operator False zurück, anderenfalls gibt er True zurück.
And	Logisches AND. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO und der Wert des Ausdrucks RVO beide wahr sind, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
Or	Logisches OR. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn der Wert des Ausdrucks LVO oder der Wert des Ausdrucks RVO wahr ist, dann gibt der Operator True zurück, anderenfalls gibt er False zurück.
Not	Logisches NOT. Dient zum negieren eines booleschen Wertes. Mit anderen Worten, wenn ein Ausdruck den Wert True zurückgibt, wird dieser durch den Not Operator in False gewandelt. Wenn umgekehrt der Ausdruck RVO False zurückgibt, wird diesen in True gewandelt.
XOr	Logisches XOR. Dient zum vergleichen der Werte der Ausdrücke LVO und RVO. Wenn nur einer der Ausdrücke LVO oder RVO True zurückgibt, dann ist das Ergebnis True. Wenn beide Ausdrücke entweder True oder False sind, gibt der XOr Operator False zurück.
<<	Arithmetisches schieben nach links. Verschiebt jedes Bit im Wert des Ausdrucks LVO um die Anzahl der durch den Wert des Ausdrucks RVO definierten Stellen nach links. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert der Variable um die Anzahl der Stellen RVO geschoben. Es ist hilfreich, das Sie Binärzahlen verstanden haben wenn Sie diesen Operator verwenden. Jede verschobene Stelle wirkt wie eine Multiplikation mit dem Faktor 2.
>>	Arithmetisches schieben nach rechts. Verschiebt jedes Bit im Wert des Ausdrucks LVO um die Anzahl der durch den Wert des Ausdrucks RVO definierten Stellen nach rechts. Wenn das Ergebnis dieses Operators nicht verwendet wird und es befindet sich eine Variable LVO, dann wird der Wert der Variable um die Anzahl der Stellen RVO geschoben. Es ist hilfreich, das Sie Binärzahlen verstanden haben wenn Sie diesen Operator verwenden. Jede verschobene Stelle wirkt wie eine Division mit dem Faktor 2.
%	Modulo. Gibt den Teilungsrest der Division des Ausdrucks LVO durch den Ausdruck RVO zurück.
( )	Klammern. Sie können Klammern verwenden um einen Teil eines Ausdrucks bevorzugt oder in einer bestimmten Reihenfolge zu bearbeiten. Ausdrücke in Klammern werden vor jedem anderen Teil im Ausdruck bearbeitet. In verschachtelten Klammern werden die innersten zuerst aufgelöst und dann immer weiter nach außen.

RVO = Rechts vom Operator LVO = Links vom Operator Abb. 15

PureBasic Ersatztypen für die Win32 API Typen

Byte	Ascii	Character	Word	Unicode
BOOLEAN BYTE CHAR	UCHAR	TBYTE † TCHAR	SHORT WCHAR WORD	USHORT

Quad	*Zeiger	Float	Double
DWORD64 INT64 LONG64 LONGLONG POINTER_64	DWORD_PTR INT_PTR LONG_PTR UINT_PTR ULONG_PTR	FLOAT	DOUBLE

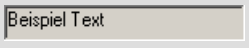
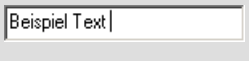
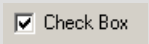
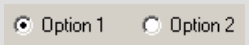
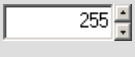
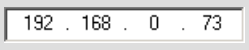
Integer

BOOL	HHOOK	LANGID	LPWORD	PSHORT
COLORREF	HICON	LCID	LPWSTR	PSTR
DWORD	HIMAGELIST	LCTYPE	LRESULT	PTBYTE
DWORD32	HIMC	LONG	PBOOL	PTCHAR
HACCEL	HINSTANCE	LONG32	PBOOLEAN	PTSTR
HANDLE	HKEY	LPARAM	PBYTE	PUCHAR
HBITMAP	HKL	LPBOOL	PCHAR	PUINT
HBRUSH	HLOCAL	LPBYTE	PCRITICAL_SECTION	PULONG
HCONV	HMENU	LPCOLORREF	PCSTR	PUSHORT
HCONVLIST	HMETAFILE	LPCRITICAL_SECTION	PCTSTR	PVOID
HCURSOR	HMODULE	LPCSTR	PCWCH	PWCHAR
HDC	HMONITOR	LPCTSTR	PCWSTR	PWORD
HDDEDATA	HPALETTE	LPCVOID	PDWORD	PWSTR
HDESK	HPEN	LPCWSTR	PFLOAT	SC_HANDLE
HDROP	HRGN	LPDWORD	PHANDLE	SC_LOCK
HDWP	HRSRC	LPHANDLE	PHKEY	SERVICE_STATUS_HANDLE
HENHMETAFILE	HSZ	LPINT	PINT	UINT †
HFILE	HWINSTA	LPLONG	PLCID	UINT32 †
HFONT	HWND	LPSTR	PLONG	ULONG †
HGDIOBJ	INT	LPTSTR	PLUID	ULONG32 †
HGLOBAL	INT32	LPVOID	POINTER_32	WPARAM

Die mit '†' markierten Typen müssen nach einem Befehlsaufruf korrekt gelesen werden. Diese Typen können vorzeichenbehaftet oder vorzeichenlos sein im Gegensatz zu den PureBasic Typen die Sie verwenden.

Abb. 54

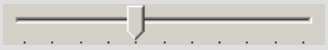
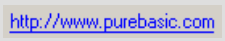
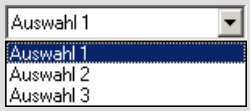
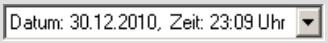
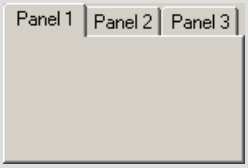
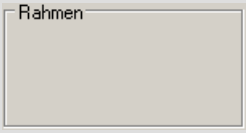
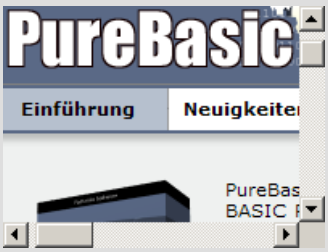
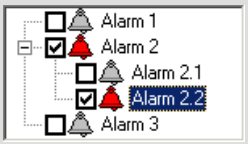
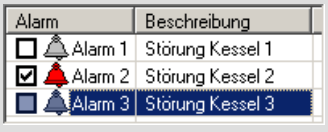
In PureBasic eingebaute Gadgets

Gadget Name	Beispiel Abbildung	Beschreibung
Button Gadget		Erstellt eine anklickbare Schaltfläche mit benutzerdefiniertem Text.
Button Image Gadget		Erstellt eine anklickbare Schaltfläche mit einem benutzerdefinierten Bild.
Text Gadget		Erstellt ein nicht editierbares Textfeld mit benutzerdefiniertem Text.
String Gadget		Erstellt ein editierbares Textfeld mit benutzerdefiniertem Text.
Check Box Gadget		Erstellt eine Box zum an- und abwählen mit einem benutzerdefinierten Text an der Seite.
Option Gadget		Erstellt Optionsknöpfe zum umschalten, mit einem benutzerdefinierten Text an der Seite.
Spin Gadget		Erstellt ein Zahlen Gadget, dessen Wert mit den Pfeilen erhöht oder erniedrigt werden kann.
IP Adress Gadget		Erstellt ein vier Felder Gadget zur Anzeige und Eingabe von IP Adressen.

(Beispiele aus Microsoft Windows)

Abb. 55

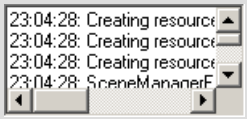
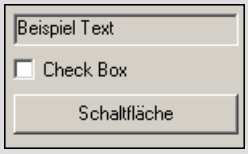
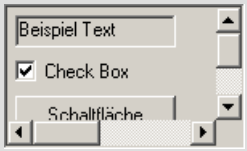
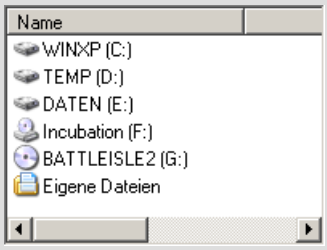
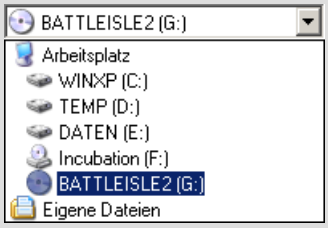
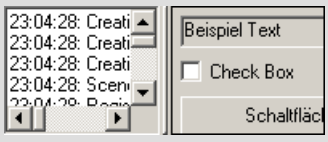
In PureBasic eingebaute Gadgets Fortsetzung

Gadget Name	Beispiel Abbildung	Beschreibung
Track Bar Gadget		Erstellt einen beweglichen Schieber um Werte in Echtzeit zu erhöhen und zu verringern.
Progress Bar Gadget		Erstellt eine steuerbare Fortschrittsanzeige die den Fortschritt einer benutzerdefinierten Aktion anzeigt.
Scroll Bar Gadget		Erstellt eine konfigurierbare Scrollbar zum Scrollen eines teilweise sichtbaren Objektes (z.B. ein Bild).
Hyperlink Gadget		Erstellt eine anklickbare Verknüpfung, ähnlich einem Hyperlink auf einer Webseite.
Combo Box Gadget		Erstellt ein editierbares, ausklappbares Auswahl Menü. Durch einen Klick auf den rechten Knopf öffnen Sie die vollständige Auswahlliste.
Date Gadget		Erstellt eine Auswahl Box, die ein benutzerdefiniertes Datum anzeigen und zurückgeben kann. Durch einen Klick auf den rechten Knopf kann ein neues Datum ausgewählt werden.
Panel Gadget		Erstellt ein Gadget mit einer benutzerdefinierten Anzahl von Tafeln. Jede Tafel hat einen benutzerdefinierten Namen und kann eine eigene Oberfläche bestehend aus anderen Gadgets haben.
Frame 3D Gadget		Erstellt einen 3D Rahmen mit einer benutzerdefinierten Überschrift. Dieser Rahmen dient nur der Zierde und eignet sich gut zum visuellen Gruppieren von Oberflächen Elementen.
Web Gadget		Erstellt ein Gadget das eine Webseite anzeigen kann. Die Seite kann aus dem Internet oder von einer lokalen Quelle wie z.B. Ihrer Festplatte stammen.
Tree Gadget		Erstellt ein Gadget das benutzerdefinierte Texte in einer baumähnlichen Struktur darstellt.
List View Gadget		Erstellt ein Gadget das benutzerdefinierte Strings in einer Liste anzeigt.
List Icon Gadget		Erstellt ein Gadget das Reihen und Spalten enthält, die mit benutzerdefinierten Strings und Icons gefüllt werden können.
Image Gadget		Erstellt ein Gadget in dem Sie ein Bild anzeigen können.

(Beispiele aus Microsoft Windows)

Abb. 55

In PureBasic eingebaute Gadgets Fortsetzung

Gadget Name	Beispiel Abbildung	Beschreibung
Editor Gadget		Erstellt ein Gadget das eine große Menge benutzerdefinierten Text aufnehmen kann, eignet sich gut als Texteditor.
Container Gadget		Erstellt ein Container Gadget das viele andere Gadgets aufnehmen kann. Der Container samt Inhalt wird bei Aktionen (z.B. Verschieben und Verstecken) als eine Einheit behandelt.
Scroll Area Gadget		Erstellt einen scrollbaren Bereich mit entsprechenden Scrollbars. Dieser Bereich kann viele andere Gadgets aufnehmen, die bei Bedarf gescrollt werden können.
Explorer List Gadget		Erstellt ein Gadget das Laufwerks- und Verzeichnis Informationen in Reihen und Spalten anzeigt, sehr ähnlich dem List Icon Gadget.
Explorer Tree Gadget		Erstellt ein Gadget das Laufwerks- und Verzeichnis Informationen in einer baumartigen Struktur anzeigt.
Explorer Combo Gadget		Erstellt ein Gadget das Laufwerks- und Verzeichnis Informationen in einem Auswahlfeld darstellt.
Splitter Gadget		Erstellt einen Teiler zwischen zwei Gadgets. Wird der Teiler bewegt, passen die Gadgets dynamisch ihre Größe an.
MDI Gadget		Erstellt ein Multiple Document Interface Gadget das mehrere Unterfenster enthalten kann. Jedes Unterfenster kann eine eigene Oberfläche bestehend aus weiteren Gadgets enthalten.
Calendar Gadget		Erstellt ein Gadget das einen konfigurierbaren Kalender anzeigt. Auf diesem Gadget kann jedes beliebige Datum angezeigt werden oder von diesem empfangen werden.

(Beispiele aus Microsoft Windows)

Abb. 55

Verknüpfte Zeichen, ASCII, Hexadezimal und Binärwert Tabelle

Zeichen	ASCII	Hex	Binär	Zeichen	ASCII	Hex	Binär
NULL	0	00	00000000	@	64	40	01000000
SOH	1	01	00000001	A	65	41	01000001
STX	2	02	00000010	B	66	42	01000010
ETX	3	03	00000011	C	67	43	01000011
EOT	4	04	00000100	D	68	44	01000100
ENQ	5	05	00000101	E	69	45	01000101
ACK	6	06	00000110	F	70	46	01000110
BEL	7	07	00000111	G	71	47	01000111
BS	8	08	00001000	H	72	48	01001000
TAB	9	09	00001001	I	73	49	01001001
LF	10	0A	00001010	J	74	4A	01001010
VT	11	0B	00001011	K	75	4B	01001011
FF	12	0C	00001100	L	76	4C	01001100
CR	13	0D	00001101	M	77	4D	01001101
SO	14	0E	00001110	N	78	4E	01001110
SI	15	0F	00001111	O	79	4F	01001111
DLE	16	10	00010000	P	80	50	01010000
DC1	17	11	00010001	Q	81	51	01010001
DC2	18	12	00010010	R	82	52	01010010
DC3	19	13	00010011	S	83	53	01010011
DC4	20	14	00010100	T	84	54	01010100
NAK	21	15	00010101	U	85	55	01010101
SYN	22	16	00010110	V	86	56	01010110
ETB	23	17	00010111	W	87	57	01010111
CAN	24	18	00011000	X	88	58	01011000
EM	25	19	00011001	Y	89	59	01011001
SUB	26	1A	00011010	Z	90	5A	01011010
ESC	27	1B	00011011	[	91	5B	01011011
FS	28	1C	00011100	\	92	5C	01011100
GS	29	1D	00011101	]	93	5D	01011101
RS	30	1E	00011110	^	94	5E	01011110
US	31	1F	00011111	_	95	5F	01011111
Space	32	20	00100000	`	96	60	01100000
!	33	21	00100001	a	97	61	01100001
"	34	22	00100010	b	98	62	01100010
#	35	23	00100011	c	99	63	01100011
$	36	24	00100100	d	100	64	01100100
%	37	25	00100101	e	101	65	01100101
&	38	26	00100110	f	102	66	01100110
'	39	27	00100111	g	103	67	01100111
(	40	28	00101000	h	104	68	01101000
)	41	29	00101001	i	105	69	01101001
*	42	2A	00101010	j	106	6A	01101010
+	43	2B	00101011	k	107	6B	01101011
,	44	2C	00101100	l	108	6C	01101100
-	45	2D	00101101	m	109	6D	01101101
.	46	2E	00101110	n	110	6E	01101110
/	47	2F	00101111	o	111	6F	01101111
0	48	30	00110000	p	112	70	01110000
1	49	31	00110001	q	113	71	01110001
2	50	32	00110010	r	114	72	01110010
3	51	33	00110011	s	115	73	01110011
4	52	34	00110100	t	116	74	01110100
5	53	35	00110101	u	117	75	01110101
6	54	36	00110110	v	118	76	01110110
7	55	37	00110111	w	119	77	01110111
8	56	38	00111000	x	120	78	01111000
9	57	39	00111001	y	121	79	01111001
:	58	3A	00111010	z	122	7A	01111010
;	59	3B	00111011	{	123	7B	01111011
<	60	3C	00111100	|	124	7C	01111100
=	61	3D	00111101	}	125	7D	01111101
>	62	3E	00111110	~	126	7E	01111110
?	63	3F	00111111	□	127	7F	01111111

Abb. 56

Verknüpfte Zeichen, ASCII, Hexadezimal und Binärwert Tabelle Fortsetzung

Zeichen	ASCII	Hex	Binär	Zeichen	ASCII	Hex	Binär
€	128	80	10000000	À	192	C0	11000000
□	129	81	10000001	Á	193	C1	11000001
‚	130	82	10000010	Â	194	C2	11000010
ƒ	131	83	10000011	Ã	195	C3	11000011
„	132	84	10000100	Ä	196	C4	11000100
…	133	85	10000101	Å	197	C5	11000101
†	134	86	10000110	Æ	198	C6	11000110
‡	135	87	10000111	Ç	199	C7	11000111
ˆ	136	88	10001000	È	200	C8	11001000
‰	137	89	10001001	É	201	C9	11001001
Š	138	8A	10001010	Ê	202	CA	11001010
‹	139	8B	10001011	Ë	203	CB	11001011
Œ	140	8C	10001100	Ì	204	CC	11001100
□	141	8D	10001101	Í	205	CD	11001101
Ž	142	8E	10001110	Î	206	CE	11001110
□	143	8F	10001111	Ï	207	CF	11001111
□	144	90	10010000	Ð	208	D0	11010000
‘	145	91	10010001	Ñ	209	D1	11010001
’	146	92	10010010	Ò	210	D2	11010010
“	147	93	10010011	Ó	211	D3	11010011
”	148	94	10010100	Ô	212	D4	11010100
•	149	95	10010101	Õ	213	D5	11010101
–	150	96	10010110	Ö	214	D6	11010110
—	151	97	10010111	×	215	D7	11010111
˜	152	98	10011000	Ø	216	D8	11011000
™	153	99	10011001	Ù	217	D9	11011001
š	154	9A	10011010	Ú	218	DA	11011010
›	155	9B	10011011	Û	219	DB	11011011
œ	156	9C	10011100	Ü	220	DC	11011100
□	157	9D	10011101	Ý	221	DD	11011101
ž	158	9E	10011110	Þ	222	DE	11011110
Ÿ	159	9F	10011111	ß	223	DF	11011111
NB Space	160	A0	10100000	à	224	E0	11100000
¡	161	A1	10100001	á	225	E1	11100001
¢	162	A2	10100010	â	226	E2	11100010
£	163	A3	10100011	ã	227	E3	11100011
¤	164	A4	10100100	ä	228	E4	11100100
¥	165	A5	10100101	å	229	E5	11100101
¦	166	A6	10100110	æ	230	E6	11100110
§	167	A7	10100111	ç	231	E7	11100111
¨	168	A8	10101000	è	232	E8	11101000
©	169	A9	10101001	é	233	E9	11101001
ª	170	AA	10101010	ê	234	EA	11101010
«	171	AB	10101011	ë	235	EB	11101011
¬	172	AC	10101100	ì	236	EC	11101100
-	173	AD	10101101	í	237	ED	11101101
®	174	AE	10101110	î	238	EE	11101110
¯	175	AF	10101111	ï	239	EF	11101111
°	176	B0	10110000	ð	240	F0	11110000
±	177	B1	10110001	ñ	241	F1	11110001
²	178	B2	10110010	ò	242	F2	11110010
³	179	B3	10110011	ó	243	F3	11110011
´	180	B4	10110100	ô	244	F4	11110100
µ	181	B5	10110101	õ	245	F5	11110101
¶	182	B6	10110110	ö	246	F6	11110110
·	183	B7	10110111	÷	247	F7	11110111
¸	184	B8	10111000	ø	248	F8	11111000
¹	185	B9	10111001	ù	249	F9	11111001
º	186	BA	10111010	ú	250	FA	11111010
»	187	BB	10111011	û	251	FB	11111011
¼	188	BC	10111100	ü	252	FC	11111100
½	189	BD	10111101	ý	253	FD	11111101
¾	190	BE	10111110	þ	254	FE	11111110
¿	191	BF	10111111	ÿ	255	FF	11111111

Abb. 56

C. Sachwortverzeichnis

Dieses Sachwortverzeichnis ist eine Sammlung von Begriffen und Abkürzungen, die Ihnen in ihrem Programmieralltag möglicherweise begegnen. In diesem Anhang habe ich versucht die einzelnen Begriffe so ausführlich wie nötig aber so kurz wie möglich zu beschreiben, um Ihnen das Leben zu erleichtern wenn Sie auf etwas treffen, mit dem Sie nicht vertraut sind.

3D Engine
Eine 3D Engine ist eine Umgebung die nur für den Zweck programmiert wurde, ein Konstrukt zur Verfügung zu stellen in dem sich 3D Grafiken realisieren lassen. Eine 3D Engine kann als eine Schicht der Abstraktion erklärt werden, die es Künstlern oder Entwicklern erlaubt, schnell und auf einfache Weise 3D Objekte zusammenzubringen. Dadurch müssen sie sich nicht mit der Programmierung auf der untersten Ebene auseinandersetzen. Eine 3D Engine stellt zum Beispiel wahrscheinlich einen 'LoadModel()' Befehl zur Verfügung, der ein 3D Modell lädt und anzeigt, was dem Benutzer die Arbeit erheblich erleichtert. 3D Engines stellen manchmal auch weitere nützliche Funktionen zur Verfügung, wie eine Kollisionserkennung und eine Physik Engine.

Absolut
Kann sich auf einen Wert oder einen Pfad beziehen. Ein absoluter Wert ist ein eigenständiger Wert, der sich nicht auf einen anderen Wert bezieht. Ein absoluter Pfad ist eine vollständige Pfadangabe. Alle Internet Adressen sind zum Beispiel absolute Pfade, wie sie in der Adressleiste Ihres Internet Browsers sehen können. Absolute Pfade werden verwendet um eindeutig sicherzustellen, dass auf eine bestimmte Datei an genau diesem Ort zugegriffen wird.

ActiveX
Eine Microsoft Spezifikation für wiederverwendbare Software Komponenten. ActiveX basiert auf COM, welches spezifiziert wie eine Komponente interagiert und mit Anderen zusammenarbeitet. ActiveX Controls können Anwendungen erweiterte Funktionalität hinzufügen, indem sie sich mit vordefinierten Modulen in die Oberfläche des Programms einhängen. Module können ausgetauscht werden, erscheinen aber stets als Teil des Original Programms. Im Internet können ActiveX Controls in Webseiten eingebunden werden, die von einen Webbrowser der ActiveX unterstützt heruntergeladen und ausgeführt werden können. ActiveX Controls können die gleiche Funktionalität zur Verfügung stellen wie jedes andere Programm das auf einem Server läuft. ActiveX Controls können vollen System Zugriff haben! In den meisten Fällen ist dieser Zugriff völlig legitim, die Vergangenheit hat allerdings gezeigt das dieser Zugriff auch von Schad-ActiveX Controls ausgenutzt wurde.

Algorithmus
Eine explizite Schritt für Schritt Prozedur, die eine Lösung zu einem gestellten Problem erstellt.

Alpha Kanal
Bildformate die diesen Unterstützen reservieren einen Teil der Pixeldaten für Transparenz Informationen. Manche 32-Bit Bildformate enthalten vier Kanäle, drei 8-Bit Kanäle für Rot, Grün und Blau, sowie einen 8-Bit Alpha Kanal. Der Alpha Kanal ist eine Maske. Sie spezifiziert wie viel der Pixelfarbe mit einer anderen, darunterliegenden Pixelfarbe eines anderen Bildes verschmolzen wird.

Anti-Virus Software
Eine Software die den Speicher und die Festplatte eines Computers auf Viren überprüft. Wenn sie einen Virus gefunden hat informiert die Anwendung den Benutzer über den Fund und bereinigt die durch den bösartigen Code verseuchte Datei, stellt sie unter Quarantäne oder löscht sie.

API (Application Programming Interface)
Eine Sammlung von Funktionen, die eine Anwendung benutzen kann um Daten anzufordern oder weiterzugeben, die auf einer niedrigeren Ebene verarbeitet wurden/werden. Hierbei kann es sich um Bibliotheken aus dem Betriebssystem oder um Bibliotheken aus Programmen von Drittanbietern handeln. Einfach gesagt ist ein API eine Schnittstelle die Ihrem Programm ermöglicht die Funktionalität eines anderen Programms zu nutzen. Das Microsoft Win32 API ist wahrscheinlich das bekannteste aller verschiedenen API's.

Applet
Eine Miniatur Anwendung die über das Internet transportiert wird, häufig als Erweiterung einer Webseite.

Array
Eine einfache Datenstruktur die gleichgroße Datenelemente speichert, die immer vom gleichen Typ sein müssen. Individuelle Elemente werden über einen Index angesprochen der aus einem fortlaufenden Bereich von Integer Zahlen besteht. Manche Arrays sind mehrdimensional, das heißt der Index setzt sich aus mehreren Integer Werten zusammen. Am gebräuchlichsten sind eindimensionale- und zweidimensionale Arrays.

ASCII (American Standard Code for Information Interchange)
Ein Zeichen Codierungs Schema, das einzelnen Zeichen wie Buchstaben, Zahlen, Satzzeichen und anderen Symbolen einen numerischen Wert zuweist. ASCII sieht nur sieben Bit Pro Zeichen vor (maximal 128 Zeichen). Die ersten 32 Zeichen sind nicht druckbare Steuerzeichen (Zeilenvorschub, Seitenumbruch, usw.), die zum steuern von Geräten verwendet werden. Erweiterte ASCII Zeichensätze fügen weitere 128 Zeichen hinzu, die zwischen verschiedenen Computern, Programmen und Schriftarten variieren können. Computer verwenden diese erweiterten Zeichen für Zeichen mit Umlauten, grafische Zeichen oder spezielle Symbole.

ASCII Art
Zeichnungen die unter Verwendung von ASCII Zeichen erstellt wurden. Üblicherweise werden nicht proportionale Schriftarten verwendet, um sicherzustellen dass jedes Zeichen in der Grafik die gleiche Breite hat.

ASM (Assembler Sprache)
Eine von Menschen lesbare Form von Maschinencode den eine spezifische Computer Architektur verwendet. Der Code wird dadurch lesbar gemacht, indem die rohen numerischen Werte mit Mnemonics genannten Symbolen ersetzt werden.

Assembler
Ein Computerprogramm das Assemblersprache in ausführbaren Code übersetzt.

Attribute
Eigenschaften die allen Dateien und Verzeichnissen zugewiesen werden. Attribute enthalten: Nur Lese-, Archiv-, Versteckt- oder System Status.

Back Door
Eine Funktion die Programmierer in Programme einbauen, die Zugriff auf besondere Privilegien gewähren, die dem normalen Anwender normalerweise verwehrt sind. Oft verwenden Programmierer diese Technik um Fehler in ihrem Programm beheben zu können. Wenn Hacker und andere lernen diese Back Doors zu nutzen, kann diese Funktionalität zum Sicherheitsrisiko werden.

Background Task
Ein Programm das vom System ausgeführt wird, für den Anwender aber unsichtbar ist. Das System weist diesen Hintergrund Programmen üblicherweise weniger Rechenzeit zu als den Programmen im Vordergrund. Bösartige Programme werden auf einem System häufig als Background Task ausgeführt, damit der Anwender die ungewünschte Software nicht bemerkt, wie z.B. Viren, Spyware, usw.

Backup
Ein Duplikat von Daten, das zu Archivierungszwecken angefertigt wurde, oder zum Schutz vor Verlust oder Beschädigung des Originals.

Batch Dateien
Text Dateien die eine MS-DOS Anweisung pro Zeile enthalten. Nach dem Start werden alle Zeilen fortlaufend abgearbeitet. Ein Beispiel für diese Dateien ist die auch unter Windows Systemen noch zu findende 'AUTOEXEC.BAT', die ausgeführt wird wenn der Rechner hochgefahren wird. Die Dateien haben die Endung '*.bat'.

Binär
Bezieht sich auf das Basis-Zwei Zahlensystem . Dieses System enthält nur zwei Zahlen, '0' und '1'.

BIOS (Basic Input/Output System)
Ein Basis Betriebssystem, das einige Programme ausführt die den Rechner initialisieren und starten bevor das Betriebssystem gestartet werden kann. Das BIOS befindet sich im ROM Bereich der Hauptplatine und ist üblicherweise konfigurierbar.

BIT (binäre Ziffer)
Die kleinste Informationseinheit die ein Computer verarbeiten kann. Ein Bit kann den Wert '0' oder '1' haben.

Bit Tiefe (Farbtiefe)
Die Anzahl der Bits die zur Beschreibung der Farbe eines einzelnen Pixels in einem digitalen Bild oder auf dem Computer Monitor benötigt werden. Übliche Farbtiefen sind 1 Bit, 8 Bit, 16 Bit, 24 Bit und 32 Bit.

Booten
Den Rechner Starten (cold boot) oder Reseten (warm boot), um ihn in Betriebsbereitschaft für die Ausführung von Anwenderprogrammen zu versetzen. Das Booten des Rechners startet verschiedene Programme die den Rechner Prüfen und diesen betriebsbereit machen.

Bug
Ein unbeabsichtigter Fehler in einem Programm der eine vom Anwender nicht beabsichtigte Funktion im Programm hervorruft.

Byte
Eine Gruppe von acht Bits (binäre Ziffern). Weiterhin der Name eines Integer Typs der durch die Verwendung von acht Bits ausgedrückt werden kann.

Cache
Ein Speicherbereich in dem häufig benutzte Daten temporär zwischengespeichert werden um die Zugriffsgeschwindigkeit zu erhöhen.

CGI (Common Gateway Interface)
Ein Standard zum Austausch von Informationen zwischen einem Web Server und einem externen Computerprogramm. Das externe Programm kann in jeder beliebigen Programmiersprache geschrieben werden die vom Betriebssystem unterstützt wird, auf dem der Web Server läuft. CGI selbst ist keine Programmiersprache.

Checksumme
Eine Identifikations Zahl die aus der Charakteristik einer Datei berechnet wird. Die kleinste Veränderung in einer Datei ändert auch ihre Prüfsumme.

Child
Ein Fenster (oder Objekt) das auf einem anderen Fenster (oder Objekt) basiert oder mit diesem eng verbunden ist.

Class
Eine OOP Vorlage die instanziert werden kann um Objekte mit gebräuchlichen Definitionen, Eigenschaften, Methoden und Verhaltensweisen zu erstellen.

Client
Ein Computer System oder Teil einer Software, das auf einen Dienst zugreift, der von einem anderen Rechner über eine Art von Netzwerk zur Verfügung gestellt wird. Ein Web Browser ist ein Client von einem Web Server.

Code
Ein umgangssprachlicher Begriff für Quelltext.

Coding
Ein umgangssprachlicher Begriff für Computer Programmierung.

Compile Time
Die Zeit zu der ein Compiler einen in einer Programmiersprache geschriebenen Quelltext in eine ausführbare Form übersetzt.

Compiler
Ein Software Entwicklungs Werkzeug, das den Quelltext einer höheren Programmiersprache in Maschinen Anweisungen (Objekt Code) übersetzt, die von einer bestimmten Computer Architektur verstanden und ausgeführt werden können.

Constant (Konstante)
Ein benanntes Element das während der Programmausführung einen Konstanten Wert zurückgibt. Sie können überall in Ihrem Code Konstanten an Stelle von echten Werten verwenden. Eine Konstante kann ein String oder ein numerischer Wert sein.

Cookies
Textblöcke die in einer Datei auf der Festplatte Ihres Computers abgelegt werden. Web Seiten verwenden Cookies um bei einem erneuten Besuch der Seite einen Benutzer identifizieren zu können. Cookies können Anmeldedaten wie Benutzername und Kennwort oder auch benutzerdefinierte Einstellungen enthalten. Wenn Sie eine bereits zuvor besuchte Web Seite aufrufen, kann der Server die im Cookie abgelegten Daten auswerten und die Web Seite an die entsprechenden Einstellungen anpassen.

CPU (Central Processing Unit)
Der Hauptprozessor des Computers der alle Prozesse im Computer steuert.

Data Member
In der OOP ist ein Data Member eine Variable die innerhalb eines Klassen Objektes definiert und dort eingekapselt ist. Jede Instanz eines Objektes enthält einen einmaligen Wert für jeden Data Member in der Klasse, im Gegensatz zur Klasse die eine Variable enthält die jede Instanz teilt. Es ist ein anderer Name für eine Eigenschaft.

Datentyp
Die Charakteristik einer Variable die beschreibt ob sie numerische, alphabetische oder alphanumerische Werte aufnehmen kann. Weiterhin werden die Maximalwerte die diese Variable aufnehmen kann definiert, und wie diese Informationen im Speicher des Computers abgelegt werden.

DirectX
Eine Sammlung von Application Programming Interfaces (API's) die von Microsoft zur Verfügung gestellt wird um Hochleistungs Multimedia Anwendungen zu erstellen. DirectX ermöglicht es Programmen über Schnittstellen die eingebauten Hochgeschwindigkeits 3D Routinen zu verwenden, die eine schnellere Verarbeitung auf der Grafikkarte ermöglichen. Das DirectX API ermöglicht auch leichten Zugriff auf die Hardware des Computers, wie z.B. Soundkarte, Joystick oder Netzwerkadapter.

DLL (Dynamic Linked Library)
Eine Datei die kompilierten Code enthält, der dynamisch von Programmen geladen und ausgeführt werden kann. Da viele verschiedene Programme den Code der gleichen DLL nutzen können, und deshalb diesen Code nicht in ihrer eigenen Datei unterbringen müssen, reduziert sich der benötigte Speicherplatz für diese Programme dramatisch.

DOS (Disk Operating System)
Generell jedes Computer Betriebssystem, jedoch oft als Kurzform für 'MS-DOS' verwendet, dem Windows Vorgänger von Microsoft.

Double (Doppelt genaue Fließkommazahl)
Eine Fließkommazahl die den doppelten Speicherplatz einer Fließkommazahl mit einfacher Genauigkeit belegt um noch genauere Berechnungen durchführen zu können und mehr Informationen über die Zahl hinter dem Komma abspeichern zu können. Doubles verwenden üblicherweise acht Byte im Speicher.

Encapsulation (Kapselung)
Ein OOP Schema zum definieren von Objekten. Kapselung versteckt die internen Spezifikationen eines Objektes und legt nur die externen Schnittstellen offen. Dadurch müssen die Nutzer dieses Objektes nur auf die Schnittstellen zugreifen. Durch die Kapselung ist es möglich die internen Daten und Methoden eines Objektes zu ändern oder zu aktualisieren, ohne dass der Anwender die Art der Verwendung des Objektes anpassen muss.

Encryption
Die Verschlüsselung von Daten um diese ohne Schlüssel nicht interpretieren zu können.

EXE (Executable) Datei
Eine Datei die ausgeführt werden kann, wie z.B. ein Computerprogramm. Üblicherweise erfolgt die Ausführung durch einen Doppelklick auf das Icon der Datei oder eine Verknüpfung auf dem Desktop, oder durch die Eingabe des Programmnamens in der Kommandozeile. EXE Dateien können auch von anderen

Dateien ausgeführt werden, wie zum Beispiel Batch Dateien oder verschiedene Script Dateien. Die meisten der bekannten Computerviren befallen EXE Dateien.

Expression (Ausdruck)
Eine Kombination von Werten, Strings, Operatoren, Befehlen oder Prozeduren, die gemäß den Regeln der Vorrangigkeit abgearbeitet werden und dann einen Wert zurückgeben.

File Extension (Dateierweiterung)
Ein Bezeichner aus drei oder vier Buchstaben, der das Datenformat einer Datei oder die Anwendung, die sie erstellt hat, identifiziert.

Firewall
Diese schützt den Computer normalerweise im Netzwerk vor der direkten Kommunikation mit anderen externen Computer Systemen. Die Firewall Software analysiert den Informationsaustausch zwischen diesen beiden Systemen und blockiert die Daten, wenn diese nicht den vorkonfigurierten Regeln entsprechen.

Fließkommazahl (oder Realzahl)
Eine Zahl die einen Dezimalanteil hat, auch wenn dieser Dezimalanteil Null ist. Eine Fließkommazahl wird häufig auch als 'Realzahl' oder abgekürzt als 'Real' bezeichnet. Fließkommazahlen belegen üblicherweise 4 Byte im Speicher.

Funktion
Das gleiche wie Prozedur

Hexadezimal
Ein Zahlensystem das die Basis 16 verwendet. Die ersten zehn Ziffern sind '0' bis '9' und die nächsten sechs sind 'A' bis 'F'. Dadurch ist es möglich sehr große Zahlen mit wenigen Zeichen darzustellen. 'FF' in Hexadezimal entspricht '255' in Dezimal. Webseiten verwenden Hexadezimal Werte um 24-Bit Farbwerte auszudrücken.

High Level Programmiersprache (Höhere Programmiersprache)
Eine Programmiersprache in der jeder Befehl mit vielen Maschinenanweisungen korrespondiert. Befehle in dieser Sprache müssen mit einem Compiler oder Interpreter übersetzt werden, bevor sie auf einer bestimmten Computer Architektur ausgeführt werden können. Eine Höhere Programmiersprache ist Anwenderfreundlicher und für Menschen leichter lesbar. Diese Sprachen arbeiten häufig auf verschiedenen Plattformen und abstrahieren von der niedrigen Ebene der Prozessor Operationen. PureBasic ist eine Höhere Programmiersprache.

HTML (Hypertext Markup Language)
Die Beschreibungssprache zum erstellen von HTML Webseiten für das Internet.

HTTP (Hypertext Transfer Protocol)
Das darunterliegende Netzwerkprotokoll im Internet. HTTP definiert wie Nachrichten formatiert und übermittelt werden, und welche Aktionen Web Server und Browser als Antwort auf verschiedene Befehle ausführen müssen. Wenn Sie zum Beispiel eine URL in Ihren Browser eingeben, wird eigentlich ein HTTP Befehl an den Web Server geschickt, der die entsprechende Web Seite auf vorhandensein überprüft und diese dann zu Ihnen übermittelt.

Hyperlink
Eine Verknüpfung in einem Dokument, die auf eine Information im gleichen Dokument oder in einem anderen Dokument verweist. Diese Verknüpfungen werden üblicherweise durch markierte Wörter oder Bilder repräsentiert. Wenn ein Leser einen Hyperlink anklickt, dann wird auf dem Bildschirm das verknüpfte Dokument angezeigt.

Inheritance (Erben)
Der Prozess in OOP in dem ein Objekt Funktionalität und Daten von einer abgeleiteten Klasse erbt.

Integer
Eine Zahl ohne Dezimalpunkt. Integer Werte können kleiner, gleich oder größer Null sein.

Internet
Ein elektronisches Netzwerk von Computern mit dem heutzutage fast alle Universitäten, Regierungen und Forschungseinrichtungen in der Welt verbunden sind. Die ursprüngliche Einrichtung fand im Jahr 1969 statt, um ein Rückgrat für die Kommunikation im Falle eines nuklearen Krieges zu haben. Seit der Entwicklung von einfach zu verwendender Software hat die Nutzung in der Öffentlichkeit für Informationsbeschaffung und für kommerzielle Zwecke Explosionsartig zugenommen.

Interpreter
Ein Programm das Anweisungen die in eine Höheren Programmiersprache geschrieben wurden zeilenweise direkt aus dem Quelltext ausführt. Dies steht im Kontrast zu einem Compiler, der nicht den Quelltext ausführt sondern ihn in ein ausführbares Format übersetzt (Objekt Code). Dieser wird dann später verlinkt und ausgeführt.

IP (Internet Protocol) Adresse
Eine Zahl die jeden Sender und Empfänger von Informationen in einem TCP/IP Netzwerk identifiziert. Die IP Adresse besteht aus vier Zahlen, die jeweils durch eine Punkt separiert sind, zum Beispiel: '127.0.0.1'.

IRC (Internet Relay Chat)
Eine Form der sofortigen Kommunikation im Internet. Er wurde hauptsächlich zur Gruppen Kommunikation in Diskussionsforen entwickelt, die sich Channels nennen. Es ist aber auch eine Eins zu Eins Kommunikation möglich.

JavaScript
Eine Scriptsprache die mit einem geeigneten Interpreter ausgeführt werden kann, wie zum Beispiel in einem Web Browser.

LAN (Local Area Network)
Ein Local Area Network ist ein Computer Netzwerk, das einen lokalen Bereich abdeckt, wie zuhause, im Büro oder eine kleine Gruppe von Gebäuden z.B. eine Schule.

Link Time
Die Zeit während der Kompilierung eines Programmes in der der Objekt Code zum finalen Executable gelinkt wird.

Linked List (Liste)
Eine Datenstruktur die den Anschein erweckt es handele sich um ein dynamisches Array. Sie erlaubt es dynamisch während der Laufzeit des Programmes Elemente hinzuzufügen oder zu löschen. Listen können mit jedem beliebigen in einer Programmiersprache verfügbaren Datentyp erstellt werden. Anders als beim Array enthält jedes Element Zeiger zum vorherigen und zum nächsten Listenelement, da nicht alle Elemente an einem Stück im Speicher liegen, was auch die Geschwindigkeit des Zugriffs gegenüber den Arrays reduziert.

Linker
Ein Programm das kompilierte Objekt Code Dateien mit anderen Datendateien verbindet um ein ausführbares Programm zu erstellen. Ein Linker kann auch weitere Funktionen haben, wie z.B. die Erstellung von Bibliotheken.

Linken
Der Prozess der Verbindung von Objekt Code zum finalen Executable.

Linux
Ein freies, quelloffenes Betriebssystem das auf Unix basiert. Linux wurde ursprünglich von Linus Torvalds mit der Hilfe von Entwicklern rund um den Globus erstellt.

Localhost
Das Computersystem an dem der Anwender arbeitet, diesem wird oft die IP Adresse '127.0.0.1' zugewiesen.

Long
Ein Integer Typ der vier Byte im Speicher belegt.

Low Level Programmiersprache
Eine Programmiersprache die nur wenig oder überhaupt nicht vom Mikroprozessor des Computers abstrahiert. Das Wort 'Low' soll nicht implizieren dass es sich im Vergleich zur Höheren Programmiersprache um eine minderwertigere Sprache handelt, sondern bezieht sich auf den niedrigen Grad der Abstraktion zwischen der Sprache und der Hardware die sie kontrolliert. Ein Beispiel für eine Low Level Sprache ist Assembler.

Mac OS X
Die letzte Version von Mac OS, dem Betriebssystem das für Apple Macintosh Computer verwendet wird. Mac OS X basiert anders als Seine Vorgänger auf einem Unix Kern und hat eine aktualisierte Benutzer Oberfläche.

Malicious Code (Bösartiger Code)
Ein Code der in der Absicht erstellt wurde ein System oder die darin enthaltenen Daten zu beschädigen, oder das System von der Verwendung auf normale Art abhält.

Malware
Ein Oberbegriff für bösartige Software wie z.B.: Viren, Trojanische Pferde, Spyware, Adware, usw.

MBR (Master Boot Record)
Das Programm das im Master Boot Sektor liegt. Dieses Programm liest die Partitionstabelle, ermittelt die zu bootende Partition und übergibt die Kontrolle an das Programm im ersten Sektor dieser Partition. Auf jeder physikalischen Festplatte gibt es nur einen Master Boot Record.

MBS (Master Boot Sector)
Der erste Sektor auf der Festplatte. Dieser Sektor befindet sich in Sektor '1', Kopf '0', Spur '0'. Dieser Sektor enthält den Master Boot Record.

Member Funktion
Eine Funktion oder Prozedur die Teil einer OOP Klasse ist und üblicherweise Berechnungen an Member Daten der Klasse vornimmt. Das ist ein anderer Name für Methode

Methode
Eine Funktion oder Prozedur die Teil einer OOP Klasse ist und üblicherweise Berechnungen an Member Daten der Klasse vornimmt.

MS-DOS (Microsoft Disk Operating System)
Das Betriebssystem, das vor Windows für die IBM Plattform entwickelt wurde. Windows 3.x, 95 und 98 basieren noch sehr stark auf MS-DOS und können die meisten MS-DOS Befehle ausführen.

Newsgroup
Ein elektronisches Forum in dem Leser Artikel und Nachfolge Nachrichten zu einem bestimmten Thema veröffentlichen können. Eine Internet Newsgroup erlaubt es Menschen rund um den Globus über gängige interessante Themen zu diskutieren. Jeder Newsgroup Name gibt Information über das Thema der Newsgroup.

NTFS (New Technology File System)
Ein Dateisystem das zum organisieren und zusammenhalten von Dateien verwendet wird. Es ist das Standard Dateisystem von Windows NT und seinen Abkömmlingen: Windows 2000, Windows XP, Windows Vista und Windows 7.

Objekt
Eine Instanz einer OOP Klasse. Diese Objekt erbt alle Funktionalität und Daten die in die Original Klasse programmiert wurden.

Object Code
Die Ausgabe des Compilers und Assemblers, und die Eingabe und Ausgabe des Linkers sind Dateien die Oject Code enthalten. Es gibt verschiedene standardisierte Proprietäre Object Dateiformate. Das bedeutet, dass der Objekt Code von einer Entwicklungsumgebung selten von einer anderen Entwicklungsumgebung gelesen werden kann.

OCX (OLE Control Extension)
Ein unabhängiges Programmmodul auf das von anderen Programmen in einer Windows Umgebung zugegriffen werden kann. OCX Controls sind mittlerweile überholt und wurden durch die ActiveX Controls ersetzt. OCX Controls haben die Dateiendung '*.ocx', und auch wenn sie mittlerweile überholt sind, sind sie kompatibel mit ActiveX. Das bedeutet, ein ActiveX kompatibles Programm kann auch diese Module nutzen.

OLE (Object Linking and Embedding)
OLE wurde umbenannt und in überholter Form durch ActiveX ersetzt.

OOP (Objekt Orientierte Programmierung)
Eine Art der Programmierung die Kapselung, Vererbung und Polymorphismus unterstützt. Manche Programmiersprachen sind zwangsweise Objekt Orientiert, wie z.B. C++, Java und Phyton, und als solche erlaube sie es, die real oder imaginär erstellten Objekte sehr effektiv in der Programmiersprache zu nutzen.

Open Source Software
Betrifft jedes Programm von dem der Quelltext zur Verwendung oder Modifikation von anderen Anwendern verfügbar ist. Open Source Software wird üblicherweise als öffentliches Gemeinschaftsprojekt entwickelt und ist für alle frei verfügbar.

OpenGL (Open Grapics Library)
Eine Spezifikation zur Definition eines sprachunabhängigen API zum erstellen von Anwendungen die 2D und 3D Grafiken Rendern. OpenGL ist Quelloffen und Plattformübergreifend, anders als DirctX welches der größte Konkurrent ist.

Operating System (Betriebssystem)
Das Betriebssystem ist die darunterliegende Software Schicht, die es ihnen erlaubt mit dem Computer zu interagieren. Das Betriebssystem verwaltet Dinge, wie z.B. den Speicher des Computers, die Kommunikation sowie das Task Management. Beispiele für gängige Betriebssysteme sind: Mac OS X, Linux und Windows.

Parent
Ein übergeordnetes oder angehängtes Fenster (oder Objekt) das Unterfenster (oder Objekte) erstellt und manchmal auch verwaltet.

Polymorphismus
Ist ein Begriff aus der OOP und bedeutet die Möglichkeit zu haben, eine einmalige Definition mit verschiedenen Datentypen nutzen zu können. Zum Beispiel könnte eine polymorphe Prozedur Definition verschiedene Typen zurückgeben, nicht nur einen, und ein einfache Polymorpher Operator könnte mit Ausdrücken aus verschiedenen Typen arbeiten.

POP (Post Office Protocol)
Ein E-Mail Protokoll das dem Client erlaubt E-Mails von einem entfernten Server über eine TCP/IP Verbindung abzuholen. Nahezu alle Nutzer von E-Mail Accounts eines Internet Service Providers können auf ihre E-Mails mit einer Client Software zugreifen die das POP (Post Office Protocol Version 3) verwendet.

Port
(1) Eine Schnittstelle über die Daten gesendet und empfangen werden können. (2) 'To Port' kann den Vorgang bedeuten, einen Programm Quelltext zu portieren (manchmal auch in eine andere Programmiersprache) um ihn in kompilierter Form auf einer anderen Plattform ausführen zu können.

Procedure (Prozedur)
Eine benannte Sequenz von Anweisungen die als eine Einheit beim Aufruf ihres Namens ausgeführt wird. Prozeduren können auch Parameter übergeben werden um benutzerdefinierte Daten in die Sequenz zu schleusen. Anders als Unterroutinen können Prozeduren einen berechneten Wert zurückgeben. Prozeduren können mehrfach aufgerufen werden und ermöglichen es dem Programm einen Code wiederholt auszuführen ohne dass dieser ein weiteres mal programmiert werden muss.

Prozess
Eine laufende Programminstanz. Ein Multitasking Betriebssystem schaltet zwischen den Prozessen um, um den Anschein einer simultanen Ausführung zu erwecken. In der Realität kann immer nur ein Prozess die CPU verwenden.

Property (Eigenschaft)
In der OOP ist eine Eigenschaft für eine in einem Klassen Objekt definierte und gekapselte Variable. Jede Instanz enthält einen einmaligen Wert für jede Eigenschaft in der Klasse, im Gegensatz zu der Klasse die eine Variable hat die in jeder Instanz geteilt wird.

Protocol
Eine formale Beschreibung von Nachrichten Formaten und die Regeln die zwei Computer einhalten müssen um diese Nachrichten auszutauschen.

Quad
Ein Integer Typ der acht Byte im Speicher belegt

Radix Punkt
In der Mathematik bezieht sich der Radix Punkt auf das Symbol, das bei der numerischen Darstellung von Zahlen zum separieren des ganzzahligen Teils (links vom Radix) vom Bruchteil der Zahl (rechts vom Radix) dient. Der Radix Punkt ist üblicherweise ein kleiner Punkt (oder ein Komma) der sich auf der Basislinie oder auf halber Ziffernhöhe befindet. Im Basis-Zehn System wird der Radix Punkt auch Dezimalpunkt genannt.

Refresh Rate (Bildwiederholrate)
Die Anzahl der Bildneuzeichnungen auf dem Monitor in einer Sekunde. Die Bildwiederholrate wird in Hertz (Hz) angegeben, so dass eine Bildwiederholrate von 75 Hz angibt, dass das Monitorbild 75 mal pro Sekunde neu gezeichnet wird. Die Bildwiederholrate eines bestimmten Monitors hängt von dessen Spezifikation und der verwendeten Grafikkarte ab.

Registry
Eine interne Datenbank die Microsoft Windows verwendet um Hardware und Software Konfigurations Informationen, Benutzereinstellungen und Installations Informationen abzuspeichern.

Relativ
Kann sich auf einen Wert oder einen Pfad beziehen. Ein relativer Wert ist ein abhängiger Wert der sich auf einen anderen Wert bezieht. Ein relativer Pfad ist eine Pfadangabe die sich auf einen anderen Pfad bezieht. Relative Pfade werden üblicherweise verwendet, wenn Sie den Ablageort von Ressourcen für Ihr Programm im Quelltext angeben, die dann relativ zum Executable sind. Wenn Sie zum Beispiel mit Ihrem Programm ein Bild aus dem Ordner 'Bilder' laden wollen, dann wird der relative Pfad als '\Bilder' definiert, da sich dieser Ordner im gleichen Verzeichnis wie das Executable befindet, und deshalb relativ zum Programm ist.

Reset
Den Computer ohne ausschalten neu starten.

RGB (Rot, Grün und Blau)
Das sind die primären Farben des Lichtes die der Computer verwendet um ein Bild auf dem Monitor anzuzeigen. RGB Farben setzen sich üblicherweise aus Rot, Grün und Blau Werten zusammen, die in einem Bereich zwischen '0' und '255' liegen. Diese Farben werden dann nach dem Additiven Farbmodell gemischt. Wenn zum Beispiel alle Farbwerte das Maximum (255) haben, dann ist die resultierende Farbe weiß.

Runtime (Laufzeit)
Die Zeit zu der das Programm läuft oder ausgeführt wird.

SDL (Simple DirectMedia Layer)
Ein Open Source, Application Programming Interface (API) zum erstellen von Hochleistungs Multimedia Anwendungen. SDL ist sehr einfach, es fungiert als dünner, plattformübergreifender Vermittler, der Unterstützung für 2D Pixel Operationen, Sound, Dateizugriff, Ereignisbehandlung, Timing, Threading und vieles mehr zur Verfügung stellt. OpenGL wird häufig mit SDL verwendet um schnelles 3D Rendern zur Verfügung zu stellen. SDL wir häufig als plattformübergreifendes DirectX bezeichnet, allerdings weist es bei fortgeschritteneren Funktionen einige Mängel auf.

Server
Jede Anwendung oder jeder Computer der einen anderen beliefert. Zum Beispiel werden die Computer die die Web Seiten speichern als Server bezeichnet, denn Sie beliefern Client Anwendungen wie einen Web Browser mit Web Ressourcen.

Service
Ein Programm das automatisch als Teil des Betriebssystems während des Start Prozesses gestartet werden kann und dann kontinuierlich als Hintergrund Task ausgeführt wird.

Shareware
Software die zu Testzwecken kostenlos angeboten wird, die allerdings eine Registrierung beim Autor erfordert um die volle Funktionalität zu erhalten. Wenn Sie sich nach der Testphase entscheiden die Software nicht weiter zu verwenden, dann löschen Sie diese einfach. Die Verwendung von Shareware jenseits der Testperiode wird als Software Piraterie behandelt.

SMTP (Simple Mail Transport Protocol)
Ein einfaches E-Mail Zustellformat zum transportieren von E-Mail Nachrichten zwischen den Servern. Das ist momentan der Standard für den E-Mail Transport im Internet.

Software Pirat (Raubkopierer)
Eine Person die das Copyright Gesetz einer Software bricht. Gesetze beziehen sich immer auf das Land in dem diese Person lebt.

Source Code (Quelltext)
Der Code, den ein Programm ausmacht bevor es kompiliert wurde. Es handelt sich hierbei um die Original Erstellungs Anweisungen des Programms, die dem Compiler mitteilen welche Funktionen das Programm im kompilierten Zustand ausführen soll.

Statement (Anweisung)
Ein Programmierbefehl der selten einen Wert zurückgibt und andere Komponenten wie z.B. Konstanten oder Ausdrücke enthalten kann. Eine oder mehrere Anweisungen sind nötig um ein Programm zu erstellen.

Static Library
Eine Bibliothek deren Code zur Link Zeit in das Executable eingebunden und dort fest im Programm verankert wird. Wenn das Programm fertig kompiliert ist benötigt es diese Bibliothek nicht länger, der gesamte Code befindet sich nun im Executable.

String
Eine sequentielle Folge von Buchstaben, Zahlen und anderen Zeichen die in (doppelten oder einfachen) Anführungszeichen eingeschlossen sind.

Subroutine (Unterroutine)
Eine benannte Sequenz von Anweisungen die als eine Einheit ausgeführt wird wenn auf den Beginn der Sequenz gesprungen wird, üblicherweise mit einem 'Goto' oder 'GoSub' Befehl. Nachdem die Sequenz ausgeführt wurde, wird das Programm an der Herkunftsstelle der Sprunganweisung weiter fortgeführt. Die Rückkehr erfolgt üblicherweise durch einen 'Return' Befehl. Anders als Prozeduren können Unterroutinen keinen berechneten Wert zurückgeben. Unterroutinen können mehrmals angesprungen werden und ermöglichen es dem Programm den darin enthaltenen Code wiederholt zu nutzen, ohne dass dieser mehrfach in das Programm geschrieben werden muss.

Syntax
Die Regeln, nach denen Wörter in einem Programm Quelltext kombiniert werden müssen um Befehle zu formen die vom Compiler als gültig interpretiert werden.

TCP/IP
Diese Garnitur von Internet Protokollen ist eine Sammlung von Kommunikations Protokollen die einen Protokoll Stack implementieren, auf dem das Internet und die meisten kommerziellen Netzwerke laufen. Der Name TCP/IP setzt sich aus den beiden wichtigsten enthaltenen Protokollen zusammen: dem Transmission Control Protocol (TCP) und dem Internet Protocol (IP), was auch die beiden zuerst definierten waren.

Thread (Thread of Execution)
Ein Thread ist eine einzelne Sequenz von ausgeführten Anweisungen die als separater Task, aber als Teil des Hauptprozesses laufen. Ein Prozess kann mehrere Threads parallel starten, und jeder arbeitet als eigenständiger Task. Ein Beispiel wäre das Warten auf die Rückgabe eines Wertes von einer zeitintensiven Aufgabe, die zuerst beendet werden muss bevor das Programm mit einer anderen Aufgabe fortfahren kann. Wenn ein Thread seine Arbeit erledigt hat wird er angehalten oder zerstört. Programme mit mehreren Threads können bei richtiger Programmierung bestimmte Aufgaben schneller abarbeiten als Programme

ohne Threads, da bestimmte Prozessoren die simultane Ausführung von zwei oder mehreren Threads unterstützen, wie z.B. die Intel Hyperthreading CPU Familie.

Trojanisches Pferd
Ein bösartiges Programm das vorgibt nützlich und nicht schädlich zu sein. Wenn es allerdings gestartet wird, tut es etwas das der Anwender nicht erwartet und manchmal überhaupt nicht realisiert. Trojaner sind keine Viren, da sie sich nicht reproduzieren, allerdings können sie auch sehr destruktive Verhaltensweisen haben.

UNC (Univeral Naming Convention)
Ein Standard um Server, Drucker und andere Ressourcen im Netzwerk zu identifizieren. Dieser wurde von der Unix Gemeinschaft initiiert. Ein UNC Pfad verwendet doppelte Forward-Slashes ('//') oder doppelte Back-Slashes ('\\') die dem Namen des Computers vorangestellt werden. Der Pfad in diesem Computer (Laufwerke und Verzeichnisse) wird dann durch einzelne Forward-Slashes oder Back-Slashes separiert. Microsoft Windows verwendet generell Back-Slashes, während Unix und Linux Forward-Slashes verwenden.

Unicode
Ein Zeichen Codierungs Schema das 16 Bit (2 Byte) pro Zeichen verwendet um alle Zeichen der wichtigen Weltsprachen (lebende und tote) in einem einzelnen Zeichensatz zusammenführen zu können. Wie beim ASCII Zeichensatz werden die einzelnen Zeichen über Nummern identifiziert. Unicode Zeichensätze können mehr als 65.000 Zeichen enthalten.

Unix
Ein Betriebssystem das zusammen mit Forschern von AT&T entwickelt wurde. Unix ist sehr bekannt für seine Hardware Unabhängigkeit und die portablen Anwendungs Schnittstellen. Unix wurde für die gleichzeitige Verwendung durch viele Personen entwickelt und hat TCP/IP Netzwerkunterstützung eingebaut. Unix und seine Derivate (Linux) sind die gängigsten Betriebssysteme für Server im Internet.

URL (Uniform Resource Locator)
Eine HTTP Adresse die im World Wide Web zum spezifizieren eines bestimmten Ortes oder einer Datei dient. Es ist die einmalige Adresse einer Webseite im Internet.

Usenet
Ein formloses System von Nachrichtenbrettern oder Diskussionsgruppen im Internet. Das Usenet entstand schon vor dem Internet, auch wenn heute das meiste Usenet Material über das Internet verbreitet wird.

VBS (Visual Basic Script)
Eine Programmiersprache die alle Systemfunktionen aktivieren kann, inklusive Starten, verwenden und schließen von anderen Anwendungen (auch ohne die Kenntnisnahme des Anwenders). VBS Programme können in HTML Seiten eingebunden werden und stellen aktive Inhalte über das Internet zur Verfügung. Visual Basic Script Dateien haben die Erweiterung '*.vbs'.

Virus
Ein Computer Programm das die Fähigkeit besitzt auf Laufwerke und andere Dateien zuzugreifen, und sich selbst reproduzieren und verbreiten kann. Üblicherweise erfolgt dies ohne die Kenntnis und Zustimmung des Anwenders. Manche Viren befallen Dateien, so dass diese Dateien bei ihrer Ausführung den Virus mit starten. Andere Viren residieren im Speicher des Computers und infizieren Dateien während diese geöffnet, bearbeitet oder erstellt werden. Manche Viren zeigen Symptome und andere zerstören Dateien und Computer Systeme, aber weder die Symptome noch die Zerstörung sind wichtig für die Definition eines Virus. Auch ein nicht zerstörerischer Virus ist ein Virus.

VM (Virtual Machine)
Eine Software die die Eigenschaften einer Hardware simuliert. Eine virtuelle Maschine ist eine eigenständige Arbeitsumgebung, die sich wie ein separater Computer verhält. Java Applets laufen zum Beispiel in einer Java Virtual Machine, die keinen Zugriff auf das Wirt Betriebssystem hat.

VSync (Vertikal Synchronisation)
Vertikal Synchronisation ist eine Option die in vielen Spielen zu finden ist, hierbei wird die Bildwiederholrate des Spiels an die Bildwiederholrate des Monitors angepasst. Generell sorgt die eingeschaltete VSync für eine bessere Bildstabilität, aber durch abschalten sind höhere Bildwiederholraten möglich. Die Negativseite der höheren Geschwindigkeit ist die Möglichkeit von visuellen Artefakten wie Bildabrisse.

WWC (Wide Web Consortium)
Das regierende Gremium für Standards im World Wide Web.

WAN (Wide Area Network)
Eine Gruppe von Computer Netzwerken die über eine lange Distanz miteinander verbunden sind. Das Internet ist ein WAN.

Word
Ein Integer Typ der zwei Byte im Speicher belegt.

World Wide Web
Ein System zum durchstöbern von Internet Seiten. Es ist nach dem Web (Netz) benannt, da es aus vielen miteinander verknüpften Seiten besteht. Der Anwender kann von einer Seite zur nächsten springen indem er einen Hyperlink betätigt.

Wurm
Ein parasitäres Computer Programm das sich reproduziert, aber anders als Viren, nicht den Computer oder andere Dateien infiziert. Würmer können Kopien auf dem selben Rechner erstellen oder Kopien an andere Rechner in einem Netzwerk versenden.

Wrapper
Eine Programmierschnittstelle zwischen einem Programm und einem separaten eingebundenen Code. Wird hauptsächlich für Kompilierungszwecke benötigt, wenn sich der eingebundene Code zum Beispiel in einer DLL befindet oder eine andere Aufruf Konvention hat. Der Sinn des ganzen liegt darin, dass auf den eingebundenen Code nur über den Wrapper zugegriffen wird.

WSH (Windows Scripting Host)
Ein von Microsoft integriertes Modul mit dem sich Operationen automatisieren lassen ohne den Windows Desktop zu verwenden.

XML (Extensible Markup Language)
Ein Standard zum erstellen von Beschreibungssprachen, die die Struktur von Daten beschreiben. Es gibt keine festgelegten Elemente wie bei HTML, sondern es handelt sich um eine Sprache zum beschreiben von Sprachen. XML ermöglicht dem Autor die Definition von eigenen Tags.

ZIP Datei
Ein Archiv das eine Sammlung von anderen Dateien in komprimierter Form enthält. ZIP Dateien sind sehr beliebt im Internet, da der Anwender viele Dateien in einem Container liefern kann, der zusätzlich noch Plattenplatz und Download Zeit einspart.

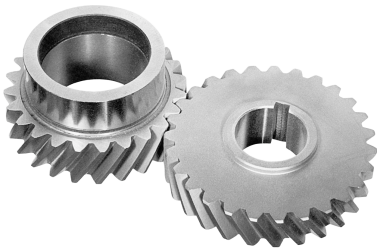
Index

Über den Autor

Gary Willoughby ist ein professioneller Grafik Designer, Web-Entwickler und Software Entwickler. Zum ersten mal biss er sich die Zähne an Programmiersprachen wie PHP, Python und JavaScript aus. Gegen Ende der neunziger Jahre beschloss Gary seine Freizeit in die Anwendungsentwicklung mit Kompilierten Sprachen zu investieren.

Nachdem er eine Weile mit C++ gearbeitet, und gemerkt hatte, welch immenser Aufwand für ein kleines Programm nötig war, begann die Suche nach einer einfacheren, intuitiveren und eleganteren Programmiersprache, um ansprechende Software in kürzerer Zeit entwickeln zu können. All diese Wünsche wurden von einer kleinen französischen Firma mit Namen Fantaisie Software erfüllt. Die Programmiersprache die diese entwickelte und Vertrieb, war PureBasic.

Seit Sommer 2002 ist Gary ein aktives Mitglied im PureBasic Forum (unter dem Pseudonym 'Kale') und hat einige erfolgreiche kommerzielle Programme mit PureBasic erstellt.



PureBasic

Eine Einführung in die Computer Programmierung

PureBasic - Eine Einführung in die Computer Programmierung ist ein essentieller Leitfaden für Einsteiger in PureBasic oder in die Computer Programmierung allgemein. Wenn Sie ein erfahrener Programmierer sind und einfach eine schnelle Einführung in PureBasic suchen oder Sie sind jemand der gerne von Grund auf lernen möchte wie man einen Computer programmiert, dann ist das Ihr Buch. PureBasic ist eine großartige Sprache für den Einstieg in die Computer Programmierung und dieses Buch wurde darauf zugeschnitten, Sie durch die ersten Schritte zu führen. Dieses Buch wurde in keinem albernen Stil geschrieben und hat sich zum Ziel gesetzt auch komplizierte Themen klar und deutlich auf eine hilfreiche Weise zu erklären.

Auszüge aus dem Inhalt:

- Die Geschichte der Programmiersprache PureBasic
- Eine komplette Referenz der Kern Sprachelemente
- Eine Einführung in die Verwendung der Hilfe für die IDE und den Visual Designer
- Richtlinien zum schreiben von gutem Code
- Anleitungen zum erstellen von Grafischen Benutzeroberflächen
- Beispiele für 2D & 3D Grafik
- Ein Bereich in dem fortgeschrittene Themen einfach erklärt werden
- Umfangreicher Anhang mit Internet Links und hilfreichen Tabellen
- Ein vollständiges Computer Sachwortverzeichnis um neue Programmierer zu unterstützen

Der Download enthält die
Code Beispiele des Buches
www.pb-beginners.co.uk (englisch)
asw.gmxhome.de (deutsch)

Gary Willoughby

RRP £0,00 - $0,00 - €0,00