PocketC 5

- Benutzerhandbuch und Referenz -

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.

Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt- Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalft vorgegangen.

Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Hersteller, Herausgeber, Autoren und Übersetzer können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Für Verbesserungsvorschlage sind Hersteller und Übersetzer dankbar.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Fachliche Beratung: Jeremy Dewey, Orbworks Inc. – http://www.orbworks.com

Übersetzung: Carsten Bollenbach – http://www.brainrunner.de

Produced in Germany

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Vorbemerkung zur Übersetzung

Die Übersetzung ist sehr eng an dem englischen Original angelehnt. Die Struktur sowie deren Inhalte wurden nahezu 1:1 in die deutsche Sprache übersetzt. Wo eine exakte Übersetzung nicht möglich oder sinnvoll war, wurde eine freie Übersetzung vorgenommen.

Abweichend von dem Original ist ein Inhaltsverzeichnis sowie das Kapitel 1 hinzugekommen. Die Idee war es, die "losen" HTML Dokumente von Orbworks in einem praktischen Buch zusammenzuführen. Meiner Ansicht nach erleichtert ein gedrucktes Buch wesentlich den Lernprozess, da man nicht ständig zwischen PocketC und einem Browser wechseln muß. Außerdem kann man in einem Buch Anmerkungen und Kommentare niederschreiben.

Die Gelegenheit möchte ich nutzen, um mich bei ein paar Leuten zu bedanken, ohne die ich diese Übersetzung nicht hätte fertigstellen können. Da ist erst einmal Jeremey Dewey von Orbworks, der geduldig erklärte was für mich auf den ersten Blick nicht verständlich genug war. Dann sind da noch Marc, Conny und Uwe - nichts ist perfekt, Jungs - deutlicher ging es kaum. Und schliesslich ist da noch meine Frau Gerda, die ebenfalls als Lektorin das hier abgeklopft hat. Ausserdem war sie mehr als geduldig mit mir - dafür (und nicht nur dafür) liebe ich sie sehr.

Natürlich bin ich (wie jeder andere Autor auch) dankbar für Anregungen und Kritiken. Daher möchte ich Sie herzlich einladen, mir zu schreiben. Und sehen Sie mir bitte den einen oder anderen Flüchtigkeitsfehler nach, schreiben Sie mir lieber – ich setze das denn auch sofort um; versprochen ;)

Schreiben Sie einfach eine Mail an "cbo@brainrunner.de"

Und nun wünsche ich Ihnen viel Freude und Erfolg mit PocketC!

Ihr

Carsten Bollenbach

"ICH LIEBE ABGABETERMINE. ICH MAG DIESES RAUSCHENDE GERÄUSCH WENN SIE AN EINEM VORBEIFLIEGEN."
- DOUGLAS ADAMS

1 Vorwort

1.1 An wen richtet sich dieses Buch?

Dieses Buch wendet sich an Programmierer, die mit der Sprache "C" vertraut sind und nun Projekte mit PocketC auf der Palm-OS Plattform realisieren wollen.

Aber auch Einsteiger werden nützliche Informationen finden. In der Regel sollten Sie keine großen Probleme haben, die ersten Programme mit Hilfe dieses Buches zu entwickeln. Einige Sachverhalte setzen wir als gegeben voraus. So sollten Sie mit der Bedienung Ihres PC's, Ihrem PDA und mit grundlegenden Programmiertechniken vertraut sein.

Daher empfehlen wir Ihnen, das Sie begleitend zusätzliche Lektüre hinzuziehen, so Ihnen wichtige Grundlagen fehlen sollten.

1.2 Wie dieses Buch aufgebaut ist

Damit Sie sich schneller in diesem Buch zurecht finden, zeigen wir Ihnen nun, was Sie im Folgenden erwartet.

1.2.1 Überblick

Dieses Kapitel wendet sich an alle Leser. Sie erfahren hier die wichtigsten Informationen, die Sie zum entwickeln von PocketC Programmen benötigen. Gleich ob Sie gerade mit der Programmierung beginnen oder sehr erfahren sind; dieses Kapitel zeigt Ihnen wie Sie den Compiler starten, wie sie einzelne Programme starten, sichern oder löschen. Außerdem stellen wir Ihnen hier das Grundgerüst eines PocketC Programmes vor.

1.2.2 Einführung in PocketC

Wir stellen hier die Sprache PocketC näher vor. Dazu gehört ein Abriss zum Thema "Variablen" und Funktionen, genau so wie eine grundlegende Erläuterung von "Ausdrücken" und "Operatoren. Abgerundet wird dieses Kapitel durch die Beschreibung von "Kontrollstrukturen" und dem fortgeschrittenen Thema "Zeiger" und "Zeigerarithmentik".

1.2.3 Sprachreferenz

Während Sie Programme entwickeln, werden Sie dieses Kapitel häufig benutzen. Wir haben es bewußt als Kapitel zum nachschlagen konzipiert. Es werden alle Elemente der PocketC Sprache vorgestellt.

1.2.4 Bibliotheksreferenz

Während Sie Programme entwickeln, werden Sie dieses Kapitel häufig benutzen. Wir haben es bewußt als Kapitel zum nachschlagen konzipiert. Es werden alle Standardfunktionen der PocketC Sprache vorgestellt.

1.3 Typographische Konventionen

Um die Lesbarkeit dieses Buches für Sie zu erhöhen, haben wir den Text nach ein paar wenigen Regeln gestaltet. So wissen Sie stets, wie Sie die entsprechenden Textpassagen zu lesen haben.

Kursivschrift	Verwenden wir für Dateinamen (sowie deren Endungen), URL's, sowie neue Begriffe
<KNOPF>	Der Name eines Knopfes auf dem Display, durch dessen Antippen Sie eine bestimmte Funktion / Aktion auslösen
"MENÜ"	Der Name eines Menüs bzw. einer Menüauswahl. Der volle Pfad zu der benötigten Menüauswahl wird durch Schrägstriche getrennt (z.B. "APPLET"/"BACKUP")
`Courier`	Verwenden wir für Einzelbefehle, Codebeispiele, Variablen, Konstanten und Direktiven
`Courier in Kursiv`	Verwenden wir für variable Elemente, wie zum Beispiel Parameterlisten nach Befehlen
`[Courier in Kursiv]`	Variable Elemente, welche wir in eckige Klammern stellen, sind optional
`"Courier"`	Ausgaben, besondere Zeichen etc. schließen wir mit Anführungszeichen ein.

1.4 Neues in der Version 5.0.2

Im Dezember 2002 erschien die neue Version 5.0.2 von PocketC. Die Änderungen bezogen sich in erster Linie auf eine bessere, interne Speicherverwaltung sowie überarbeitete Editierfunktionen.

Die grundlegenden Funktionalitäten wurden hiervon nicht berührt, so dass das vorliegende Handbuch zwar auf Basis von 4.4.1 geschrieben wurde, jedoch ohne weiteres auch für die neuere Version Gültigkeit hat. Jeremey hat hierauf ausdrücklich hingewiesen, so das Sie sich keine weitere Gedanken darüber machen brauchen.

2 Überblick

2.1 Einleitung

PocketC ist ein Compiler, der auf Palm-OS basiert. Quelltexte, die mit dem Notizbuch Ihres PDA's erstellt worden sind, werden mit PocketC übersetzt und auch ausgeführt. Alternativ können Sie auch ein Run-Time Modul installieren, wenn Sie Ihre Programme auf dem PC entwickeln wollen. In der aktuellen Version ist es noch nicht möglich, Applikationen ohne dieses Modul auf Ihrem PDA auszuführen. Der Compiler bzw. das Run-Time Module muß in jedem Fall installiert sein.

Die Version 5 (mit der dieses Handbuch ausgeliefert wurde) ist Shareware. Sie können diese Software unter *http://www.orbworks.com/pcpalm/buy.html* erwerben und registrieren. Abgesehen von dem guten Gefühl, eine gültige und legale Lizenz zu besitzen, haben Sie noch weitere Vorteile. So erhalten Sie zum Beispiel "BuildPRC", welches Ihnen ermöglicht Ihre Programme direkt vom Launcher und nicht erst über den Umweg "Compiler" aus zu starten. Besuchen sie uns (*http://www.orbworks.com*) um mehr Informationen zu erhalten.

Zusätzlich zu dem Compiler auf Ihrem PDA empfehlen wir Ihnen die "PocketC Desktop Edition", mit der Sie auf Ihrem PC Programme entwickeln können. Diese Suite ist auch in der Lage, ausführbaren Code zu erzeugen. Sie (und hoffentlich vielen anderen Benutzer Ihrer Software) werden dann das "PocketC Run-Time" nicht mehr auf Ihrem PDA installieren müssen.

2.2 Installation eines Programmes

Sie installieren Ihre entwickelten Programme genauso, wie Sie fertige Software (wie den PocketC Compiler) auf Ihrem PDA installieren. Benutzen Sie das "Palm Install" Werkzeug, wählen Sie die entsprechende *.prc Datei(-en) aus und starten Sie den Synchronisationsvorgang.

2.3 Installation von Quelltext

Ihren Quelltext können Sie mit jedem beliebigen Editor auf dem PC erstellen. Erzeugen Sie anschließend eine neue Notiz mit dem "Palm Desktop" und kopieren Sie den fertigen Quelltext dort hinein. Anschließend führen Sie einen Hot-Sync durch.

2.4 Schreiben eines Programmes

Damit PocketC Ihre Quelltexte erkennen kann, müssen Sie Ihre Programme mit einer Kopfzeile versehen.

Ein Quelltext wird auf dem Palm entweder als Notiz oder als **.DOC* Datei erstellt. Dabei muß die erste Zeile von zwei Schrägstrichen "`//`" eingeleitet werden, zusammen mit dem Namen Ihres Programmes. Falls Ihr Quelltext eine **.DOC* Datei sein sollte, dann muß diese Zeile mit **.C* bzw. **.PC* enden, damit er in dem "Compile" Dialog angezeigt wird.

Beispiel für eine Kopfzeile, erstellt mit dem Notizbuch:

```
// Mein Programm
```

Beispiel für eine Kopfzeile, erstellt mit einem Editor wie zum Beispiel QED

```
// Mein Programm.c
```

oder

```
// Mein Programm.pc
```

Selbstverständlich können Sie auch Ihre Quelltexte zunächst auf Ihrem PC erstellen und anschließend auf Ihren PDA übertragen.

2.5 Kompilieren eines Programmes

Tippen Sie auf den <COMPILE...> Knopf des PocketC Compilers. Sie erhalten nun eine Auswahl von Quelltexten, die Sie übersetzen lassen können. Wählen Sie einen aus und tippen Sie erneut auf <COMPILE...>.

Ihr Programm wird nun kompiliert. Sollten dabei Fehler auftreten, so werden Sie mit entsprechenden Fehlermeldungen über die Art und die Position des Fehlers informiert.

2.6 Ausführen eines Programmes

Um ein Programm auszuführen, starten Sie PocketC (oder das Run-Time Modul). Wählen Sie nun eines Ihrer Programme aus der Liste der kompilierten Programme aus und tippen Sie auf <EXECUTE> ("Ausführen").

Nun wird die Ausgabeansicht geöffnet und Ihr Programm ausgeführt. Sobald das Programm beendet wird (entweder weil das Programmende erreicht wurde oder Sie zur Laufzeit auf <DONE> getippt haben), erscheint die Hauptansicht wieder.

Um wieder auf die Ausgabeansicht umzuschalten, tippen Sie einfach auf <OUTPUT>.

2.7 Sichern eines Programmes

Indem Sie Ihr Programm sichern, können Sie die ausführbare Datei weitergeben, der Quelltext jedoch verbleibt bei Ihnen. Mit anderen Worten: wenn Sie ein Backup ausführen, wird das Programm (nicht der Quelltext) in das Backup Verzeichnis des Palmdesktops übertragen.

Um das Backup durchzuführen, wählen Sie das betreffende Programm in der Hauptansicht aus und tippen auf "APPLET"/"BACKUP".

Ihr Programm wird bei dem nächsten Hot-Sync gesichert.

2.8 Löschen einer Datenbank

Einige Ihrer Programme werden Datenbanken erzeugen um zum Beispiel Voreinstellungen oder andere nützliche Daten zu speichern. Wenn Sie jedoch Ihr Programm wieder löschen wollen, möchten Sie auch die dazugehörende Datenbank löschen.

Dazu wählen Sie "APPLET" / "DATABASES". Nun werden Ihnen die Datenbanken angezeigt, die Ihre Programme angelegt haben. Wählen Sie nun die zu dem betreffenden Programm gehörende Datenbank aus und tippen Sie auf <DELETE>. Die Datenbank wird nun gelöscht.

2.9 Benutzen von "**.DOC*" Dateien

Möglicherweise möchten Sie einen anderen Editor als das Notizbuch auf Ihrem PDA benutzen, um Ihre Programme zu entwickeln. Dazu bieten sich unter anderem Programme wie QED an. Diese Editoren erzeugen sogenannte **.doc* Dateien, welche Sie vielleicht auch als eBook kennen. Der Vorteil dieser Editoren liegt in der wesentlich komforTabelleren Art des Schreibens.

Um solche Editoren nutzen zu können, müssen Sie die Kopfzeile Ihres Programmes nicht nur mit zwei Schrägstrichen einleiten, sondern auch entweder mit der Endung **.c* oder **.pc* abschließen.

Beispiel für eine Kopfzeile, erstellt mit einem Editor wie zum Beispiel QED

```
// Mein Programm.c
```

oder

```
// Mein Programm.pc
```

Nur so sind diese Quelltexte für PocketC als solche erkenn- und damit kompilierbar.

Dabei muß der Dateiname selbst nicht mit der Kopfzeile identisch sein. Dies gibt Ihnen zusätzliche Möglichkeiten, beispielsweise bei der Versionskontrolle Ihrer Entwicklungsarbeit.

Programme, die Sie per `#include` in Ihren Quelltext einbinden wollen, müssen sich nicht dieser Regel unterwerfen.

Einige Editoren unterstützen sogenannte *Lesezeichen* (auch bekannt als *Bookmarks* oder *Favoriten*). Dabei wird eine Zeile z.B. mit einem Wort eingeleitet. Am Ende des Quelltextes befindet sich zum Abschluß eben dieses Wort, eingeschlossen von eckigen Klammern.

Um Lesezeichen unter PocketC zu benutzen, muß als Zeichen das umgekehrte Apostroph (`) definiert werden. Diese Funktion ist besonders Nützlich um schnell zu Funktionen in Ihrem Quelltext springen zu können.

Grundsätzlich muß die unkomprimierte Dateigröße einer **.doc* Datei als PocketC Quelltext unter 64 Kilobytes bleiben.

3 Einführung in PocketC

3.1 Übersicht

Zunächst sei bemerkt, das es sich um PocketC um eine Sprache handelt, welche Groß- und Kleinschreibung berücksichtigt (*Case Sensitive*). Das bedeutet, das der Kompiler zwischen `WORT`, `Wort` und `wort` unterscheidet. Achten Sie darauf, wenn Sie Ihr Programm entwickeln, dies ist eine Fehlerquelle, die nur schwer zu finden ist [Anm.d.Ü.].

Ein Programm unter PocketC hat drei grundlegende Elemente:

- eine Kopfzeile
- einen Bereich, in dem Sie global gültige Variablen festlegen
- Funktionen

Ich gehe im Folgenden davon aus, das Sie Ihre Programme mit dem Notizbuch Ihres PDA's erstellen [Anm.d.Ü.]

3.2 Die Kopfzeile

Dies ist der weitaus einfachste Teil der Sprache. Die erste Zeile Ihrer Programmes besteht aus zwei Schrägstrichen, gefolgt von dem Namen Ihres Programmes.

Beispiel für eine Kopfzeile

```
// Mein Programm
```

Eine solche Zeile ist auch als *Kommentarzeile* bekannt. Jedesmal, wenn der Kompiler `//` in Ihrem Programm findet, wird er den Rest dieser Zeile ignorieren. Dies gibt Ihnen die Möglichkeit, Kommentare in Ihr Programm einzufügen.

Es gibt eine weitere Möglichkeit, Kommentare einzufügen: indem Sie diesen mit `/*` einleiten und mit `*/` abschließen. So können Sie einen Kommentar über mehrere Zeilen einfügen.

Beispiel für einen mehrzeiligen Kommentar

```
/* Dies ist ein mehrzeiliger
Kommentar. Alles zwischen den
Sternchen wird vom
Compiler ignoriert */
```

Allerdings ist es nicht möglich Kommentare zu verschachteln.

Anders ausgedrückt …

Beispiel für einen unzulässigen Kommentar

```
/* Kommentar 1 /* Kommentar 2 */ a=b+c: */
```

oder deutlicher

```
/* Kommentar 1
/* Kommentar 2 */
a=b+c: */
```

3.3 Globale Variablen

Variablen sind Dinge, in denen Sie Werte (Zahlen, Zeichen etc. [Anm.d.Ü.]) Ihres Programmes zur Laufzeit speichern. PocketC bietet Ihnen vier Variablentypen an:

Typ	Name	Beispiel
Integer (32-Bit, mit Vorzeichen)	`int`	`1, 2, 5, -789, 452349`
Fließkomma(32-Bit)	`float`	`-1.2, 3.141592, 5.7e-4`
Einzelzeichen (8-Bit, mit Vorzeichen)	`char`	`'a', 'b', '#'`
Zeichenkette	`string`	`"Bob", "Katie", "Hello"`
Zeiger	`pointer`	wird später erläutert

Tabelle 1: Datentypen in PocketC

Ein Wort noch zu der Zeichenkette: diese dürfen höchsten 255 Einzelzeichen lang sein. Um eine längere Zeichenkette in einer Variable abzulegen, müssen Sie diese miteinander verknüpfen.

Beispiel für eine überlange Zeichenkette in einer Variable:

```
str = "[Zeichenkette bis 255 Zeichen]" + "[Zeichenkette bis 255 Zeichen]";
```

Sie deklarieren Variablen, indem Sie den Datentypen, gefolgt von den Variablennamen angeben.

```
daten-typ variable[, variable…];
```

Beispiel für eine globale Variablendeklaration:

```
int myInteger, row, column;
string name;
float pi,
char c, last, first;
pointer ptr;
```

Genauso ist es möglich Felder (sogenannte *Arrays*) von Werten zu deklarieren. Ein Feld ist eine Liste von Werten, die in einer einzigen Variable gespeichert werden. Felder werden wie gewöhnliche Variablen deklariert, jedoch mit dem Unterschied das nach dem Variablenname noch mit der Feldgröße abgeschlossen wird. Die Feldgröße bezeichnet die Anzahl der Werte, welche die Variable aufnehmen kann. Natürlich können Variablen und Felder vom gleichen Datentyp auch gemeinsam in einer Zeile deklariert werden.

Dabei dürfen Sie natürlich auch schon Vorgabewerte definieren. Schauen wir uns die nächsten Beispiele genauer an:

Beispiel für eine einfache Felddeklaration:

```
/* Ein Feld mit möglichen zehn Integerwerten
int values[10];
/* Ein Feld mit möglichen sieben Zeichenketten
string names[7];
```

Beispiel für eine gemischte Deklaration mit gleichem Datentyp

```
int row, values[10], column;
string name, colors[8];
```

Beispiel für eine einfache Felddeklaration mit Vorgabewerten:

```
/* 'zero' hat keinen Vorgabewert
int nine = 9, eight = 8, zero;
string days[7] = {"Sun", "Mon", "Tues"};
```

Im Falle von Arrays müssen Vorgabewerte in geschweifte Klammern gesetzt werden, sowie untereinander durch Kommata getrennt.

Falls dabei die Anzahl der Vorgabewerte geringer ist, als die deklarierte Feldgröße so werden die nicht belegten Elemente mit einem Vorgabewert belegt. Was die Wochentage in dem Beispiel angeht, so sind die letzten vier Tage eine leere Zeichenkette (`''`).

Wir werden uns im Abschnitt "3.6 Zuweisungen" mit Variablen später eingehender beschäftigen.

3.4 Funktionen

Funktionen enthalten konkrete Anweisungen die ein Programm erst zu einem Programm machen. Jede Funktion hat einen Namen und eine Parameterliste (die auch leer sein kann).

Deklariert werden sie wie folgt:

```
funktions-name([parameter-typ parameter-name, …]) { weitere Anweisungen}
```

Wir beschäftigen uns später mit Anweisungen. Zunächst ein paar Beispiele für Funktionen:

Beispiel für eine Funktion:

```
area (int width, int height){
   return width * height;
}

square (float x) {
   return x * x;
}

five() {
   return 5;
}
```

Es gibt eine ganz besondere Funktion, die jedes Programm haben muß: `main()`. Die `main()` Funktion wird vor allen anderen als erste aufgerufen. Wenn die `main()` Funktion beendet wird, wird das Programm selbst beendet. `main()` wird ohne Parameter deklariert.

```
// Mein Programm
main() {
  puts ("Hello World");
}
```

Funktionen können auch lokale Variablen haben, welche nur innerhalb der Funktion genutzt werden können in der sie deklariert wurden. Anders ausgedrückt: Sie können nicht von "aussen" auf diese Variablen zugreifen [Anm.d.Ü.].

Anders verhält es sich mit globalen Variablen. Sie können auf diese von jeder Stelle des Programmes zugreifen. Es ist dabei unerheblich ob Sie auf diese Variablen aus der deklarierenden Funktion selbst, oder aus einer anderen Funktion her zugreifen wollen (Anm.d.Ü.).

Lokale Variablen werden genauso wie globale Variablen deklariert, mit dem Unterschied das sie sofort der ersten geschweiften Klammer der betreffenden Funktion folgen.

Beispiel für eine lokale Variablendeklaration:

```
// Mein Programm

main () {

   string localstring;


   localstring = "Hello World";

   puts(localstring);

}
```

Eine Anmerkung: wenn Sie große Arrays deklarieren wollen, so ist es am besten dies global zu tun. In diesem Fall vermeiden Sie bitte nach Möglichkeit die lokale Deklaration .

Bevor wir dieses Thema vertiefen, sollten wir uns ein wenig über Ausdrücke unterhalten.

3.5 Ausdrücke

Ein Ausdruck ist eine Abfolge von Konstanten, Variablen und Funktionsaufrufen die mittels Operatoren, Klammern und Anführungszeichen miteinander verbunden werden.

Dabei ist eine Konstante ein beliebiger Wert, der direkt in das Programm eingefügt wird. Dieser Vorgang ist auch als "*Harte Kodierung*" bzw. "*Hard Coded*" bekannt (Anm.d.Ü.).

Beispiele für eine Konstante: `5` `5.4` `'a'` `"Zeichenkette"`

Auf einen Wert innerhalb einer Variable können Sie zugreifen, indem Sie ihn über den Variablennamen ansprechen. Falls diese Variable ein Array sein sollte, dann muß der jeweilige Wert über seinen Index angesprochen werden. Gültige Indizes für einen Array beginnen bei `0` und enden bei `n-1`, wobei n die Anzahl der Werte in einem Array angibt.

Beispiele für einen gezielten Zugriff auf einen Array:

```
/* Wir deklarieren ein Array */

string names[4];

/* Wir füllen diesen nun mit Werten, indem

wir einzeln auf die Elemente zugreifen */

names[0] ="first name";

names[1] = "second name";

names[2] = "third name";

names[3] = "fourth name";
```

Ein Funktionsaufruf kann nur stattfinden nachdem die Funktion selbst definiert worden ist. Falls Sie jedoch eine Funktion aufrufen wollen bevor sie definiert worden ist, können Sie sogenannte Prototypen verwenden.

Dabei handelt es sich um eine Programmzeile (welche sich nicht innerhalb einer anderen Funktion befinden darf), welche den Namen der Funktion sowie deren Parameterliste beinhaltet. Diese wird von einem Semikolon abgeschlossen.

Beispiele für die definition von Prototypen:

```
area (int x, int y);

square (float); // Die Benutzung von Variablennamen ist optional in einer
Deklaration
```

Diese drei Elemente lassen sich nun durch Operatoren miteinander verbinden:

Beispiele für einen gezielten Zugriff auf einen Array:

```
5 + 7 - area(12, 34);

square(5) * pi;

"Hello, " + "World";
```

Natürlich können Funktionsaufrufe mit Ausdrücken durchgeführt werden:

Beispiele für einen Funktionsaufruf mit eingebetteten Ausdrücken:

```
area (6+3, 8*9);

area (8 * square(4), 7); // Natürlich können Sie auch einen Ausdruck mit
                            einer Funktion selbst verknüpfen
```

3.6 Zuweisungen

Eine Zuweisung ist im Grunde nichts anderes als eine andere Form eines Ausdrucks. Sie erfolgt in der Regel auf zwei Arten.

Zum Einen für eine normale Variable:

```
name = ausdruck
```

Zum Anderen für ein Array:

```
name[index] = ausdruck
```

Beispiele für Zuweisungen:

```
int myInt, numbers[3];
string mystring;
…
myInt = 8;
myString = "Animaniacs";
numbers [0] = myInt + 5;
numbers[2] = numbers[0] * 8;
```

Und weil PocketC sehr tolerant Datentypen umgeht, können Sie Variablen des einen Datentyps mit einem Wert eines völlig anderen Datentyps belegen. Der Wert wird automatisch konvertiert.

Beispiele für Zuweisungen mit gemischten Datentypen:

```
myString = 95;       // Der Wert ist nun "95"
numbers[1] = "78";         // Der Wert ist nun 78
numbers["2"] = "2";        // Ein hübscher Trick: numbers [2] ist nun 2
```

Welche Operatoren gibt es nun und welche Priorität bzw. Assoziativität haben sie? Gute Frage, lesen Sie einfach weiter.

3.7 Operatoren

Die nun folgende Tabelle gibt Ihnen einen Überblick über die gültigen Operatoren. Sie ist sortiert nach Priorität, die niedrigste zuerst.

Anm.d.Ü.: Priorität bedeutet in diesem Zusammenhang die Behandlung des jeweiligen Operators durch den Compiler. Vereinfacht ausgedrückt hat er es hier mit einer komplexeren Form des bekannten Schulspruchs "Punktrechnung geht vor Strichrechnung" zu tun. In diesem Beispiel hat die Strichrechnung eine niedrigere Priorität, und wird als zweites kompiliert. In einer Liste wie der folgenden wäre das Minuszeichen an erster Stelle, gefolgt von dem Pluszeichen. Die Assoziation bedeutet, auf welche Seite der Compiler zuerst hinschaut und arbeitet.

Operator	Assoziation	Beschreibung
=	rechts	Wertet zunächst den Ausdruck auf der rechten Seite des Operators aus. Anschließend wird der ermittelte Wert der Variable zu seiner linken zugewiesen.
||	links	logisches "ODER", liefert den Wert 0 falls der Vergleich unwahr ist, 1 falls er wahr sein sollte
&&	links	logisches "UND"
|	links	bitweises "ODER"
^	links	bitweises "XODER"
&	links	bitweises "UND"
== != < <= >= >	links	vergleichende Operatoren: • gleich • ungleich • kleiner als • kleiner oder gleich • größer oder gleich • größer Liefert den Wert 0, falls der Vergleich unwahr ist, den Wert 1 falls er Wahr ist.
<< >>	links	bitweises verschieben von Werten. Die Operanden müssen entweder vom Datentyp int oder char sein.
+ -	links	• addition (möglich auch mit Einzelzeichen bzw. Zeichenketten) • subtraktion (nur möglichen mit Zahlenwerten)
* / %	links	• multiplikation • division • modulus Der Modulus kann weder mit Zeichen, noch mit Fliesskommawerten verwendet werden.
- ! ++ - - ~ [ ] ( ) &		• Negation (Umkehrung) • logisches "NICHT" • Inkrementierung (Werterhöhung) • Dekrementierung (Wertminderung) • bitweises negieren (bitweise das Vorzeichen ändern) • Array Index • Funktions Parameterliste

Operator	Assoziation	Beschreibung
`@[]`		• Addressen Referenz • Zugriff auf ein Einzelzeichen einer Zeichenkette Bis auf das logische "NICHT" und den Zugriff auf ein Einzelzeichen einer Zeichenkette können Sie diese Operatoren mit Zeichenketten als Operanden verwenden.

Tabelle 2: Operatoren in PocketC

Eine Bemerkung: es wird keine abgekürzte Logik auf die Operatoren `||` und `&&` angewendet. Gleichsam werden die gemischten Operatoren (wie z.B. `+=, *=`), das Komma sowie die bedingten Operatoren (`?:`) nicht unterstützt.

3.8 String Zugriffsmethoden

Um auf ein Einzelzeichen innerhalb einer String Variable zuzugreifen (oder es aber auch zu setzen) werden Sie eine besondere Form eines Ausdruck benutzen.

```
stringVariable@ [index]
```

Die Position bzw. der Index des ersten Zeichens ist 0. Wenn Sie also auf das dritte Element einer Zeichenkette zugreifen wollen, so müssen Sie zuvor "eins" von der gewünschten Position abziehen - in diesem Fall wäre dann der gewünschte Index nicht "3" sondern "2".

Beispiele für Stringzugriffe:

```
string str = "bob";

...

puts (str@[1]);     // Gibt das zweite Zeichen der definierten Zeichenkette
                       zurück

str@[1] = 'X';      // Ändert "bob" in "bXb"
```

Gehen Sie vorsichtig mit diesem Operator um: Sie werden Laufzeitfehler (im Günstigsten Fall das vorzeitige Beenden Ihres Programmes, Anm.d.Ü.) produzieren, wenn Sie versuchen auf eine Position zuzugreifen, die größer ist als die tatsächliche Länge der Zeichenkette.

Eine Bemerkung: der Stringzugriffsoperator kann nicht mit Zeigern benutzt werden. Auch kann so nicht die Adresse der Zeichenkette ermittelt werden. Mit anderen Worten, Ausdrücke wie `&str@[i], *pstr@[i]` und `(*pstr)@[i]` sind nicht möglich.

3.9 Schrittweise Inkrementierung bzw. Dekrementierung

Die Operatoren ++ und -- haben eine besondere Wirkung, abhängig davon ob sie direkt vor oder direkt nach einen Variablennamen gesetzt werden.

Wenn Sie einen dieser Operatoren vor einen Variablennamen setzen, liefert der Ausdruck den neuen Wert der Variable *nachdem* er erhöht wurde.

Wenn Sie einen dieser Operatoren nach einem Variablennamen setzen, liefert der Ausdruck den neuen Wert der Variable *bevor* er erhöht wird.

Beispiele für schrittweise Inkrementierung bzw. Dekrementierung:

```
int myInt;

...

myInt = 8;

puts (++myInt);           // Gibt "9" aus


myInt = 8;

puts (myInt++);           // Gib "8" aus und weist myInt jedoch den Wert "9"
                             zu
```

3.10 Automatische Datentypumwandlung ("*Casting*")

Genauso wie in Zuweisungen findet eine automatische Datentypumwandlung auch in jedem Teil eines Ausdrucks statt. Falls zwei Operanden eines Operators unterschiedlichen Typs sind, so wird einer davon dem weniger formaleren Typ angepasst.

Dabei wird von Integer nach Zeichen nach Fliesskomma nach Zeichenkette umgewandelt.

Also wird in dem Ausdruck `"Das Ergebnis ist: " + 5;` zunächst die Konstante 5 in ein Zeichen umgewandelt um anschließend mit `"Das Ergebnis ist: "` verknüpft zu werden. Das Resultat ist also `"Das Ergebnis ist 5"`.

Dies kann unter Umständen ungewünschte Seiteneffekte nach sich ziehen. Wenn Sie zum Beispiel einen Ausdruck zusammen mit dem entsprechenden Ergebnis ausgeben wollen, so würden Sie dies vielleicht auf die folgende Art tun:

```
puts ("5 + 7" + 5 + 7);              // So bitte nicht!
```

Das Ergebnis wäre folgende Ausgabe:

```
"5 + 7 = 57"
```

Das ist vermutlich nicht das, was Sie beabsichtigen. Statt dessen wollen Sie, das der Compiler zunächst den Ausdruck selbst berechnet um das Ergebnis dann mit der Zeichenkette zu verknüpfen.

Benutzen Sie Klammern, um das gewünschte Resultat zu erzielen:

```
puts ("5 + 7" + (5+7));              // Gibt "5 + 7 = 12" aus
```

Allerdings haben wir immer noch ein kleines Problem. Nehmen wir an, Sie wollen den Fliesskommawert eines Teiles von zwei Integerzahlen ermitteln:

```
puts ("7 / 5 = " + (7 / 5);          // Gibt "7 / 5 = 1" aus
```

Dies kommt daher, weil Sie mit zwei ganzen Zahlen arbeiten, konsequenterweise ist das Ergebnis auch eine ganze Zahl. Um dies zum umgehen, wandeln wir nun eine ganze Zahl in eine Fliesskommazahl um:

```
puts ("7 / 5 = " + ((float) 7 / 5)); // Gibt "7 / 5 = 1.4" aus
```

Sie zwingen nun "7" nicht mehr eine ganze Zahl, sondern vielmehr eine Fliesskommazahl zu sein. Und zwar bevor sie durch fünf dividiert wird.

3.11 Anweisungen

Anweisungen sind einzelne Bestandteile, welche eine Funktion bilden. Die folgende Tabelle listet alle auf:

Anweisung	Beschriebung
`return;`	Flusskontrolle: Springt sofort aus der aktuellen Funktion in diejenige zurück, von der sie aufgerufen wurde. Dabei wird eine ganze Zahl "*0*" zurückgegeben.
`return expr;`	Flusskontrolle: Springt sofort aus der aktuellen Funktion in diejenige zurück, von der sie aufgerufen wurde. Dabei wird eine ganze Zahl *expr* zurückgegeben.
`if (expr) stmt`	Kontrollstruktur: Interpretiert den Ausdruck *expr* und führt *stmt* aus, wenn das Ergebnis "Wahr" (nicht-null oder eine gefüllte Zeichenkette) ist. Ist das Ergebnis "Falsch" wird *stmt* übersprungen, und Ihr Programm wird fortgesetzt.
`if (expr) stmtA else stmtB`	Kontrollstruktur: Interpretiert den Ausdruck *expr* und führt *stmtA* aus, wenn das Ergebnis "Wahr" (nicht-null oder eine gefüllte Zeichenkette) ist. Ist das Ergebnis "Falsch" wird *stmstA* übersprungen, und *stmtB* wird an seiner Stelle ausgeführt.
`while (expr) stmt`	Schleifenstruktur: Interpretiert den Ausdruck *expr* und führt *stmt* solange aus, wie das Ergebnis "Wahr" (nicht-null oder eine gefüllte Zeichenkette) ist. Ist das Ergebnis "Falsch" wird *stmst* übersprungen, und Ihr Programm wird fortgesetzt.
`do stmt` `while (expr)`	Schleifenstruktur: Genauso wie `while`, jedoch mit dem Unterschied das *stmt* zuvor mindestens einmal ausgeführt wird, bevor *expr* interpretiert wird.
`for (init; cond; iter) stmt;`	Schleifenstruktur: *init* ist gleichermaßen der Ausgangswert der Schleife wie ihr Zähler. Die Bedinung *cond* legt das Abbruchkriterium der Schleife fest. Ist es nicht erreicht, wird der Zähler um das Inkrement *iter* erhöht und *stmt* ausgeführt. Dies wiederholt sich solange, bis *cond* erreicht ist. Der ursprüngliche Wert von *init* wird jedoch nur einmal ausgewertet.
`break;`	Flusskontrolle: Springt sofort aus Schleifen wie `while, do, for` oder Kontrollstrukturen mit `switch`.
`continue;`	Startet erneut mit den Anweisungen innerhalb Schleifen wie `while, do, for` oder Kontrollstrukturen mit `switch`.
`switch (expr) {stmt}`	Kontrollstruktur: Interpretiert den Ausdruck *expr* und führt *stmt* aus, wenn das Ergebnis "Wahr" (nicht-null oder eine gefüllte Zeichenkette) ist. *expr* darf keine

Anweisung	Beschriebung
	Fliesskommazahl sein. Falls *stmt* eine `case` Anweisung beinhaltet, dessen Konstante dem Wert *expr* entspricht, so werden die Anweisungen ausgeführt bis ein `break` gefunden oder das Ende des `switch` erreicht wird. Falls es kein `case` gibt, dafür jedoch ein `default`, so werden die Anweisungen nach dem `default` ausgeführt, solange bis ein `break` gefunden oder das Ende des `switch` erreicht wird. Falls es weder ein `case` noch ein `default` gibt, so werden alle Anweisungen des `switch` übersprungen.
`case constant:`	Eine Bedingung innerhalb einer `switch` Anweisung. Die Konstante muss entweder ein Einzelzeichen (`case 'a':`), eine ganze Zahl (`case 3:`) oder eine Zeichenkette (`case "Apple":`) sein. Falls die Konstante dem Wert entspricht die als Ausdruck im `switch` definiert wurde, so werden alle folgenden Anweisungen solange ausgeführt bis ein `break` gefunden oder das Ende des `switch` erreicht wird.
`default:`	Eine optionale Bedingung innerhalb einer `switch` Anweisung. Falls keine der `case` Anweisungen zutraf, so werden alle folgenden Anweisungen solange ausgeführt bis ein `break` gefunden oder das Ende des `switch` erreicht wird.
`{stmt}`	Eine geschweifte Klammer, gefolgt von einer (oder mehreren Anweisungen), wiederum abgeschlossen von einer geschweiften Klammer wird vom Compiler als einzelne Anweisung interpretiert.
`expr;`	Ein Ausdruck, der in einer Zeile mit einem Semikolon abgeschlossen wurde, wird ebenfalls vom Compiler als Anweisung interpretiert.

Tabelle 3: Anweisungen und Kontrollstrukturen in PocketC

Schauen wir uns im Folgenden ein paar Beispiele an, welche das eben Erläuterte ein wenig illustrieren.

3.11.1 RETURN

```
five() {
      return 5;
}
```

Weil der Rückgabewert immer fünf ist, können wir diese Funktion an jeder Stelle unseres Progammes einsetzen, wo wir normalerweise die Konstante fünf verwenden würden.

```
puts ("Five is " + five());         // Gibt "Five is 5" aus
```

Und weil `return` sofort die entsprechende Funktion beendet und verläßt, könnten wir jetzt folgendes tun:

```
five () {
      return 5;
      puts ("This won't print");
}
```

Das hätte genau den selben Effekt, wie im ersten Beispiel.

3.11.2 IF

```
lessThan5 (int x) {
      if (x < 5)
            puts ("Less than five!");
      puts ("Hello");
}
```

Falls diese Funktion mit einem Parameterwert kleiner als fünf aufgerufen wird, so wird `"Less than five!"` ausgegeben, sofort gefolgt von `"Hello"`. Andernfalls (also ein Parameterwert von "5" oder größer) wird nur `"Hello"` ausgegeben.

Anm.d.Ü.: Wenn Sie schon ein wenig Erfahrungen mit der Sprache C haben, so wird Ihnen aufgefallen sein, das im Gegensatz zur ANSI-C nach der Kontrollstruktur `if` die folgenden Anweisungen nicht in geschweifte Klammern gesetzt werden.

3.11.3 IF … THEN

```
lessThan5 (int x) {
      if (x < 5)
            puts ("Less than five");
      else
            puts ("Greater than or equal to five");
}
```

Falls diese Funktion mit einem Parameterwert kleiner als "5" aufgerufen wird, so wird `"Less than five!"` ausgegeben. Andernfalls (also ein Parameterwert von "5" oder größer) wird `"Greater than or equal to five"` ausgegeben.

3.11.4 WHILE

```
count() {
      int x;
      x = 5;
      while (x > 0) {
            puts(x);
            x=x-1;
      }
}
```

Dieses kleine Programm wird von fünf auf eins herunterzählen. Sicherlich haben Sie bemerkt, das (anders bei if) die Anweisungen innerhalb der while Schleife in geschweifte Klammern gesetzt wurden, so das sie wie eine einzelne Anweisung betrachtet werden.

3.11.5 DO … WHILE

```
count() {
	x = 6;
	do {
		x = x - 1;		//Tipp: Sie könnten auch x-- schreiben
		puts(x);
	} while (x > 0);
}
```

Dieses kleine Programm (ähnlich wie das vorhergehende Beispiel) wird von fünf jetzt allerdings auf null herunter zählen. Dabei wird "0" ebenfalls ausgegeben, weil die Bedingung am Ende der `do...while` Schleife erst zum Schluß ausgewertet wird.

3.11.6 FOR

```
output() {
	string list[4];
	int index;

	list[0] = "Zero";
	list[1] = "One";
	list[2] = "Two";
	list[3] = "Three";

	for (index = 0; index < 4; index++)
		puts (list[index]);
}
```

Dieses kleine Programm wird `"ZeroOneTwoThree"` ausgeben. Wenn wir uns das Programm genau anschauen, bemerken wir zunächst das ein Array deklariert und mit Werten gefüllt wird.

Nun erreichen wir die `for` Schleife selbst. Zunächst wird sie mit Hilfe der Variable `index` initialisiert und mit dem Wert `0` versehen. Dann wird ein Abbruchkriterium festgelegt, in diesem Fall also alle Werte kleiner als vier. Da die Schleife so lange ausgeführt wird, wie die Bedingung "Wahr" ist, wird nun der Inhalt des Arrays ausgegeben. Wenn der Zähler den Wert vier angenommen hat, wird die Schleife abgebrochen und das Programm wird fortgesetzt.

Anm.d.Ü.: Wenn Sie schon ein wenig Erfahrungen mit der Sprache C haben, so wird Ihnen aufgefallen sein, das im Gegensatz zur ANSI-C nach der Schleifenanweisung `for` die folgenden Anweisungen nicht in geschweifte Klammern gesetzt werden.

3.11.7 BREAK

```
count() {
	int x;
	x = 5;
	while (x > 0) {
		if (x == 1)
			break;
		puts(x);
		x = x - 1;
	}
}
```

Dieses - etwas komplexere - Programm zählt wie gewohnt von fünf auf eins herunter. Allerdings wird nur "5432" ausgegeben. Wenn nämlich der Zähler x den Wert "1" erreicht, wird ein `break` ausgeführt. Das führt dazu, das die `while` Schleife vorzeitig abgebrochen wird, bevor "1" ausgegeben werden kann.

3.11.8 CONTINUE

```
count() {
	int x;
	x = 6;
	while (x > 1) {
		x--;		// Verringere x zunächst um 1
		if (x == 3)
			continue;
		puts (x);
	}
}
```

In diesem, völlig frei aus der Luft gegriffenem Beispiel wird "5421" ausgegeben ("3" wird ausgelassen). Wenn nämlich der Zähler x den Wert "3" erreicht, wird ein `continue` ausgeführt. Das führt dazu, das die `while` Schleife erneut gestartet wird, dieses mal allerdings ohne die "3".

3.11.9 SWITCH, CASE, DEFAULT

```
which_number (int x) {
      switch(x) {
            case 1:
                  puts ("x == 1\n");
                  break;
            case 2:
            case 3:
                  puts ("x == 2 or x == 3\n";
                  break;
            case 8:
                  puts("X == 8\n");
            case 10:
                  puts("x == 8 or x == 10\n");
                  break;
            default:
                  puts("x is neither 1, 2, 3, 8 or 10\n";
      }
}
```

Dies ist bislang das längste Beispiel in diesem Kapitel. Der Funktion wird eine ganze Zahl übergeben und soll eine Aussage dazu treffen.

Falls der Übergabewert "1" ist, so wird `case1` ausgeführt wobei das anschließende `break` den `switch` Block verläßt und das Programm beendet wird.

Falls der Übergabewert "2" oder "3" ist, so wird `case3` ausgeführt wobei das anschließende `break` den `switch` Block verläßt und das Programm beendet wird.

Falls der Übergabewert "8" ist, so wird `case8` und `case10` ausgeführt, weil `case8` nicht mit einem `break` beendet wurde. So etwas nennt mach auch *durchfallen*. Das Programm läuft also solange weiter, bis es auf ein `break` oder auf ein `default` stößt.

Falls der Übergabewert nicht einem der definierten Fälle entspricht, so wird des `default` des `switch` blocks ausgeführt. Anschließend wird der Block verlassen und das Programm endet.

3.12 Zeiger

Zeiger sind ein fortgeschrittenes Thema. Bevor Sie hier weiterlesen, sollten Sie die zuvor geschilderten Grundlagen verstanden haben.

Jede Variable ist an einer bestimmten Adresse im Speicher des PDA's abgelegt. Ein Zeiger ist nun eine Art von Variable, welche die Adresse einer Variable speichert auf die verwiesen werden soll.

Es gibt zwei wichtige Operatoren, welche im Zusammenhang mit Zeigern genutzt werden, zum einen das Sternchen (*) und das kaufmännische Und (&).

Das Sternchen dereferenziert den Zeiger, das heißt der Zeiger verhält sich in der Anwendung genau so wie die Variable selbst auf die er zeigt.

Das (&) gibt die Adresse der betreffenden Variable zurück. So können wir genau die Position einer Variable im Speicher ermitteln.

Beispiel für die Benutzung von Zeigern

```
pointer p, q;

int i;


main (){

  i = 5;

  p = &i;           // Weise 'p' die Adresse von 'i' zu

                    //jetzt können Sie '*p' genauso wie 'i' ansprechen und
                      benutzen

  puts(*p);         // Gibt den Wert on 'i' aus

  *p = 7            // Weise den Wert Sieben 'i' zu

  q = p;            // Weise 'q' den Wert von 'p' (welches die Adresse von
                      'i' ist) zu

                    //jetzt können Sie '*q' genauso wie 'i' ansprechen und
                      benutzen


// Was Sie wirklich NICHT tun sollten

  p = 8;                  // weisen Sie einem Zeiger nie eine Konstante zu

  *i = 9;                 // versuchen Sie nie eine gewöhnliche Variable zu
                             de-referenzieren

}
```

Ein Zeiger kann auch dazu benutzt werden, die Position einer Funktion im Speicher zu ermitteln. Dies funktioniert allerdings nicht mit den Standardfunktionen von PocketC. Im Unterschied zu Variablen wird hier das (&) nicht als Operator benutzt.

Eine Funktion mittels eines Zeigers aufzurufen ist nicht ohne weiteres möglich. Wir müssen dazu ein wenig in die Trickkiste greifen.

Zunächst einmal sieht das vorherige Beispielprogramm schrecklich aus. Wir haben keine sprechenden Variablennamen, was uns das Lesen dieses Programmes erschwert (Anm.d.Ü.).

Zweitens wird die Richtigkeit von Parametern nicht geprüft, was bedeutet das Datentypsumwandlungen nicht stattfinden und die Vollständigkeit der benötigten Parameter ebenfalls nicht gesichert ist.

Beispiel für die Benutzung von Zeigern (wesentlich eleganter):

```
func(int x) {return 5+x;}       // Prototyp einer Funktion


main () {
   int result;
   pointer ptr;


   ptr = func;             // Hole die Addresse des Funktionsprototypen
   result = (*ptr)(5);     // Rufe die Funktion auf (ziemlich häßlicher
                           // Aufruf)
// Was Sie nicht tun sollten
   result = (*ptr)("5");   // Dies wird nicht funktionieren, das ein Zeichen
                           // nicht in eine ganze Zahl konvertiert wird
   result = (*ptr)(5,7);   // Es wird zwar erfolgreich kompiliert, wird aber
                           // zur Laufzeit zu einem Stapeüberlauf führen
                           // (Anm.d.Ü.: Ihr PDA wird abstürzen) weil Sie
                           // mehr Parameter übergeben, als benötigt werden.
}
```

3.12.1 Zeiger und Arrays

Zeiger und Arrays sind ziemlich ähnlich. Zeiger können den `[]`-Operator benutzen und das Array (wenn man hier nicht den `[]`-Operator benutzt) liefert die Adresse seines ersten Elements.

Beispiel für die Benutzung von Zeigern und Arrays:

```
int array[5];
pointer p;


main (){
   p = array;        // Weise dem Zeiger 'p' die Adresse des ersten Elementes
                     // von 'array' zu
   *p = 7;           // Weise "7" dem ersten Element von 'array' zu
                     // (array[0])
   p[1] = 8;         // Weise "8" dem ersten Element von 'array' zu
                       (array[1])
}
```

Diese Methode erlaubt es Zeigern, welche auf Arrays referenzieren auch als Funktionsparameter zu dienen. Sie können mit der gleichen Methode eine eigene Art von zwei-dimensionalen Arrays implementieren. Indem Sie ein Array von Zeigern erzeugen, wird jedes Element ein Zeiger zu einem weiteren Array sein (oder einem Teil dessen).

So können Sie ein zwei-dimensionales Array simulieren.

Beispiel für die Erzeugung eines zwei-dimensionalen Arrays:

```
int array[100];

pointer twod[10];          // Nachdem die Funktion init() aufgerufen wurde,
                           // können sie dies wie eine 10*10 große Matrize
                           // benutzen


init() {
   int i;
   for (i=0; i<10;i++)
         twod[i] = array + i*10        // Zeiger Arithmetik
}


main() {
   int x,y;
   init();
   for (x=0; x<10;x++)
         for (y=0;y<10;y++)
               twod[x][y] = x * y;     // Weist Ihrem 2D Array das Produkt
                                       // aus x*y zu
}
```

3.12.2 Zeiger Arithmetik

Die Werte von Zeigern können mit einer begrenzter Zahl von Ausdrücken benutzt werden. Sie können etwas zu einem Zeiger addieren, oder etwas von ihm subtrahieren. Konsequenterweise können Sie dabei auch die Inkrement / Dekrement Operatoren benutzen.

Wenn Sie "1" zu einem Zeiger addieren, wird er zu dem nächsten Wert im Speicher verweisen. Umgekehrt wird er zu dem vorherigen Wert im Speicher zeigen, wenn Sie "1" von einem Zeiger subtrahieren.

Seien Sie jedoch vorsichtig, wenn Sie mittels Zeigern direkt in den Speicher schreiben. Fall Sie in einen reservierten Speicherbereich geraten, so wird dies zwangsläufig zu einem Fehler in Ihrer Anwendung führen (Anm.d.Ü: Ihr PDA wird abstürzen - hier wird in der Regel nur noch ein Reset weiterhelfen).

3.13 Includes

Sobald Sie mit `include` arbeiten, wird es Ihnen möglich sein Programmcode zu schreiben, der die 4 Kilobyte Grenze Ihres Notizbuches überschreitet. Sie können auch Routinen schreiben, die Sie häufig benötigen (Anm.d.Ü: das ist die eigentliche Idee von `include`). So müssen Sie das Rad nicht immer neu erfinden. Der Inhalt der mit `include` eingebetteten Datei wird praktisch an Stelle der `include` Zeile in dem Hauptprogramm kompiliert.

Ein Include kann in der Kopfzeile mit (`/$`) statt (`//`) beginnen, um es nicht in der Liste zu kompilierenden Programme in PocketC erscheinen zu lassen. Sie können natürlich auch **.DOC* Dateien einbetten. Denken Sie jedoch daran, das PocketC zunächst nach einer Notizbuch Datei sucht und erst danach nach einer **.DOC* Datei.

Wenn Sie die PocketC Desktop Edition benutzen, müssen Schrägstriche in den Pfaden mit einem Escape-Zeichen eingeleitet werden.

Beispiel für die Benutzung von Includes:

Zunächst das Include:

```
/$ MyFunctions
times5(int x){
   return x*5;
}
```

Nun das Hauptprogramm:

```
// My Program
include "MyFunctions"

main() {
   int y;
   y = times5(7);
   puts(y);                // Gibt 35 aus
}
```

Der Compiler wird dies so betrachten:

```
// My Program

times5(int x){
   return x*5;
}
```

```
main() {
   int y;
   y = times5(7);
   puts(y);                // Gibt 35 aus
}
```

Bitte denken Sie daran, das Sie das Schlüsselwort `include` nur zu Beginn Ihres Programmes benutzen können. Es ist nicht möglich, dies innerhalb einer Funktion zu tun.

3.14 Bibliotheken

Um eigene Bibliotheken in einem von Ihnen erstellten Programm nutzen zu können, müssen Sie dem Compiler zunächst mitteilen das er diese vorab in den Speicher laden soll. Sie erreichen dies, in dem Sie das Schlüsselwort `library` benutzen

Rufen Sie jedoch nicht die "MathLib" Bibliothek mit dieser Methode auf! Das ist insofern auch nicht nötig, als das sämtliche Funktionen dieser Bibliothek in den Standardfunktionen von PocketC vorliegen.

Beispiel zur einbindung von Bibliotheken

```
// Mein Programm
library "PocketCLib"
main() {
 int x;
 x= times5(7);
}
```

Der Name der Bibliotheksfunktion darf nicht mit Ihrer Benutzerfunktion übereinstimmen.

Allerdings kann er identisch mit dem Name einer Standardfunktion sein. In diesem Fall wird der Compiler Ihre Funktion aufrufen, statt auf die Standardfunktionen zurückzugreifen.

3.15 Sonderzeichen

Es gibt zwei Möglichkeiten einer Zeichenkette ein (oder mehr) Sonderzeichen zu zuweisen. Die erste Möglichkeit ist, die Zeichenkette mit dem ASCII Zahlenwert des Sonderzeichens zu verknüpfen:

```
str = "Here is a neat little square: " + (char)149;
```

Die andere (wesentlich komforTabellere Anm.d.Ü.) Methode ist es, Escape Sequenzen zu benutzen.

Sonderzeichen	Escape Sequenz	Beschreibung
\	\\	--
'	\'	--
"	\"	--
Zeilenvorschub	\n	Auch Bekannt als: "CR" und "Newline"
Tabulator	\t	--
Beliebiges Sonderzeichen	\x[zeichen]	Ein Sonderzeichen, wobei [zeichen] der analoge hex-wert des gewünschten Sonderzeichens ist.

Tabelle 4: Sonderzeichen und Escape Sequenzen

3.16 Der Prä-Kompiler

Genau wie in C, enthält PocketC einen Prä-Compiler, der es Ihnen ermöglicht Makros zu definieren oder Teile Ihres Programmes kompilieren zu lassen welche sich auf ein Makro beziehen.

```
#define macro macro_data
```

Ein Makro ist ein Bezeichner der, sobald er vom Compiler gelesen wird durch die entsprechenden *Makrodaten* ersetzt wird. Makrodaten können jede beliebige Anzahl von Zeichen (einschliesslich Null) beinhalten. Die Makrodaten werden durch eine neue Zeile beendet.

Beispiel für die Benutzung von Makros

```
// My Program
#define calc 5 * 7 (x + 7)


main (){
   int x,y;
   x = 9;
   y = calc;
   puts ("y = " + y);
}
```

Der Compiler wird dies so betrachten:

```
// My Programm
```

```
main (){
   int x,y;
   x = 9;
   y = 5 * 7 (x + 7);                  // Hier werden die Makrodaten
                                          eingesetzt
   puts ("y = " + y);
}
```

Um ein Makro abschzuschliessen, benutzen Sie das #undef Schlüsselwort. Falls Sie zuvor kein Makro definiert haben sollten, so wird an dieser Stelle nichts passieren:

```
Beispiel für die Benutzung von Macros (fortgesetzt)


// My Program
#define DEBUG
main() {
   #ifdef DEBUG
         puts ("In Debugging Mode");
   #else
         puts (In Normal Mode");
   #endif
}
```

Der Compiler wird standardmäßig die folgenden Makros definieren:

```
#define __PKTC__
#define __PKTC_PALM__ 1
#define __PKTC_PRC__ 1    // nur falls die PocketC DesktopEdition ein *.prc
                          // kompiliert
```

Makro Schlüsselwörter	Beschreibung
`#define macro macro_data`	Definiert ein Makro
`#ifdef macro`	Falls "`macro`" zuvor definiert worden ist, so werden die Programmzeilen zwischen diesem Schlüsselwort und `#endif` kompiliert. Andernfalls wird dieser Teil ignoriert.
`#ifndef macro`	Falls "`macro`" nicht zuvor definiert wurde, so werden die Programmzeilen zwischen diesem Schlüsselwort und `#endif` kompiliert. Andernfalls wird dieser Teil ignoriert.
`#endif`	Legt das Ende des Programmbereiches fest, welcher durch `#ifdef` bzw. `#ifndef` eingeleitet wurde.
`#else`	Teilt einen `#ifdef / #endif` bzw. `#ifndef / #endif` Block in zwei Teile. Nun wird lediglich der Programmbereich kompiliert, der zwischen `#else / #endif` liegt und auch nur dann, wenn der erste Teil nicht schon kompiliert worden ist.

Tabelle 5: Prä-Compiler Makro Schlüsselworte

4 Sprachreferenz

4.1 Übersicht

PocketC ist stark an die Syntax von "ANSI-C" angelehnt. Jedoch gibt es Unterschiede in einigen Merkmalen. Vor allen Dingen gibt es keine Strukturen und keine Typdefinitionen. Es werden einige Operatoren nicht unterstützt, wie zum Beispiel das Komma oder der bedingte Operator [?:].Strings werden nun unterstützt, und die Datentypen ihrer Werte werden tolerant behandelt.

Dieses Kapitel richtet sich an diejenigen unter Ihnen, die schon mit der Sprache C vertraut sind. Andernfalls empfehlen wir Ihnen, die vorhergehenden Kapitel und Abschnitte eingehend zu studieren um sich mit PocketC vertraut zu machen.

4.2 Daten Typen

PocketC unterstützt fünf verschiedene Datentypen:

Datentyp	Beschreibung
`int`	Integer, 32Bit mit Vorzeichen
`float`	Fliesskomma, 32Bit Anm.d.Ü.: Als Komma wird der Punkt, nicht aber wie im Deutschen das Komma verwendet!
`char`	8Bit Zeichen, mit Vorzeichen
`strings`	Zeichenketten, ohne Abschluss
`pointer`	Zeiger

Tabelle 6: Zusammenfassung der unterstützen Datentypen

Mit Hilfe dieser Datentypen lassen sich eindimensionale Arrays erzeugen.

4.3 Ausdrücke und Operatoren

Ein Ausdruck enthält eine Anzahl von Variablen, Funktionsaufrufen und Konstanten jeglicher Art. Diese werden mit Operatoren und Klammern miteinander verknüpft.

Beispiele für gültige Ausdrücke:

```
3 + 5

6.7 + 23

"Hello" + "World"

5*3 + foobar()
```

Falls der Ausdruck Operanden unterschiedlichen Datentyps enthält, so werden diese angepasst.

In dem Fall des Ausdrucks

```
5.7 + 8
```

wird die "8" in eine Fließkommazahl umgewandelt, anschliessend werden beide Operanden addiert.

Es wird noch etwas interessanter, wenn jetzt noch ein String mit in's Spiel kommt:

```
"The result is" + (5+8)
```

Zunächst wird der Ausdruck in Klammern berechnet (mit dem Ergebnis 13). Nun wird die "13" in einen String umgewandelt um anschließend mit "`The result is`" verknüpft zu werden.

Andererseits führt der Ausdruck

```
"The result is" + 5 + 8
```

zu der Ausgabe "The result is 58", da zunächst die "5" in ein Zeichen konvertiert wird um anschliessend mit der Zeichenkette "The result is" verknüpft zu werden.

Natürlich kann man auch eine explizite Konvertierung vornehmen.

```
56 + (int)"7"
```

wird "63" ergeben.

Die nun folgende Tabelle gibt Ihnen einen Überblick über die gültigen Operatoren. Sie ist sortiert nach Priorität, die niedrigste zuerst.

Operator	Assoziation	Beschreibung
=	rechts	Wertet zunächst den Ausdruck auf der rechten Seite des Operators aus. Anschließend wird der ermittelte Wert der Variable zu seiner linken zugewiesen.
||	links	logisches "ODER", liefert den Wert 0 falls der Vergleich unwahr ist, 1 falls er wahr sein sollte
&&	links	logisches "UND"
|	links	bitweises "ODER"
^	links	bitweises "XODER"
&	links	bitweises "UND"
== != < <= >= >	links	vergleichende Operatoren: • gleich • ungleich • kleiner als • kleiner oder gleich • größer oder gleich • größer (Liefert den Wert 0, falls der Vergleich unwahr ist, den Wert 1 falls er Wahr ist.)

Operator	Assoziation	Beschreibung
<< >>	links	bitweises verschieben von Werten. Die Operanden müssen entweder vom Datentyp `int` oder `char` sein.
+ -	links	• addition (möglich auch mit Einzelzeichen bzw. Zeichenketten) • subtraktion (nur möglichen mit Zahlenwerten)
* / %	links	• multiplikation • division • modulus Der Modulus kann weder mit Zeichen, noch mit Fliesskommawerten verwendet werden.
- ! ++ -- ~ [] () & @[]		• Negation (Umkehrung) • logisches "NICHT" • Inkrementierung (Werterhöhung) • Dekrementierung (Wertminderung) • bitweises negieren (bitweise das Vorzeichen ändern) • Array Index • Funktions Parameterliste • Addressen Referenz • Zugriff auf ein Einzelzeichen einer Zeichenkette Bis auf das logische "NICHT" und den Zugriff auf ein Einzelzeichen einer Zeichenkette können Sie diese Operatoren mit Zeichenketten als Operanden verwenden.

Tabelle 7: Übersicht der Operatoren

Eine Bemerkung: es wird keine abgekürzte Logik auf die Operatoren `||` und `&&` angewendet. Gleichsam werden die gemischten Operatoren (wie z.B. `+=, *=`), das Komma sowie die bedingten Operatoren (`?:`) nicht unterstützt.

4.4 Variablen

Folgende Grundsätze sollten Sie bei der Benutzung von Variablen berücksichtigen:

- Variablen unterstützen folgende Typen:
  - `int`
  - `float`
  - `char`
  - `string`
  - `pointer`
  - Eindimensionale Arrays
- Lokale Variablen müssen am Anfang einer Funktion deklariert werden.
- Am Besten erzeugen Sie große Arrays global, statt lokal.

- Sie können seit der Version 3.5 Variablen initialisieren. Allerdings müssen diese Ausdrücke sowohl Konstanten sein, als auch den korrekten Datentyp haben. Sie können ebenfalls Arrays initialisieren (z.B. `string days[7]={"Sun", "Mon"};`) Falls nicht alle Elemente initialisiert werden, werden die verbleibenden Elemente mit Null oder einer leeren Zeichenkette gefüllt.
- Alle Variablen werden (gleich ob Global oder Lokal) mit Null oder einer Leerenzeichenkette initialisiert, wenn Sie nicht nach Ihren Vorstellungen initialisieren
- Die Namen von Variablen könne bis zu 31 Zeichen lang sein. Dabei wird bei der Groß- und Kleinschreibung unterschieden.

4.5 Zeiger

Ein Zeiger wird durch `pointer` definiert, nicht wie Beispielsweise in ANSI-C mit `*int`. Dabei ist es wichtig, das Zeiger keinen Datentyp kennen. Statt dessen nehmen sie den Datentyp der Daten an, auf welche sie zeigen. Darüber hinaus können Sie mit Zeigern auch auf Funktionen referenzieren.

Beispiel für Zeiger auf Funktionen:

```
// My Program
func (int x) {return 5*x};
main () {
  pointer ptr;
  int result;
  ptr = func;
  result = (*ptr)(7);
  puts ("5+7=" + result);
}
```

Zeiger sind keine Adressen im physikalischen PDA Speicher.

4.6 String Zugriffsmethoden

Um ein (oder mehrere) Zeichen innerhalb einer Zeichenkette zu erhalten oder zu setzen, benutzen Sie

`stringVariable@[index]`

Der Index des ersten Zeichen ist Null. Sie werden einen Laufzeitfehler produzieren, wenn Sie versuchen auf ein Element zuzugreifen, das ausserhalb der bekannten Bandbreite liegt.

Beispiele für einen gültigen String Zugriff:

```
string str = "bob";
...
puts (str@[1]);           // Gibt das zweite Zeichen der Zeichenkette aus
str@[1] = 'X';            // Ersetzt das "o" von "bob" mit "x"
```

Der Zugriffsoperator kann weder mit Zeigern benutzt, noch kann auf diesem Weg die Adresse einer Zeichenkette ermittelt werden. Anders Ausgedrückt sind die folgenden Ausdrücke ungültig:

```
&str@[i]          *pstr@[i],        (*pstr)@[i]       // SO NICHT!!!!!
```

4.7 Anweisungen

Die folgenden Anweisungen werden unterstützt:

- `for`
- `while`
- `do`
- `break`
- `continue`
- `if`
- `then`
- `return`
- `switch`
- `case`
- `default`

Dabei können Zeichenketten in `switch` Strukturen benutzt werden.

Wichtig! for benötigt in jedem Fall eine Bedingung. So ist zum Beispiel for (;;) nicht zulässig. Benutzen Sie stattdesen `for (;true;)` oder `while (true)`. Auch wird der Komma Operator nicht unterstüzt.

4.8 Funktionen

- Funktionen werden ohne den Typ seines Rückgabewertes deklariert. Weil PocketC nicht so streng mit den Datentypen umgeht, ist es sogar möglich jeden Datentyp (oder gar mehrere verschiedene zurückzugeben). Der zurückgegebene Datentyp ist der, welcher in dem Ausdruck der return Anweisung benutzt wurde. Wird kein Wert explizit zur Rückgabe vorgegeben, so wird eine "0" als Integer zurückgegeben.
- Lokale Variablen müssen vor Funktionen (sowie deren Aufrufe) deklariert werden
- Es muss immer eine `main()` Funktion ohne Parameter geben
- Funktionen können nicht mit Arrays als Parameterliste ausgeführt werden. Allerdings können Sie Zeiger als Paramter empfangen, womit die gleiche Funktionalität gewährleistet ist
- Eine Funktion kann nicht aufgerufen werden, ohne das Sie zuvor definiert worden ist.
- Seit der Version 3.02 werden Funktionsprototypen unterstützt
- Rekursion ist möglich

4.9 Includes

- Es werden drei Ebenen von eingebettetem Code unterstützt, also:
- Notiz A enthält Notiz B welche Notiz C enthält und diese enthält Notiz D, welche nun keine weiteren mehr enthalten darf.
- Das führende "`//`" gehört nicht in den Dateinamen des Aufrufes
- Die eingebettete Datei kann in der Kopfzeile mit "`/$`" beginnen um es vor der Liste der zu kompilierenden Programme von PocketC zu verbergen.
- Die eingebettete Datei kann auch eine **.doc* Datei sein. Der Compiler wird zunächst nach einer Notiz Datei suchen, anschließend erst nach einer **.doc* Datei.
- Wenn Sie die PocketC Desktop Edition benutzen, müssen die Schrägstriche durch eine Escape Sequenz eingeleitet werden.

Beispiele für eingebettete Funktionen:

```
// My large Applet
include "Part1"
include "Part2"


main (){
   // rufen Sie hier die eingebetteten Funktionen auf
}
```

Wichtig! `include` kann nicht innerhalb von Funktionen benutzt werden.

4.10 Bibliotheken

- Bibliotheken werden mit dem Schlüsselwort `library` in den Speicher geladen
- Benutzen Sie `library` nicht um die *math.lib* aufzurufen. Diese Funktionen sind schon innerhalb von PocketC verfügbar.

Beispiele für gültige Ausdrücke:

```
// My Applet
library "PocketCLib"


main() {
   int x;
   // Rufen Sie nun die Bibliotheksfunktionen auf
   x = times5 (7);         // So wie in "PocketCLib" definiert
}
```

4.11 Sonderzeichen

Es gibt zwei Möglichkeiten einer Zeichenkette ein (oder mehr) Sonderzeichen zu zuweisen. Die erste Möglichkeit ist, die Zeichenkette mit dem ASCII Zahlenwert des Sonderzeichens zu verknüpfen:

```
str = "Here is a neat little square: " + (char)149;
```

Die andere (wesentlich komforTabellere Anm.d.Ü.) Methode ist es, Escape Sequenzen zu benutzen.

Sonderzeichen	Escape Sequenz	Beschreibung
\	\\	--
'	\'	--
"	\"	--
Zeilenvorschub	\n	Auch Bekannt als: "CR" und "Newline"
Tabulator	\t	--
Beliebiges Sonderzeichen	\x[zeichen]	Ein Sonderzeichen, wobei [*zeichen*] der analoge hex-wert des gewünschten Sonderzeichens ist.

Tabelle 8: Übersicht der Escapesequenzen in PocketC

4.12 Direktiven des Prä-Kompilers

Die folgenden Direktiven werden vom Prä-Compiler unterstützt

- `#define`
- `#undef`
- `#idef`
- `#ifndef`
- `#endif`
- `#else`

Dabei können Makros keine Parameter erhalten.

Die folgenden Konstanten sind vordefiniert:

- `__PKTC__`
- `__PKTC_PALM__`
- `__PKTC_PRC__`
  sobald die Palm Desktop Edition eine *.prc Datei kompiliert

5 Bibliotheksreferenz

5.1 Übersicht

Die folgende Abschnitte beschreiben die internen Funktionen, welche in PocketC als Standard zur Verfügung stehen. Es ist nicht nötig, eine Bibliothek in den Speicher zu laden um auf diese Funktionen zugreifen zu können.

5.2 Einfache I/O Funktionen

Funktion	Beschreibung
`alert (string msg)`	Eine Nachrichtenbox auf der Konsolenansicht ausgeben. `string` gibt die Titelzeile wieder, `msg` die entsprechende Meldung.
`clear()`	Löscht das Ausgabefenster.
`confirm (string msg)`	Eine Nachrichtenbox auf der Konsolenansicht ausgeben. `string` gibt die Titelzeile wieder, `msg` die entsprechende Meldung. Tippt der User auf <YES>, wird ein integer "1" zurückgegeben. Tippt er auf <NO>, wird ein integer "0" zurückgegeben.
`gets (string prompt)`	Erzeugt einen Eingabedialog auf der Ausgabe, mit `prompt` als Eingabeaufforderung. Gibt die Eingabe als string zurück, wenn der Benutzer auf <OK> tippt. Tippt der Benutzer auf <CANCEL>, wird ein leerer String zurückgeliefert.
`getsd (string prompt, string default)`	Erzeugt einen Eingabedialog auf der Ausgabe, mit `prompt` als Eingabeaufforderung und dem definierten Vorgabewert `default`. Gibt die Eingabe als String zurück, wenn der Benutzer auf <OK> tippt. Tippt der Benutzer auf <CANCEL>, wird ein leerer String zurückgeliefert. Der Dialog kann zwei Zeilen als Eingabeaufforderung beinhalten, benutzen Sie "`\r`" um einen Zeilenumbruch zu erzeugen.
`getsi (int x, int y, int w, string default)`	Erzeugt einen Eingabedialog an der `x/y` Position auf der Ausgabe, mit `prompt` als Eingabeaufforderung und dem definierten Vorgabewert `default`. Das Eingabefeld nimmt dabei die Breite `w` an, auch wenn der Dialog selbst breiter ist. Gibt die Eingabe als string zurück, wenn der Benutzer auf <OK> tippt. Tippt der Benutzer auf <CANCEL>, wird eine leere Zeichenkette zurückgeliefert.

Funktion	Beschreibung
`getsm (int x, int y, int w, int l, string default)`	Erzeugt einen Eingabedialog an der `x/y` Position auf der Ausgabe, mit `prompt` als Eingabeaufforderung und dem definierten Vorgabewert `default`. Das Eingabefeld nimmt dabei die Breite `w` an, auch wenn der Dialog selbst breiter ist. Die Linie des Eingabefeldes ist dabei `l` pixel stark. Gibt die Eingabe als string zurück, wenn der Benutzer auf <OK> tippt. Tippt der Benutzer auf <CANCEL>, wird eine leere Zeichenkette zurückgeliefert.
`puts(string text)`	Gibt eine Zeichenkette auf der Ausgabe aus. Es wird nicht mit einem newline abgeschlossen. Dies erreichen Sie, in dem Sie mit dem Sonderzeichen "\n" abschliessen.

5.3 Ereignissteuerung

Funktion	Beschreibung
`bstate()` Mögliche Ereignisse `1` "Seite Hoch" `0` Kein Ereigniss `-1` "Seite runter" Taste	Prüft den Zustand der Seiten Tasten
`event (int time)` Mögliche Ereignisse `0` Kein Ereigniss `7-10` Knopf 1 bis 4 `1` Zeichen `11` "Menü" Knopf `2` Stift abgesetzt `12` "Home" Knopf `3` Stift hoch `13` "Suchen" Knopf `4` Stift Bewegung `14` "Rechner" Knopf `5` "Seite Hoch" Taste `15` "HotSync" Taste `6` "Seite runter" Taste	Prüft die anstehenden Ereignisse ab. Falls `time` gleich "0" sein sollte, wird die Funktion beendet. Sollte `time` gleich "1" sein, so wartet die Funktion auf ein eintretendes Ereignis. Falls `time` größer als "1" ist, so repräsentiert `time` einen time-out in der Größe einer Hundertstel Sekunde (so wartet z.B. `event(50)` für die Dauer einer halben Sekunde). Um Ereignisse von den Tasten bzw. Displayknöpfen (7-15) zu erhalten, müßen diese mit den entsprechenden `hook` Anweisungen abgefragt werden.
`getc()`	Wartet auf ein geschriebenes Zeichen und liefert es zurück.
`hookhard(int bHook)`	Wenn `bHook` ungleich integer "0" ist, werden die Tasten nicht mehr vom Betriebssystem ausgewertet. Wenn es integer "0" ist, wird die Kontrolle wieder zurückgegeben.
`hookmenu(int bHook)`	Wenn `bHook` ungleich integer "0" ist, wird der Menüknopf nicht mehr vom Betriebssystem ausgewertet. Wenn es integer "0" ist, wird die Kontrolle wieder zurückgegeben.
`hooksilk(int bHook)`	Wenn `bHook` ungleich integer "0" ist, werden Knöpfe auf dem Display nicht mehr vom Betriebssystem ausgewertet. Wenn es integer "0" ist, wird die Kontrolle wieder zurückgegeben.

Funktion	Beschreibung
`hooksync(int bHook)`	Wenn `bHook` ungleich integer "0" ist, wird der HotSync Knopf der Cradle nicht mehr vom Betriebssystem ausgewertet. Wenn es integer "0" ist, wird die Kontrolle wieder zurückgegeben.
`key()`	Fragt das Zeichen ab, welches während des letzten Ereignisses vorlag.
`penx()`	Fragt die x Position des Stiftes ab, der während des letzten Aufrufes von `wait(), waitp()`, oder `event()` vorlag.
`peny()`	Fragt die x Position des Stiftes ab, der während des letzten Aufrufes von `wait(), waitp(),` oder `event()` vorlag.
`pstate()`	Liefer integer "1" wenn der Stift abgesetzt wurde, andernfalls integer "0"
`()`	Wartet auf den Stift oder eingegebenes Zeichen.

5.4 String Funktionen

Funktion	Beschreibung
`ctostr (pointer ptr)`	Gibt eine Zeichenkette zurück, welche aus einem Array mit Einzelzeichen besteht auf die der gegebene Zeiger referenziert. Der betreffende Speicherbereich muß vom Typ char und mit einer "0" abgeschlossen sein.
`format (float f, int prec)`	Wandelt eine Fließkommazahl `f` mit der Anzahl der Nachkommastellen `prec` in eine Zeichenkette um
`hex (int n)`	Rechnet den integer `n` in eine hexadezimale Zahl um
`strleft (string, int len)`	Gibt eine Zeichenkette zurück, welche `len` Zeichen lang ist und an der ganz linken Position der ursprünglichen Zeichenkette beginnt.
`strlen (string)`	Gibt die Länge einer Zeichenkette zurück
`strright (string, int len)`	Gibt eine Zeichenkette zurück, welche `len` Zeichen lang ist und an der ganz rechten Position der ursprünglichen Zeichenkette beginnt.
`strtoc(string str, pointer ptr)`	Füllt einen Array an der Position des Zeigers `ptr` mit den Zeichen der gegebenen Zeichenkette auf. Der Zeiger muss entwerder auf ein Array von Zeigern (inklusive die abschliessende "0") referenzieren oder es muss ein Zeiger sein, welcher vorher mit `malloc()` allokiert wurde. Falls letzteres der Fall ist, müssen Sie sicherstellen das der in Frage kommende Speicherbereich vom Typ `char` ist. Dies erreichen Sie, in dem Sie die Funktion `settype()` aufrufen.

Funktion	Beschreibung

`strtok (`*`string source, pointer result, string delims, int first`*`)`
 	
Beispiel:
 	
`string result;`
 	
`int first;`
 	
`first = strtok("1:2#3", &result, "#:", 0);`
 	
`while (first != -1){`
 	
`puts(result);`
 	
`first = strtok ("1:2#3", &result, first);`
 	
`}`
 	
Das Ergebniss auf der Ausgabe: `"1" "2" "3"`
	Teilt einen String in Einzelzeichen auf. Dabei werden sie von `delims` als Trennzeichen unterteilt. Die so entstandene Zeichenkette wird an der Stelle im Speicher abgelegt, auf die der Zeiger `result` referenziert. Gibt die für den nächsten Aufruf benötigte Adresse in first zurück. Falls das Ende des Strings erreicht wurde, wird integer "-1" zurückgegeben. Falls `result` integer "0" oder NULL ist, so wird die Anzahl der Zeichen in der neuen Zeichenkette zurückgegeben. Die Funktion muss nun nicht erneut aufgerufen werden.

`strupr (`*`string`*`)`	Wandelt eine gegebene Zeichenkette vollständig in Großbuchstaben um.
`strupr (`*`string`*`)`	Wandelt eine gegebene Zeichenkette vollständig in Kleinbuchstaben um.

`substr (`*`string, int first, int len`*`)`
 	
Beispiel:
 	
`substr("Hello", 1, 3)` liefert `"ell"`
	Gibt eine Zeichenkette zurück, welche von der angegebenen Länge und der angegeben Position beginnt.

5.5 Mathematische Funktionen

Bitte beachten Sie: Funktionen, welche die MathLib benötigen, werden immer einen integer "0" zurückliefern, falls diese Bibliothek nicht verfügbar sein sollte.

Anm.d.Ü.: Sie können mathlib() nutzen, um dies zu übeprüfen. Die Funktion ist in der folgenden Tabelle in grauer Farbe hinterlegt.

Funktion	Beschreibung
`acos (`*`float`*`)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`acosh (`*`float`*`)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`asin (`*`float`*`)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`asinh (`*`float`*`)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!

Funktion	Beschreibung
`atan (float)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`atan2 (float x, float y)`	Gibt den Arcustangens aus y/x zurück. Wichtig: *MathLib* muß installiert sein!
`atanh (float)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`cos (float)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`cosh (float)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`exp (float x)`	Gibt e^x zurück. Wichtig: *MathLib* muß installiert sein!
`log (float x)`	Gibt den natürlichen Logarythmus aus x zurück. Wichtig: *MathLib* muß installiert sein!
`log10 (float x)`	Gibt den Logarythmus auf der Basis 10 zurück Wichtig: *MathLib* muß installiert sein!
`mathlib()`	Ist wahr, wenn die Bibliothek installiert wurde und Falsch wenn sie nicht vorhanden ist.
`pow (float x, float y)`	Gibt das Ergebniss aus x^y zurück. Wichtig: *MathLib* muß installiert sein!
`rand()`	Gibt einen zufälligen Wert zwischen 0 und 1 als Fliesskommazahl zurück.
`random (int n)`	Gibt einen zufälligen Wert zwischen 0 und n-1 als Fliesskommazahl zurück.
`sin (float)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`sinh (float)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`sqrt (float x)`	Gibt die Quadratwurzel aus x zurück. Wichtig: *MathLib* muß installiert sein!
`tan (float)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!
`tanh (float)`	Gibt das entsprechende trigonometrische Ergebnis zurück, wobei der Radiant zu Grunde gelegt wird. Wichtig: *MathLib* muß installiert sein!

5.6 Grafische Funktionen

Funktion	Beschreibung
`bitmap (int x, inty, string bits)`	Zeichnet eine Bitmap an der Position `x/y`. `bits` ist eine Liste von hexadezimalen Werten in der Form "`wwxxxxxxxx...`". `ww` ist dabei die Breite der Bitmap (z.B. ist "`0a`" eine 10 Pixel breite Grafik) und "`xxx`" sind die einzelnen Bits der Bitmap, wobei jede Stelle vier Pixel repräsentiert. Diese Funktion wird später in diesem Abschnitt genauer erläutert.
`clearg ()`	Löscht die Grafikausgabe:
`frame (int col, int xl, int yl, int x2, inty2, int radius)` Mögliche Farben `0` Weiß — `2` Grau `1` Schwarz — `3` XOR	Zeichnet ein Rechteck: Eckpunkte sind `x1, y1` und `x2, y2`. Die Ecken selbst werden mit `radius` abgerundet (wobei `radius = 0` normale Ecken sind) und das Rechteck kann die Farbe `col` annehmen. Diese Funktion unterstützt keine Graustufen:
`frame2 (int col, int xl, int yl, int x2, inty2, int radius, int width)` Mögliche Farben `0` Weiß — `2` Grau `1` Schwarz — `3` XOR	Zeichnet ein Rechteck: Eckpunkte sind `x1, y1` und `x2, y2`. Die Ecken selbst werden mit `radius` abgerundet (wobei `radius = 0` normale Ecken sind) und das Rechteck kann die Farbe `col` annehmen. Diese Funktion unterstützt keine Graustufen. `width` (ein integer zwischen "1" und "3") legt die Linienstärke fest.
`graph_off()`	Schaltet in den Standard Modus der Ausgabe. Die bisher vorhandene Grafik wird nicht erhalten bleiben.
`graph_on()`	Schaltet in den grafischen Modus der Ausgabe:
`line (int col, int xl, int yl, int x2, inty2)` Mögliche Farben `0` Weiß — `2` Grau `1` Schwarz — `3` XOR	Zeichnet eine Linie: Eckpunkte sind `x1, y1` und `x2, y2`. Die Linie kann die Farbe `col` annehmen.
`rect (int col, int xl, int yl, int x2, inty2, int radius)` Mögliche Farben `0` Weiß — `2` Grau `1` Schwarz — `3` XOR	Zeichnet ein gefülltes Rechteck: Eckpunkte sind `x1, y1` und `x2, y2`. Die Ecken selbst werden mit `radius` abgerundet (wobei `radius = 0` normale Ecken sind) und das Rechteck kann die Farbe `col` annehmen. Diese Funktion unterstützt keine Graustufen:
`restoreg()`	Lädt die mit `saveg()` zuvor gesicherte Grafik wieder in den Speicher. Diese Funktion kann nur aufgerufen werden, wenn zuvor `saveg()` ausgeführt wurde.
`saveg ()`	Speichert die Grafik intern ab. Gibt einen integer "0" bei einem Fehlschlag zurück, einen integer "1" bei einem erfolgreichen Schreibvorgang.
`text(int x, int y, string str)`	Zeigt eine Zeichenkette `str` an der Position `x/y` an:

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td>
<code>textalign(char xy)</code>
<table>
<tr><td colspan="4">Mögliche Ausrichtungen</td></tr>
<tr><td><code>0</code></td><td>Links</td><td><code>2</code></td><td>Rechts</td></tr>
<tr><td><code>1</code></td><td>Mitte</td><td></td><td></td></tr>
</table>
</td><td>Legt die Ausrichtung des Textes fest, der mit <code>text()</code> ausgegeben wird. <code>xy</code> ist dabei ein integer zwischen "0" und "22", wobei die erste Stelle die vertikale, die zweite Stelle die horizontale Ausrichtung festlegt.</td></tr>
<tr><td>
<code>textattr (int font, int color, int underline)</code>
<table>
<tr><td colspan="4">Mögliche Schriftarten</td></tr>
<tr><td><code>0</code></td><td>Normal</td><td><code>4</code></td><td>Symbol11</td></tr>
<tr><td><code>1</code></td><td>Fett</td><td><code>5</code></td><td>Symbol7</td></tr>
<tr><td><code>2</code></td><td>Groß</td><td><code>6</code></td><td>LED</td></tr>
<tr><td><code>3</code></td><td>Symbol</td><td><code>7</code></td><td>Fett und Groß (OS3.0)</td></tr>
<tr><td colspan="4">Mögliche Farben</td></tr>
<tr><td><code>0</code></td><td>Weiss</td><td><code>2</code></td><td>Invers</td></tr>
<tr><td><code>1</code></td><td>Schwarz</td><td></td><td></td></tr>
<tr><td colspan="4">Mögliche Unterstreichungen</td></tr>
<tr><td><code>0</code></td><td>Ohne</td><td><code>2</code></td><td>Gepunktet</td></tr>
<tr><td><code>1</code></td><td>Normal</td><td></td><td></td></tr>
</table>
</td><td>Setzt Textattribute wie Schriftgrad und -farbe, sowie eine mögliche Unterstreichung in verschiedenen Stärken:</td></tr>
<tr><td><code>title (string title)</code></td><td>Legt den Titel der Grafikausgabe fest.</td></tr>
</table>

5.6.1 bitmap()

Diese Funktion bedarf ein wenig Erläuterung. Eine Bitmap soll durch einen hexadezimalen Wert dargestellt werden.

Sie wird durch `bitmap (int x, inty, string bits)` aufgerufen. Ausgehend von der X/Y Position wird nun eine Grafik erzeugt.

Dazu wird aus einer Matrix ein hexadezimaler Wert ermittelt. Zuvor muß noch die Höhe der Bitmap festgelegt werden. Am besten zeichnet man sich eine Matrize, die in Blöcken zu jeweils vier Spalten eingeteilt und mit "8-4-2-1" beschriftet wird. Je breiter die Grafik werden soll, desto mehr Blöcke werden benötigt. Die Anzahl der Zeilen ist wahlfrei, jedoch muß mindestens eine vorhanden sein.

8	4	2	1	
X	X	X	X	= f

Zunächst addiert man die dezimalen Werte des Blockes, die in der entsprechenden Zeile ein Pixel darstellen sollen. In dem obigen Fall wären das

```
8 + 4 + 2 + 1 = 16
```

Nun das ganze in Hex umwandeln, und wir erhalten `16 = f`

Analog, wird nun die Grafik um die entsprechende Anzahl von Blöcken und Zahlen erweitert. Das Ganze stellt sich dann wie folgt dar:

8	4	2	1	8	4	2	1	8	4	2	1	
X	X	X	X	X	X	X	X	X	X	0	0	= ffc
X	0	0	0	0	0	0	0	0	X	0	0	= 804
X	0	0	0	0	0	0	0	0	X	0	0	= 804
X	X	X	X	X	X	X	X	X	X	0	0	= ffc

Abbildung 1 : Beispiel einer 10x4 Bitmap

8	4	2	1	8	4	2	1	
X	X	X	X	X	0	0	0	= f8
X	0	0	0	X	0	0	0	= 88
X	0	X	0	X	0	0	0	= a8
X	0	0	0	X	0	0	0	= 88
X	X	X	X	X	0	0	0	= f8

Abbildung 2: Beispiel einer 5x5 Bitmap mit einem Punkt in der Mitte

5.7 Sound Funktionen

<table>
<tr><th colspan="4">Funktion</th><th>Beschreibung</th></tr>
<tr><td colspan="4"><code>beep (int type)</code>
Mögliche Schriftarten</td><td rowspan="6">Erzeugt einen Systemton. Bitte beachten Sie, das Systemtöne nicht bei allen PalmOS Versionen identisch sind.</td></tr>
<tr><td>1</td><td>Hinweis</td><td>5</td><td>Alarm</td></tr>
<tr><td>2</td><td>Warnung</td><td>6</td><td>Bestätigung</td></tr>
<tr><td>3</td><td>Fehler</td><td>7</td><td>Klicken</td></tr>
<tr><td>4</td><td>Startup</td><td></td><td></td></tr>
<tr><td colspan="4"><code>tone (int freq, int dur)</code></td><td>Erzeugt einen Ton mit der Frequenz freq und der Dauer dur (in Millisekunden).</td></tr>
</table>

5.8 Zeit- und Datums Funktionen

<table>
<tr><th colspan="4">Funktion</th><th>Beschreibung</th></tr>
<tr><td colspan="4"><code>date (int mode)</code>
Mögliche Modi</td><td rowspan="3">Liefert das augenblickliche Systemdatum zurück.
[Modus 0]: jahr * 10000 + monat * 100 + tag
[Modus 1]: wird aus den Systemeinstellungen ermittelt
[Modus 2]: wird aus den Systemeinstellungen ermittelt</td></tr>
<tr><td>0</td><td>integer</td><td>2</td><td>lange Zeichenkette</td></tr>
<tr><td>1</td><td>kurze Zeichenkette</td><td></td><td></td></tr>
<tr><td colspan="4"><code>seconds()</code></td><td>Die Anzahl der abgelaufenen Sekunden seit dem 01. Januar 1904 minus 2^31.</td></tr>
<tr><td colspan="4"><code>ticks()</code></td><td>Anzahl der abgelaufenen Zeiteinheiten seit dem letzten Reset. Auf aktuellen Geräten liegt dieser Wert bei 100 pro Sekunde. Sie können <code>getsysval()</code> benutzen um den richtigen Wert für Ihr Gerät herauszufinden.</td></tr>
<tr><td colspan="4"><code>time (int mode)</code>
Mögliche Modi</td><td rowspan="3">Liefert das augenblickliche Systemdatum zurück.
[Modus 0]: Stunde * 100 + minute
[Modus 1]: wird aus den Systemeinstellungen ermittelt
[Modus 2]: Stunde * 1000 + minute * 100 + sekunden</td></tr>
<tr><td>0</td><td>integer</td><td>3</td><td>integer</td></tr>
<tr><td>1</td><td>zeichenkette</td><td></td><td></td></tr>
</table>

5.9 Datenbankbezogene I/O Funktionen

Sämtliche Datenbankfunktionen laufen mit beliebigen *CreatorID's*. Allerdings kann die Verwendung von Systemdatenbanken Seiteneffekte verursachen.

Wenn Sie eine neue Datenbank erzeugen, so wird die Datenbank die CreatorID "`PktC`" erhalten und vom Typ "`user`" sein.

Der Versuch eine Datenbank mit einem bestimmten Namen, jedoch einer abweichenden CreatorID und abweichendem Typ zu öffnen wird scheitern.

Es kann nur eine Datenbank zur selben Zeit geöffnet werden.

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td>dbarcrec (int recnum)</td><td>Archiviert den gegebenen Datensatz aus der Datenbank. Der Inhalt des Datensatzes bleibt erhalten, und das "delete" Bit wird gesetzt. Dies führt dazu, das der betreffende Datensatz bei dem nächsten HotSync archiviert wird. Liefert integer "0" bei einem Fehlschlag.</td></tr>
<tr><td>dbbackup (int flag)
Mögliche Backupflags
0 Kein Backup 2 Anfrage
1 Backup</td><td>Setzt das Backup Flag der aktuellen Datenbank. Wenn als Flag "2" gesetzt wird ("Anfrage"), so wird das aktuelle Flag aus dem Fileheader zurückgegeben.</td></tr>
<tr><td>dbcatname (int catID)</td><td>Liefert den logischen Namen der KategorieID.</td></tr>
<tr><td>dbclose()</td><td>Schließt die aktuell geöffnete Datenbank.</td></tr>
<tr><td>dbcreate (string name)</td><td>Erzeugt und öffnet eine Datenbank name. Liefert integer "0" bei einem Fehlschlag zurück. Falls eine Datenbank mit dem selben Namen und der selben CreatorID gleichen Typs existiert, so wird diese zunächst gelöscht. Der aktuelle Datensatz wird auf integer "0" gesetzt.</td></tr>
<tr><td>dbcreatex (string name, string creator, string type)</td><td>Erzeugt und öffnet eine Datenbank name mit der CreatorID creator vom typ type. Liefert integer "0" bei einem Fehlschlag zurück. Falls eine Datenbank mit dem selben Namen und der selben CreatorID gleichen Typs existiert, so wird diese zunächst gelöscht. Der aktuelle Datensatz wird auf integer "0" gesetzt.</td></tr>
<tr><td>dbdelete()</td><td>Schließt und löscht die aktuell geöffnete Datenbank.</td></tr>
<tr><td>dbdelrec (int recnum)</td><td>Löscht den gegebenen Datensatz aus der Datenbank. Der Inhalt des Datensatzes wird entfernt, und das "delete" Bit wird gesetzt. Dies führt dazu, das der betreffende Datensatz bei dem nächsten HotSync endgültig aus der Datenbank entfernt wird. Liefert integer "0" bei einem Fehlschlag.</td></tr>
<tr><td>dbenum (int first, string type, string creator)</td><td>Zählt die installierten Datenbanken auf dem PDA durch, sortiert nach typ und / oder creator. Liefert integer "0" zurück, wenn keine Datenbank mehr zum Zählen vorhanden ist. typ und creator sind jeweils Zeichenketten von vier Zeichen Länge, oder eine leere Zeichenkette als Platzhalter. Um die erste Datenbank zu erhalten, müssen Sie first = 1 setzen. Um die darauf folgenden Datenbanken zu erhalten, setzen Sie first = 0.</td></tr>
<tr><td>dberase()</td><td>Löscht den Inhalt des aktuellen Datensatzes. Der Datensatz selbst bleibt jedoch erhalten.</td></tr>
</table>

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td>dbgetappinfo()</td><td>Wandelt den Infoblock der aktuellen Datenbank in einen Datensatz um und macht diesen vorläufigen Datensatz zu dem aktuellen. Bei einem Erfolg wird die ID dieses Datensatzes zurückgeliefert. Bei einem Fehlschlag wird Integer "-1" geliefert. Bevor Sie die Datenbank schliessen, müssen Sie entweder den vorläufigen Datensatz entfernen oder aber diesen wieder zurück in einen Infoblock umwandeln.</td></tr>
<tr><td>dbgetcat()</td><td>Gibt die Kategorien ID des aktuellen Datensatzes zurück.</td></tr>
<tr><td>dbinfo (string name, pointer pstrType, pointer pstrCreator)</td><td>Liefert den Typ und die CreatorID der gegebenen Datenbank. pstrType and pstrCreator müssen Zeiger auf Zeichketten sein. Liefert Integer "1" bei Erfolg, Integer "0" bei einem Fehlschlag.</td></tr>
<tr><td>dbmovecat (int from, int to, int dirty)</td><td>Setzt für alle Datensätze der Kategorie from die neue Kategorie to. Wenn dirty wahr ist, so wird der betreffende Datensatz vom PDA zum Desktop synchronisiert.</td></tr>
<tr><td>dbnrecs()</td><td>Liefert die Anzahl der vorhandenen Datensätze in der aktuellen Datenbank zurück.</td></tr>
<tr><td>dbopen (string name)</td><td>Öffnet die Datenbank name. Liefert integer "0" bei einem Fehlschlag zurück. Der aktuelle Datensatz wird auf integer "0" gesetzt.</td></tr>
<tr><td>dbpos()</td><td>Liefert die aktuelle Position in der Datenbank zurück. Dabei bedeutet "-1", dass das Ende der Datenbank erreicht ist.</td></tr>
<tr><td>dbread(char type)
Mögliche Typen
c Zeichen
f Fliesskomma
i Integer</td><td>Liest einen Wert von der aktuellen Position in der Datenbank und liefert ihn zurück. Der Lesevorgang muss vom gleichen Datentyp sein, wie der tatsächliche Wert des Datensatzes.</td></tr>
<tr><td>dbreadx (pointer ptr, string format)</td><td>Liest Daten aus der aktuellen Datenbank mit dem gegebenen format und legt sie in den Datenbereich ab, auf den durch ptr referenziert wird. Liefert die Anzahl der gegebenen Werte zurück. Dabei gilt für format gleiches wie für dbwritex().</td></tr>
<tr><td>dbrec (int recnum)</td><td>Setzt den aktuellen Datensatz auf recnum, sowie die aktuelle Position auf integer "0". Falls der Wert des aktuellen Datensatz größer ist als die Anzahl der vorhandenen Datensätze, so werden alle weiteren Leseoperationen fehlschlagen. Allerdings wird eine Schreiboperation einen Datensatz an das Ende der Datenbank anfügen und ihn zum aktuellen Datensatz machen.</td></tr>
<tr><td>dbremrec (int recnum)</td><td>Löscht den gegebenen Datensatz endgültig aus der</td></tr>
</table>

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td></td><td>Datenbank. Liefert integer "0" bei einem Fehlschlag.</td></tr>
<tr><td>dbrename (string name)</td><td>Benennt die aktuell offene Datenbank in name um. name muss dabei <= 31 Zeichen lang sein.</td></tr>
<tr><td>dbseek (int loc)</td><td>Setzt die aktuelle Position in der Datenbank. Falls loc größer als die Anzahl der vorhandenen Datensätze in der Datenbank ist, so wird der nächste Aufruf von dbread() mit der letzten Position durchgeführt.</td></tr>
<tr><td>dbsetappinfo()</td><td>Kopiert den Applikationsinfoblock in einen neuen Datensatz. Liefert integer "1" bei Erfolg zurück. Bei einem Fehlschlag "-1". Diese Funktion ermöglicht es die Informationen des Infoblockes einfach zu modifizieren ohne aufwendig den entsprechenden Speicherbereich direkt modifizieren zu müssen.</td></tr>
<tr><td>dbsetcat (int catID)</td><td>Setzt die gegebene Kategorie ID für den aktuellen Datensatz. Dies ist ein Integer zwischen "0" und "15".</td></tr>
<tr><td>dbsetcatname (int catID, string name)</td><td>Weist catID den Namen name zu.</td></tr>
<tr><td>dbsize()</td><td>Liefert die Größe des aktuellen Datensatzes in Bytes zurück.</td></tr>
<tr><td>dbtotalsize (string name)</td><td>Liefert die Gesamtgröße aller Datensätze der gegebenen Datenbank. Liefert Integer "0" bei einem Fehlschlag.</td></tr>
<tr><td>dbwrite(data)</td><td>Schreibt data an das Ende des aktuellen Datensatzes. data kann dabei von jedem Typ sein, benutzen Sie also die Typumwandlung um sicherzustellen, das Sie mit dem für den Datensatz richtigen Datentyp schreiben. Wenn Sie in die Mitte der Datenbank Zeichenketten ohne "NULL-Abschluss" schreiben, so gehen Sie dabei bitte Vorsichtig vor, da diese Einträge keine feste Länge haben.</td></tr>
<tr><td>dbwritex (pointer ptr, string format)
Mögliche Formate
<table>
<tr><td>c</td><td>ein single-byte</td><td>sz</td><td>NULL-Abschluss Zeichkette</td></tr>
<tr><td>i2</td><td>2-byte Wort</td><td>s#</td><td>Zeichenkette der Länge #</td></tr>
<tr><td>i4</td><td>4-byte Langwort</td><td>f</td><td>4-byte Fliesskomma</td></tr>
</table></td><td>Schreibt Daten mit dem gegebenen Format an die aktuelle Position der geöffneten Datenbank aus dem Datenbereich, auf den durch ptr referenziert wird. format ist dabei eine Liste von Datentypen (typ / werte Pärchen). Um mehr als einen möglichen Wert pro Typ zu definieren, setzen Sie die entsprechende Zahl vor den Typ (z.B. bedeutet '12i2', das Sie 12 2-byte Wörter verwenden wollen). Falls Sie einen Zählerwert direkt neben einem Größenwert haben, so trennen Sie Ihn mit einem Punkt´(z.B. '12i2.8f' bedeutet, das Sie 12 2-byte Wörter gefolgt von 8 Fliesskommazahlen schreiben wollen). Liefert die Anzahl der geschriebenen Datensätze zurück.</td></tr>
<tr><td>dbwritexc (pointer ptr, string format, int count)</td><td>Siehe dbwritex(). Allerdings wird nun die definierte Formatliste count-mal wiederholt. Liefert die Anzahl der geschriebenen Datensätze zurück.</td></tr>
</table>

5.10 Notizbuchbezogene I/O Funktionen

Funktion	Beschreibung
`mmclose()`	Schliesst die aktuelle Notiz.
`mmcount`	Liefert die Anzahl der Notizen in der Notizdatenbank.
`mmdelete`	Schliesst und löscht die aktuelle Notiz.
`mmeof()`	Liefert eine "1" am Ende einer Notiz, andernfalls "0".
`mmfind(`*`string name`*`)`	Öffnet eine Notiz mit name in der ersten Zeile. Liefert Integer "0" bei einem Fehlschlag.
`mmfindx(`*`string name`*`)`	Öffnet eine Notiz mit name in der ersten Zeile. Liefert die NotizID bei Erfolg und Integer "0" bei einem Fehlschlag.
`mmgetl()`	Liefert eine Zeichenkette von der aktuellen Position in der Notiz zurück. Dabei wird ein Zeilenvorschub nicht berücksichtigt.
`mmnew()`	Erzeugt eine neue, leere Notiz. Liefert Integer "0" bei einem Fehlschlag.
`mmopen (`*`int id`*`)`	Offnet eine Notiz mit der gegebenen ID. Liefert Integer "0" bei einem Fehlschlag. Wir empfehlen Ihnen diese Funktion nicht zu benutzen. Sie ist nur der vollständigkeithalber hier aufgeführt.
`mmputs(`*`string`*`)`	Fügt *`string`* an das Ende der aktuellen Notiz an.
`mmrewind()`	Springt zum Anfang der aktuellen Notíz.

5.11 Serielle Schnittstelle

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td><code>serbuffsize (int size)</code></td><td>Allokiert einen seriellen Buffer mit der angegeben Größe (+32 Bytes für den PalmOS overhead). Diese Funktion sollten Sie nur nutzen, wenn Sie ein Überlaufproblem bei der seriellen Datenübertragung haben. Liefert "1" bei Erfolg, "0" bei einem Fehlschlag.</td></tr>
<tr><td><code>serclose()</code></td><td>Schliesst das serielle Interface.</td></tr>
<tr><td><code>serdata()</code></td><td>Liefert die Anzahl von Bytes zurück, welche im Empfangs Buffer liegen.</td></tr>
<tr><td><code>seropen (int baud, string flags, int timeout)</code>
<table>
<tr><td colspan="4">Mögliche Parameterlisten</td></tr>
<tr><td><code>baud</code></td><td colspan="3">300 bis 57600</td></tr>
<tr><td><code>flags</code></td><td>Bits pro Zeichen</td><td colspan="2"><code>6, 7, 8</code></td></tr>
<tr><td></td><td>Parität</td><td><code>N</code></td><td>None</td></tr>
<tr><td></td><td></td><td><code>E</code></td><td>Even</td></tr>
<tr><td></td><td></td><td><code>O</code></td><td>Odd</td></tr>
<tr><td></td><td>Stopp Bits</td><td colspan="2"><code>1, 2</code></td></tr>
<tr><td></td><td>Flusskontrolle</td><td><code>X</code></td><td>Software</td></tr>
<tr><td></td><td></td><td><code>C</code></td><td>CTS</td></tr>
<tr><td></td><td></td><td><code>R</code></td><td>RTS</td></tr>
<tr><td></td><td></td><td><code>H</code></td><td>Handshake</td></tr>
<tr><td></td><td></td><td><code>N</code></td><td><code>Keine</code></td></tr>
</table></td><td>Öffnet das serielle Interface mit den gegebenen Parametern. <code>timeout</code> ist die Spanne von Zeiteinheiten (1/100 sekunde), welche zwischen zwei übertragenen Bytes verstreichen darf. Liefert "0" bei Erfolg zurück. Allerdings sollte ab PalmOS 3.3 vorzugsweise <code>seropenx()</code> genutzt werden.

<code>flags</code> ist dabei eine Zeichenkette von vier Zeichen länge in der Form "8N1C" (Bits pro Zeichen, Parität, Stop Bits, Flusskontrolle)</td></tr>
</table>

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td>
<code>seropenx (int port, int baud)</code>
<table>
<tr><td colspan="3">Mögliche Parameterlisten</td></tr>
<tr><td><code>baud</code></td><td colspan="2">300 bis 57600</td></tr>
<tr><td><code>port</code></td><td>Cradle</td><td><code>0x8000</code></td></tr>
<tr><td></td><td>IR Interface</td><td><code>0x8001</code></td></tr>
</table>
</td><td>Öffnet das serielle bzw. IR Interface mit den angegebenen Parametern. Setzt PalmOS 3.3 oder größer voraus.</td></tr>
<tr><td>
<code>textattr (int font, int color, int underline)</code>
<table>
<tr><td colspan="4">Mögliche Schriftarten</td></tr>
<tr><td><code>0</code></td><td>Normal</td><td><code>4</code></td><td>Symbol11</td></tr>
<tr><td><code>1</code></td><td>Fett</td><td><code>5</code></td><td>Symbol7</td></tr>
<tr><td><code>2</code></td><td>Groß</td><td><code>6</code></td><td>LED</td></tr>
<tr><td><code>3</code></td><td>Symbol</td><td><code>7</code></td><td>Fett und Groß (OS3.0)</td></tr>
<tr><td colspan="4">Mögliche Farben</td></tr>
<tr><td><code>0</code></td><td>Weiss</td><td><code>2</code></td><td>Invers</td></tr>
<tr><td><code>1</code></td><td>Schwarz</td><td colspan="2"></td></tr>
<tr><td colspan="4">Mögliche Unterstreichungen</td></tr>
<tr><td><code>0</code></td><td>Ohne</td><td><code>2</code></td><td>Gepunktet</td></tr>
<tr><td><code>1</code></td><td>Normal</td><td colspan="2"></td></tr>
</table>
</td><td>Setzt Textattribute wie Schriftgrad und -farbe, sowie eine mögliche Unterstreichung in verschiedenen Stärken.</td></tr>
<tr><td><code>serrecv()</code></td><td>Empfängt ein Byte, Liefert "0" bis "255" bei Erfolg, alle Werte größer als "255" sind ein Fehlschlag.</td></tr>
<tr><td><code>serrecva(pointer pData, int size)</code></td><td>Empfängt <code>size</code> Bytes an Daten und legt sie in einem Array ab, auf den mit <code>pData</code> referenziert wird. Der betreffende Speicherbereich darf nur vom Typ <code>int</code> oder <code>char</code> sein und jedes Element wird ein Byte lang sein.</td></tr>
<tr><td><code>sersend(char byte)</code></td><td>Sendet ein Byte. Liefert "0" bei Erfolg.</td></tr>
<tr><td><code>sersendda(pointer pData, int size)</code></td><td>Sendet <code>size</code> Bytes an Daten aus einem Array, auf den mit <code>pData</code> referenziert wird. Der betreffende Speicherbereich darf nur vom Typ int oder char sein und jedes Element darf nur ein Byte lang sein.</td></tr>
<tr><td><code>sersettings (string flags, int timeout)</code></td><td>Setzt die Übertragungsparameter wie in <code>seropen()</code> beschrieben. Nutzen Sie diese Funktion nur in Verbindung mit <code>seropenx()</code>.</td></tr>
</table>

Funktion	Beschreibung
`serwait(int nBytes, int timeout)`	Wartet solange, bis *nBytes* im Empfangsbuffer zur Verfügung stehen. *timeout* ist die Spanne von Zeiteinheiten (1/100 sekunde), welche zwischen zwei aufeinanderfolgenden bytes verstreichen darf. Liefert "-1" bei einem Übertragungsfehler, "1" bei einem Erfolg.

`unpack(pointer pInts, pointer pSerData, string dataSizes, int count)`
 	
Beispiel:
 	
`int data[3], sbuff[8];`
 	
`...`
 	
`serrecva(sbuff, 8);`
 	
`unpack (data, sbuff, "2<24", 8);`
	Entpackt einen Array mit der Größe `count` in einen 1-, 2- und 4-byte langen Integer. *pSerData* ist ein Speicherblock bzw. ein Array aus Bytes (entweder vom Typ Integer oder Character). *pInts* ist ein Speicherblock bzw. ein Array von Integern welcher mit normalgroßen Integern gefüllt wird. *dataSizes* ist eine Zeichenkette, welche die Daten enthält. Jede Zahl (1,2,4) in *dataSizes* repräsentiert dabei einen Integer. Falls einer Zahl ein "<" vorangestellt wurde, dann nimmt die Funktion an das die Bytes in *Little Endian Order* vorliegen. Andernfalls wird *Big Endian Order* angenommen. Die *dataSizes* Zeichkette wird wiederholt abgearbeitet bis *count* unkomprimiert vorliegt.
	
	Ein Beispiel: Sie erhalten 8 Bytes von der Funktion, welche als Big Endian16-Bit Integer, Little Endian, 16-Bit Integer und einen Big Endian 32-Bit Integer interpretiert werden sollte.
	
	Anm.d.Ü.: Sollten Ihnen die Begriffe unheimlich vorkommen, so können Sie mehr unter
	
	http://w3.siemens.de/solutionprovider/_online_lexikon/1/f005791.htm
	
	bzw. unter
	
	http://w3.siemens.de/solutionprovider/_online_lexikon/8/f005068.htm
	
	erfahren.
	
	Eine vollständige (wenn auch englische) Abhandlung zu diesem Thema hat Dr. William T. Verts unter
	
	http://www.cs.umass.edu/~verts/cs32/endian.html
	
	veröffentlicht.

5.12 System Funktionen

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td><code>atexit (pointer func)</code></td><td>Die Funktion mit der gegebenen Adresse wird sofort ausgeführt, falls der Benutzer versucht zwischen zwei Applikationen zu wechseln. Dies findet jedoch nicht statt, wenn der Benutzer auf <DONE> tippt oder aus "APPLET" / "STOP" auswählt.

Eine solche Funktion muss sehr schnell ablaufen und darf weder die Ausgabe nutzen noch das Eventsystem (z.B. mit <code>event(), puts(), alert()</code> oder den Grafikfunktionen).

Eine Warnung: es ist nicht sinnvoll darüber Annahmen zu treffen, an welcher Stelle im Speicher Ihr Programm unterbrochen wurde, als Sie <code>func()</code> aufriefen.</td></tr>
<tr><td><code>clipget()</code></td><td>Liefert den aktuellen Inhalt der Zwischenablage zurück, falls dieser aus Text bestehen sollte.</td></tr>
<tr><td><code>clipset(string text)</code></td><td>Schreibt Text in die Zwischenablage.</td></tr>
<tr><td><code>deepsleep (int sec)</code></td><td>Schaltet das System für <code>sec</code> Sekunden aus. Wir haben eine verläßliche Genauigkeit für 30 Sekunden ermittelt. Ihre Erfahrungen können hiervon durchaus abweichen.</td></tr>
<tr><td><code>exit()</code></td><td>Beendet augenblicklich Ihr Programm. Bei PalmOS 2.x führt Sie dies zu PocketC zurück. Ab PalmOS 3.0 führt Sie dies zum Launcher.</td></tr>
<tr><td><code>getsysval (int index)</code>
Mögliche Systemwerte
<table>
<tr><td><code>0</code></td><td>Benutzername</td><td><code>3</code></td><td>Seriennummer des PDA</td></tr>
<tr><td><code>1</code></td><td>PalmOS Version</td><td><code>4</code></td><td>Zeiteinheit pro Sekunde</td></tr>
<tr><td><code>2</code></td><td>PalmOS V. Zeichenkette</td><td></td><td></td></tr>
</table></td><td>Liefert einen Systemwert.</td></tr>
<tr><td><code>launch (string creatorID)</code>

Beispiele:

<code>launch("memo") // Startet den Notizblock</code>

<code>launch("lnch") // Startet den Launcher ab
PalmOS 3.0</code></td><td>Schliesst PocketC und startet die Applikation mit der angegebenen CreatorID. Liefert "0" bei einem Fehlschlag, kann nichts mehr bei Erfolg liefern.</td></tr>
<tr><td><code>resetaot()</code></td><td>Setzt den Timer für die automatische Abschaltung zurück.</td></tr>
<tr><td><code>sleep (int ms)</code></td><td>System "schläft" für ms Millisekunden.</td></tr>
<tr><td><code>version()</code></td><td>Liefert die aktuelle Versionsnummer von PocketC zurück. Haben Sie z.B. 4.0.3 installiert, ist das Ergebnis "403".</td></tr>
</table>

5.13 Speicherverwaltungsfunktionen

Ein paar Bemerkungen vorab sollen Ihnen helfen, die folgende Übersicht besser zu verstehen:

Normalerweise wird Speicher in Bytes und Wörtern aufgeteilt. PocketC unterteilt den Speicher in "*Werte*". Jeder Wert beinhaltet einen grundlegenden PocketC Datentyp (int, char, float, string, pointer). Ausserdem kennt jedes Speicherelement seinen Datentyp (deswegen wird die `settype()` Funktion benötigt).

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td><code>free (pointer ptr)</code></td><td>Gibt den Speicherbereich wieder frei, auf dessen Block der Zeiger <code>ptr</code> referenziert. Wenn Ihr Programm beendet wird, so werden sämtliche bis dahin allokierten Speicherblöcke automatisch wieder freigegeben, um so Speicherlecks zu vermeiden.</td></tr>
<tr><td><code>malloc (int size)</code></td><td>Allokiert einen Block von <code>size</code> Werten des Typs Integer. Jedoch wird kein konkreter Inhalt definiert. Liefert einen Zeiger zu diesem Block bei Erfolg. Andernfalls wird "0" zurückgegeben. Die Datentypen der Elemente des zurückgelieferten Speicherblocks können von Integer mit Hilfe der <code>settype()</code> Funktion in jeden anderen Datentyp umgewandelt werden. Allerdings sollten Sie <code>malloct()</code> benutzen, da diese Funktion zumeist nicht benutzt, ja sogar von vielen abgelehnt wird.</td></tr>
<tr><td><code>malloct (int nBlocks, string blockTypes)</code>
Mögliche Speichertypen
<table><tr><td><code>i</code></td><td>Integer</td><td><code>f</code></td><td>Fliesskomma</td></tr><tr><td><code>p</code></td><td>Zeiger</td><td><code>s</code></td><td>Zeichenkette</td></tr><tr><td><code>c</code></td><td>Zeichen</td><td></td><td></td></tr></table></td><td>Allokiert nBlocks im Speicher, wobei die Datentypen durch <code>blockTypes</code> definiert werden. <code>blockTypes</code> ist eine Zeichenkette, welche die gültigen Datentypen repräsentiert. Der Speicherblock wird mit "0" initialisiert, wenn der Typ numerisch ist. Sonst mit einem leeren Stringh, wenn es sich im einen String handelt. So wird Beispielsweise <code>malloct(3, "piis")</code> 12 Werte allokieren. Der erste wird ein Zeiger sein, der zweite und dritte ein Integer, der vierte ein String, der fünfte ein weiterer Zeiger, der sechste und siebte ein Integer ...

Einem Datentyp kan ein Zähler vorangestellt werden, der angibt, wie oft dieser Datentyp vorkommen soll. So ist "<code>p2is</code>" das selbe wie "piis" und "<code>8f</code>" das selbe wie "<code>ffffffff</code>".</td></tr>
<tr><td><code>memcpy (pointer des, pointer src, int size)</code></td><td>Kopiert den Inhalt des Speicherblocks <code>src</code> in den Speicherblock <code>des</code>. Der Datentyp des Zielblock bleibt dabei nicht erhalten. Wenn z.B. <code>dest</code> auf einen String zeigt, und <code>src</code> auf einen Integer, so wird <code>dest</code> den Typ Integer annehmen.</td></tr>
</table>

<table>
<tr><th>Funktion</th><th>Beschreibung</th></tr>
<tr><td>
settype(pointer ptr, int size, char type)
<table>
<tr><td colspan="4">Mögliche Speichertypen</td></tr>
<tr><td>i</td><td>Integer</td><td>f</td><td>Fliesskomma</td></tr>
<tr><td>p</td><td>Zeiger</td><td>s</td><td>Zeichenkette</td></tr>
<tr><td>c</td><td>Zeichen</td><td colspan="2"></td></tr>
</table>
</td><td>Legt den Datentypen der mit size zusammenhängenden Werte im Speicher fest, auf den mit ptr referenziert wird. Benutzen Sie diese Funktion nur im Zusammenhang mit malloc(). Liefert bei einem Fehlschlag "0".</td></tr>
<tr><td>
typeof(pointer ptr)
<table>
<tr><td colspan="4">Mögliche Speichertypen</td></tr>
<tr><td>i</td><td>Integer</td><td>f</td><td>Fliesskomma</td></tr>
<tr><td>p</td><td>Zeiger</td><td>s</td><td>Zeichenkette</td></tr>
<tr><td>c</td><td>Zeichen</td><td colspan="2"></td></tr>
</table>
</td><td>Ermittelt den Datentyp des Speicherblocks, der mit ptr angesprochen wird. Die Ausgabe ist ein char.</td></tr>
</table>

6 Die Entwicklung eigener Bibliotheken

6.1 Übersicht

Die Entwicklung eigener Bibliotheken ermöglicht es Ihnen, Programme zu schreiben die schneller Ablaufen, als reine ProcketC Programme. Es ermöglicht weiterhin, dem Benutzer Funktionalitäten in einem Programm zur Verfügung zu stellen, welche die internen PocketC Funktionen nicht bieten. Solche Bibliotheken können durch Veröffentlichung anderen Entwicklern zur Verfügung gestellt werden. So ist jedem geholfen, der in PocktC programmiert und Teil der PalmOS Gemeinschaft ist.

Dieses Handbuch kann nicht die PalmOS System Bibliotheken abhandeln. Das, was Sie theoretisch zur Entwicklung eigener Bibliotheken darüber wissen müssen, ist entweder in diesem Handbuch vorhanden oder in der Beispielsbibliothek hinterlegt. Sollten Sie dennoch mehr Informationen suchen, empfehlen wir das Sie sich ein wenig mit dem PalmOS SDK auseinandersetzen.

Da wir einfach einmal annehmen , das Sie über eine Lizenz von "Code Warrior" verfügen, sollte die Entwicklung einer eigenen Bibliothek für Sie kein Fremdwort sein. Sollten Sie jedoch andere Tools benutzen (wie zum Beispiel GCC), so müssen Sie ein wenig mehr tun. Daher müssen wir leider jede Unterstützung für GCC Entwickler ablehnen, da wir mit diesen Tools weder arbeiten, noch PocketC damit getestet haben.

Allerdings hat Jeremy Rixon herausgefunden wie es geht und er erklärt es Ihnen gerne. Schauen Sie einfach mal bei http://rixon.org/FractalLib/index.html vorbei.

Es gibt ein paar **grundlegende Schritte** um eine PocketC Bibliothek zu entwickeln:

1. Kopieren Sie das "PocketCLib" Projekt in ein neues Verzeichnis in der CodeWarrior Verzeichnisstruktur. Benennen Sie es wie Sie es mögen und öffnen Sie es.

2. Modifizieren Sie die Einstellungen dieses neuen "PocketCLib" Projektes, in dem Sie den Ausgabenamen und die CreatorID ändern. CreatorID's müssen eindeutig sein. Sollten Sie also Ihre Bibliothek veröffentlichen wollen, so müssen Sie eine CreatorID bei Palm beantragen. Möglicherweise müssen Sie noch einige include / lib Pfade ändern.

3. Fügen Sie alle globalen Variablen in "*PocketCLib.h*" hinzu, welche Sie in "PocketCLibGlobalsType" festgelegt haben.

4. Nun wenden Sie sich der Datei "*PocketCLib.cpp*" zu:
   - Fügen Sie Programmteile der Funktion `PocketCLibOpen()` hinzu, um globale Variablen zu initialisieren.
   - Fügen Sie Programmteile der Funktion `PocketCLibClose()` hinzu, um globale Variable zu löschen.
   - Schreiben Sie die Funktionen, die Ihre Bibliothek PocketC zur Verfügung stellen soll
   - Fügen Sie die entsprechenden Funktionsprototypen Ihrer Funktionen der Funktion `PocketCLibAddFunctions()` hinzu.
   - Fügen Sie die nötigen Aufrufe zu Ihren Funktionen in dem switch der Funktion `PocketCLibExecuteFuntions()` hinzu

5. Es werden keine Modifikationen an der Datei *PocketCLibDispatch.cpp* benötigt

6. Die Beschreibungsdatei *PocketCLib.lib* der Desktop Edition muss ein paar Einträge erhalten
   - Listen Sie jede Funktion in der Reihenfolge, welche Sie in der Funktion `PocketCLibAddFunctions()` festgelegt haben, auf
   - Deklarieren Sie jede Funktion genauso, wie Sie es in einem PocketC Programm tun würden. Schliessen Sie dabei mit einem Semikolon statt mit einem Funktionsrumpf ab.

7. Kompilieren Sie Ihre neue Bibliothek und erfreuen Sie sich an Ihren neuen Funktionen!

6.2 Allgemeine Informationen

6.2.1 Werte

Ein Wert ist ein C++ Datentyp, der alle PocketC Datentypen in Form einer Vereinigung (Union) repräsentiert. Einem Wert ist dabei ein eigenes Feld zugeordnet, welches seinen gültigen Datentypen beschreibt. Abhängig des Typs wird nur eines dieser anderen Felder benutzt. Dabei kann ein Typ einer der folgenden sein:

- `vtInt`
- `vtFloat`
- `vtChar`
- `vtString`

Wichtig: Normalerweise wird eine Zeichenkette irgendwo in einem dynamischen Speicherbereich abgelegt und ein Wert speichert einen Handle dazu, keinen Zeiger. Wird eine Zeichenkette nicht mehr benötigt, so muß sie nun mit `cleanup()` entfernt werden.

6.2.2 Funtionsprototypen

Die Deklaration einer Funktion (erzeugt mit `addFunc()`) besteht in der Regel aus einem Namen, einer Reihe von Argumenten und einer Liste mit bis zu zehn Argumenttypen. Dabei können die Argumente aus Standardtypen zusammengesetzt sein oder vom Typ `vmVoid` sein. Ein Parameter vom Typ *`vmVoid`* weist den Compiler an, keine Typumwandlungen für Parameter mehr durchzuführen. Anders ausgedrückt können Sie jeden Typ für einen Wert übergeben und es findet keine automatische Typumwandlung mehr statt.

6.2.3 Der Stapel

PocketC legt alle Parameter in der Reihenfolge auf dem Stapel ab, wie sie in der Deklaration der Funktionen aufgetreten sind (FILO Prinzip, Anm.d.Ü:).

Wird also einer Funktion mit zwei Parametern deklariert (z.B. `func(int x, int y)`), so wird zunächst `x` auf dem Stapel abgelegt, gefolgt von `y`. Analog wird `y` als erstes vom Stapel geholt.

6.2.4 Rückgabewerte

Eine Funktion setzt den Rückgabewert in dem der Wert der globalen Variable *`retVal`* zugewiesen wird. Wenn PocketC eine Bibliothek aufruft, so wird der Rückgabewert auf `type = vtInt` und `vtInt = 0` gesetzt.

6.2.5 Die C++ Funktion

Die C++ Funktion, welche Ihre PocketC Funktion implementiert nutzt nur einen Parameter: einen Zeiger auf globale Daten. Diese wird den PocketC Parameter erhalten, indem sie die `pop()` Funktion benutzt. Anschliessend wird `cleanup()` auf alle Stringparameter angewendet.

6.2.6 Verweise von Zeigern auf Strings

Wir haben in der Version 3.5 die interne Repräsentation von Zeichenketten geändert. Ein Wert, den eine Bibliothek vom Stapel schiebt wird dennoch im alten Format vorgehalten, um abwärtskompatibel zu bleiben, weil `retVal` den Rückgabewert für die Bibliotheken setzt.

Falls jedoch eine Bibliothek einen Zeiger dereferenziert, welcher zuvor der Funktion übergeben worden ist, so wird dieser Wert das neue Format annehmen. Dabei gibt es zwei Arten von Zeichketten, Konstanten und referenzierten Zeichenketten. Falls eine Zeichkette eine Konstante darstellt, das *`sVal`* Attribut des Wertes ist ein *`char*`* der nicht freigegeben werden sollte. Falls es sich um eine referenzierte Zeichenkette handelt, so ist das *`sVal`* Attribut ein Handle dieser Struktur (und hat damit das wichtigste Bit gesetzt).

Beispiel für Verweise auf Strings

```
struct String {

   StrType sType;          // wird ignoriert und nur in debug Kompilaten
                              genutzt

   unsigned int nRef;      // aktueller Verweis

   char data[1];           // Array, dessen Größe mit einer Variable gesetzt
                              wird

}
```

6.3 Standardbibliothek Exports

Funktion	Beschreibung
`PocketCLibOpen()`	Diese Funktion wird sowohl während des übersetzens, als auch zur Laufzeit des Programmes aufgerufen. Sie sollte Ihre globalen Variablen setzen und einen Zeiger auf die reservierten globalen Variablen liefern. Die Funktionszeiger in der globalen Struktur werden nach dem Aufruf gesetzt, also sind sie zur Zeit des Aufrufes nicht verfügbar.
`PocketCLibClose()`	Diese Funktion wird sowohl während des übersetzens, als auch zur Laufzeit des Programmes aufgerufen. Sie sollte Ihre globalen Strukturen freigeben..
`PocketCLibAddFunctions()`	Diese Funktion wird nur während des übersetzens aufgerufen. Sie sollte durch den Aufruf von `addFunc()` Funktionsprototypen hinzufügen. Die erste hinzugefügte Funktion hat dabei den Index "0"
`PocketCLibExecuteFunction()`	Diese Funktion wird nur zur Laufzeit aufgerufen. Sie sollte einen `switch` enthalten, der Ihre C++ Funktionen aufruft. Diese werden die PocketC Funktionsaufrufe behandeln. Die Indizes dieser Funktionen müssen mit der Reihenfolge übereinstimmen, die mit `PocketCLibAddFunctions()` festgelegt wurde.

6.4 Globale Funktionen

Funktion	Beschreibung
`pop(`*`Wert&`*`)`	Schiebt einen Wert vom Stapel. Falls es sich um eine Zeichenkette handelt, müssen Sie anschließend `cleanup()` aufrufen.
`push(`*`Wert&`*`)`	Schiebt einen Wert auf den Stapel. Der Wert wird genaugenommen auf den Stapel kopiert. Falls es sich dabei um eine Zeichenkette handelt, so wird diese zwar ebenfalls kopiert, Sie jedoch müssen anschließend `cleanup()` auf den ursprünglichen Wert anwenden.
`cleanup(`*`Wert&`*`)`	Gibt den gegebenen Speicherbereich wieder frei, welcher zuvor durch eine Zeichenkette belegt wurde. Falls der Wert keine Zeichenkette ist, so wird keine Behandlung durchgeführt. Wir empfehlen Ihnen jedoch, diese Funktion dennoch aufzurufen, auch wenn es nicht Nötig ist.
`typeCast(`*`Wert&, VarTyp`*`)`	Wandelt einen Wert in den gegebenen Datentyp um. Falls es sich bei dem neuen Wert um eine Zeichenkette handelt, so muss der Speicher wieder freigegeben werden, sobald er nicht mehr benötigt wird. Falls der Ursprungswert eine Zeichenkette ist welche umgewandelt wurde, so wird der Speicher automatisch freigegebnen.
`typMatch(`*`Wert&, Wert&`*`)`	Wandelt beide Werte in einen übereinstimmenden Datentypen um. Wenn zum Beispiel diese Funktion als Integer aufgerufen wird und der zweite Wert eine Flieskommazahl ist, so werden beide Werte Integer sein. Falls einer von beiden Werten eine Zeichenkette ist, so sind beide anschliessend jeweils vom Typ String.
`UIYield(`*`bool blocking`*`)`	Erlaubt dem System PalmOS Ereignisse zu behandeln. Diese Funktion setzt auch die Rückgabewerte der PocketC `event()` Funktion und behandelt so PalmOS Ereignisse "im Auftrag" von PocketC. Falls Ihre Operation viel Zeit benötigt, sollten Sie diese Funktion benutzen. Wenn der *`blocking`* Parameter wahr ist, so wird nur ein PalmOS Ereignis behandelt. Sollten keine zur Verfügung stehen, so wird das System auf eines warten. Falls *`blocking`* Falsch sein sollte, so wird das System alle Nachrichten behandeln und zur ursprünglichen Funktion zurückkehren.
`callFunc(`*`int location`*`)`	Ruft die PocketC Funktion an der Adresse *`location`* im Speicher auf. Bevor Sie diese Funktion aufrufen, müssen Sie die Parameter auf den Stapel schieben (in der Reihenfolge, in der die Funktion die Parameter übergeben bekommen soll). Sie können die Adresse einer Funktion ermitteln, in dem Sie in Ihrem Programmtext diese Funktion ohne Anführungszeichen aufrufen (z.B. `puts(main);` wird die Adresse der Funktion `main()` liefern). Allerdings kann diese Funktion nicht dazu genutzt werden, um die internen Funktionen von PocketC aufzurufen. Es ist auch nicht sehr geschickt, diese Funktion rekursiv aufzurufen. Wichtig: Der Rückgabewert dieser Funktion wird ebenfalls auf dem Stapel abegelegt. Sie müssen ihn vorher herunterschieben, nachdem diese Funktion aufgerufen wurde. Ein Aufruf wird *`retVal`* verändern.

Funktion	Beschreibung
`callBI(char* name)`	Ruft die gegebene PocketC Funktion mit dem definierten Namen auf. Bevor Sie dies tun, müssen Sie die Parameter auf den Stapel schieben, und zwar in der Reihenfolge, wie sie durch die betreffende Funktion benötigt werden. Der Rückgabewert wird in *retVal* abgelegt. Wenn die Funktion eine Zeichenkette zurückgibt, so müssen Sie zunächst `cleanup()` aufrufen bevor Sie *retVal* überschreiben. Diese Funktion gibt "Falsch" zurück, wenn die aufgerufene Funktion nicht existiert.
`deref(int ptr)`	Derefenziert einen PocketC Zeiger und gibt einen Zeiger zurück, der auf den Speicherbereich zeigt, auf den der Ursprüngliche Zeiger referenziert hat. Bedenken Sie bitte, das dies ein Zeiger auf einen tatsächlichen Wert, nicht auf eine Kopie von ihm, ist.
`vmCtrl(UInt32 id, UInt32 val)`	Führt eine Operation aus, welche das Verhalten der virtuellen Machine beinflusst. Die `id` ist eine Steuer ID, und `val` der Wert der durch diese Steuer ID benötigt wird. Zur Zeit ist die folgende SteuerID implementiert: `VMCTR_ENABLE_EVENTS` (De-) Aktiviert die im Hintergrund ablaufende Ereignisbehandlung der virtuellen Maschine. *val = 1* aktiviert sie, *val = 0* dekativiert diese. Wenn die virtuelle Maschine startet, so wartet sie alle 100 VM Befehle auf Ereignisse. Sollten Sie jedoch ein *appStop* Ereigniss empfangen, so müssen Sie die Eventsteuerung wieder aktivieren und die Ereignisnachricht erneut an die Nachrichtenschlange senden. Wenn Sie diese Funktion deaktivieren, wird zwar einerseits die Performance des Systems erhöht, andererseits kann dann Ihr Programm nicht mehr auf eintretende Ereignisse reagieren. Und das betrifft auch so wichtige Dinge, wie den Ein- / Aus Schalter!

6.5 Die Benutzen eigener Bibliotheken

Um eigene Bibliotheken in einem von Ihnen erstellten Programm nutzen zu können, müssen Sie dem Compiler zunächst mitteilen das er diese vorab in den Speicher laden soll. Sie erreichen dies, in dem Sie die entsprechende "LIBRARY" in den Speicher laden.

BEISPIEL

```
// Mein Programm

library "PocketCLib"

main() {

 int x;

 x= times5(7);

}
```

Der Name der Bibliotheksfunktion darf nicht mit Ihrer Benutzerfunktion übereinstimmen.

Allerdings kann er identisch mit dem Name einer Standardfunktion sein. In diesem Fall wird der Compiler Ihre Funktion aufrufen, statt auf die Standardfunktionen zurückzugreifen.