

# RedCrab<sup>*PLUS*</sup>

The Calculator

Programers Manual

Version 4.43

copyright © by Redchillicrab, Singapore 2009..2014

# Inhalt

- 1.1 Einleitung
- 1.2 RedCrab Program Interpreter
- 1.3 Kommentare
- 1.4 Identifikatoren
- 1.5 Gültigkeitsbereich eines Identifikators
- 1.6 String Konstante
- 1.7 Programmvariable
- 1.8 Numerische Ausdrücke
- 1.9 Boolesche Ausdrücke
- 1.10 Ausdrücke
- 1.11 Mehrzeilige Anweisungen

- 2.1 Program
- 2.2 Uses
- 2.3 Define
- 2.4 Let
- 2.5 While do
- 2.6 If Then
- 2.7 Else
- 2.8 Elseif
- 2.9 Function
- 2.10 Forward
- 2.11 Call
- 2.12 Result
- 2.13 Next
- 2.14 Index

- 3.1 Input
- 3.2 Display
- 3.3 Menu

- 4.0 Debugger
- 4.1 Trace
- 4.2 Break
- 4.3 Watch

- 5.1 PHP Programm Schnittstelle
- 5.2 PHP Input
- 5.3 PHP Output

# 1.1 Einleitung

In *RedCrab* können ein oder mehrere Datei (Module) geöffnet werden, die programmierte Funktionen oder Daten enthalten. Diese Anleitung beschreibt die Syntax des *RedCrab* Interpreters (*RCI*) und die *PHP* Schnittstelle.

## 1.2 RedCrab Program Interpreter

*RedCrab* verwendet eine eigene einfache Programmiersprache. Auch Benutzern ohne Programmier-Kenntnisse sollen in der Lage sein nach kurzer Einarbeitung einfache Funktionen zu schreiben.

Analog zum Arbeitsblatt wird bei reservierten Namen (Systemfunktionen und Anweisungen) nicht zwischen Groß- und Kleinschreibung unterschieden. Bei selbst-definierten Funktionen und Variablen wird Groß- und Kleinschreibung berücksichtigt.

Ein Programm besteht aus einer Folge von Anweisungen. Jede Anweisung beginnt mit einem Schlüsselwort. Das Zeilenende (*Linefeed*) beendet eine Anweisung. Die Schlüsselwörter *function*, *if* und *while* gelten für alle folgenden Anweisungen, bis der Anweisungsblock mit dem Schlüsselword *end* beendet wird. Extra Leerzeichen, Tabulatoren, Linefeeds und Kommentare werden vom Programm ignoriert.

Beispiel:    **let** a = 12  
              **let** b = 22

Es können mehrere Anweisungen in eine Zeile geschrieben werden, wenn sie mit einem Doppelpunkt getrennt werden.

Beispiel:    **let** a = 123 : **let** b = 22

## 1.3 Kommentare

Aus Gründen der Übersicht und zur Beschreibung der Funktionen, wird empfohlen, dass Sie Ihren Code dokumentieren, indem Sie Kommentare einfügen. Sie können auch mit Kommentar Symbole während der Programmentwicklung Teile des Programms deaktivieren ohne sie zu löschen. Kommentare können auf zweierlei Weise eingefügt werden:

- Ein Kommentar kann mit der runden Klammer, gefolgt von dem Multiplikation Symbol (\*) beginnen und mit den Zeichen \*) enden. Sie können mit den Symbolen keine Kommentare verschachteln.
- Sie können einen Kommentar mit einem doppelter Schrägstrich einleiten //. Der Kommentar endet mit dem Ende der Zeile.

## 1.4 Identifikatoren

*RedCrab* Programme enthalten Verweise auf Module, Funktionen, lokale und globale Variablen und Konstanten. Mit Ausnahme von Konstanten, wird jeder Verweis über seinen Name identifiziert. Der Name besteht aus eine Folge von Buchstaben, Ziffern und Unterstrichen. Das erste Zeichen muss ein Buchstabe oder eine Unterstrich sein.

## 1.5 Gültigkeitsbereich eines Identifikators

Lokale Variablen werden innerhalb einer Funktion definiert. Auf Sie kann nicht von außerhalb ihrer Funktion zugegriffen werden.

Globale Variablen werden außerhalb einer Funktion definiert. Auf sie kann von allen Funktionen im Modul zugegriffen werden. Von externen Modulen und vom Arbeitsblatt können sie nur ausgelesen werden.

Funktionen können von allen Modulen und vom Arbeitsblatt aufgerufen werden.

## 1.6 String Konstante

String Konstante sind Sequenzen von Zeichen die in Anführungszeichen eingeschlossen sind. Strings müssen in einer Zeile geschrieben werden. Mit dem Punkt (.) können einzelne String Konstante miteinander verbunden werden.

Beispiel: `let s = "Hallo "`  
`let t = s ."Welt"`

Die Variable *t* enthält den Text: „*Hallo Welt*“

## 1.7 Programmvariable

Programmvariable müssen definiert werden , bevor sie verwendet werden. Dazu wird das Schlüsselwort *define* verwendet. Optional kann in der Definition auch ein Wert zugewiesen werden. Für weitere Informationen lesen Sie unten die Beschreibung zu *define*.

## 1.8 Numerische Ausdrücke

Ein numerischer Ausdruck besteht aus einer Konstante, Variable, Zelle ein Datenfeldes oder einer Funktion die einen Zahlenwert liefert, oder mehrere davon die durch folgende arithmetische Operatoren verbunden sind:

*	Multiplikation
/	Division
Mod	Modulo
Div	Integer Division
+	Addition
-	Subtraktion

## 1.9 Boolesche Ausdrücke

Ein boolescher Wert bewertet einen Ausdruck als wahr (*TRUE*) oder unwahr (*FALSE*) und hat folgendes Format:

Ausdruck Operator Ausdruck

Die Ausdrücke können numerisch oder Text Strings sein. Wenn ein numerischer Ausdruck mit einem String verglichen wird der eine Zahl enthält, wird zum Vergleich der Wert der Zahl verwendet. Werden zwei Strings miteinander verglichen, werden sie immer als Strings behandelt, unabhängig davon, ob sie Text oder Zahlen beinhalten. Die Operatoren zeigt die folgende Liste.

Operator	Operation
==	Gleich
<>	Ungleich
>	Größer als
>=	Größer oder gleich.
<	Kleiner als
<=	Kleiner oder gleich

Boolesche Ausdrücke können mit den *UND*( & ) und *ODER*( / ) Operatoren zusammengefaßt werden.

Beispiel:     (a >= b) & (c <= d)  
             (a == d) | c

## 1.10 Ausdrücke

Für *if* und *while*-Anweisungen ist *TRUE* jede Zahl ungleich Null und *FALSE* ist gleich Null. Deshalb können Sie immer einen numerischen Ausdruck einsetzen, wo ein boolescher Ausdruck gefragt ist. Sie können ebenso einen booleschen Ausdruck verwenden, wo ein numerischer Ausdruck erwartet wird, was als 1 oder

0 interpretiert wird. Sie können einen String-Ausdruck, der eine Zahl darstellt einsetzen wo ein numerischer Ausdruck erlaubt ist.

## 1.11 Mehrzeilige Anweisungen

Eine Feld-Definition kann in der nächsten Zeile fortgesetzt werden. Die aktuelle Zeile muß mit einem Komma oder einem Semikolon beendet werden.

Ein Statement kann in der folgenden Zeile fortgesetzt werden, wenn die aktuelle Zeile mit einem Backslash ( \ ) beendet wird.

Eine lange Zahl kann in der folgenden Zeile fortgesetzt werden, wenn die aktuelle Zeile mit einem doppelten Backslash (\\) beendet wird.

Die folgende Tabelle zeigt einige Beispiele zu mehrzeiligen Statements:

Zweizeilige Statements	Interpretation
<code>Let m = [1,2,3;           4,5,6]</code>	<code>Let m = [1,2,3;4,5,6]</code>
<code>Let m = [1,2,3,4,           5,6,7,8]</code>	<code>Let m = [1,2,3,4,5,6,7,8]</code>
<code>Let v \       = 1 + 2</code>	<code>Let v = 1 + 2</code>
<code>Let v = 12345\\           6789</code>	<code>Let v = 123456789</code>
<code>Let s = "hello " \           ."world"</code>	<code>Let s = "hello " ."world"</code>



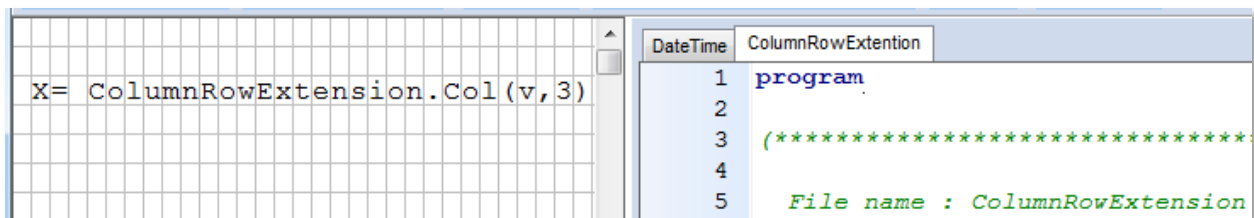
## 2.1 Programm

Eine Programmdatei wird immer mit dem Schlüsselwort *program* eingeleitet. Optional kann ein Name angegeben werden. Die Programmdatei wird in einer Registerseite unter ihrem Dateinamen angezeigt, oder unter dem optional angegebenen Namen.

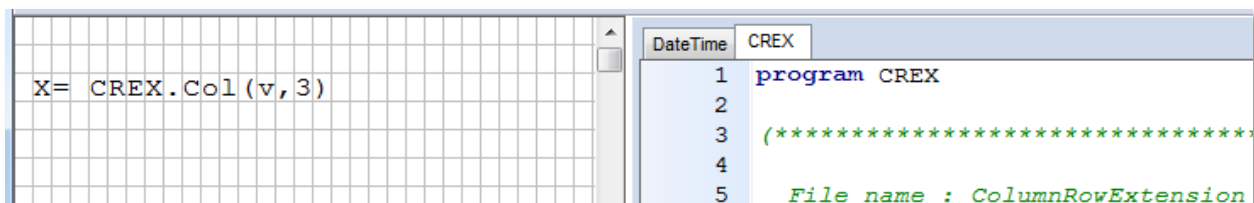
Die Syntax des Schlüsselwords *Program* ist:

```
program  
program name
```

In dem folgenden Beispiel wird aus dem Arbeitsblatt die Funktion *Col* des Moduls *ColumnRowExtension* aufgerufen. Die Programmdatei wird über den Namen des Registers qualifiziert.



Komfortabler ist der Aufruf der Funktion *Col* in dem folgenden Beispiel, in dem ein kurzer Programmname angegeben ist. Beim Laden des Moduls wird die Registerseite nach dem Programmnamen statt des Dateinamens benannt.



## 2.2 Uses

Die *uses* Anweisung enthält eine Liste externer Module (Dateinamen), die von den Funktionen in diesem Modul genutzt werden. Die *uses* Anweisung muß der *program* Anweisung unmittelbar folgen. Die einzelnen Modulnamen werden durch Komma getrennt.

Die Syntax der *uses* Anweisung ist:

```
uses module1, module2
```

## 2.3 Define

Die *define* Anweisung deklariert den Namen einer Variablen und weist ihr optional einen Wert zu. Wenn kein Wert zugewiesen wird, ist der Wert der Variable Null. Eine Variable kann nur verwendet werden, wenn der Name zuvor per *define* Anweisung oder in der Parameterliste einer Funktion deklariert wurde.

Die Syntax der *define* Anweisung ist:

```
define Name  
define Name = Value
```

Als Wert kann eine Zahl, eine Variable oder ein mathematischer Ausdruck eingesetzt werden. Das folgende Beispiel definiert die Variable *x* als ein leeres Datenfeld der Größe 20 \* 8 (Zeilen,Spalten).

Beispiel : **define** x[] = [1..20] \* [1..8] **fill** 0

## 2.4 Let

*Let* weist einer Variable einen Ausdruck zu. Die Syntax der *Let* Anweisung ist:

```
let variable = Ausdruck
```

Der Ausdruck bestehen aus einer oder mehreren Konstanten, Variablen oder Zellen eines Datenfeldes. Der Wert kann eine numerischer oder boolscher Ausdruck oder eine Text String sein.

Beispiel:

```
let x = 12
let x = y
let x = (12 + y) * z
let x = sin(45)
let x = "hello"
let x[5] = 16
```

Im letzten Beispiel wird dem Index [5] der Feldvariablen *x* der Wert 16 zugewiesen. Index [5] ist das fünfte Element des Datenfeldes. Beachten Sie, daß das Datenfeld mit Index [1] beginnt, im Gegensatz zu verschiedenen anderen Programmiersprachen, in den das erste Feld mit [0] indiziert wird.

## 2.5 While do

Mit *While* wird die Folge von Anweisungen (statements) zwischen *do* und *end* wiederholt, solange die Aussage der Bedingung (expression) wahr, bzw. ungleich Null ist. Wenn die Aussage falsch bzw. Null ist, wird das Program mit der Anweisung nach *End* fortgeführt.

Die Syntax der *While do* Anweisung ist:

```
while expression do
  statements....
end
```

Beispiel:

```
let i = 0
while i < 100 do
  Statement Sequence....
  let i = i + 1
end
```

## 2.6 If Then

*If Then* bietet die Möglichkeit einer bedingten Programmverzweigung. Die Anweisungen (statements) zwischen *then* und *end* werden nur ausgeführt, wenn der Ausdruck (expression) einen Wert ungleich Null (*TRUE*) liefert. Sonst wird der Anweisungsblock übersprungen und das Program mit der Anweisung nach *end* fortgesetzt.

Die Syntax der *if then* Anweisung ist:

```
if expression then
  statements....
end
```

## 2.7 Else

Die *else* Anweisung ist eine Erweiterung der *if then* Anweisung. Die Syntax der *if then else* Anweisung ist:

```
if expression then
  statements....
else
  statements.
end
```

Wenn der Ausdruck zwischen *if* und *then* Null (*FALSE*) ergibt werden die Anweisungen (statements) zwischen *then* und *else* übersprungen und statt dessen die Anweisungen zwischen *else* und *end* ausgeführt. Wenn der Ausdruck zwischen

*if* und *then* ungleich Null (*TRUE*) ergibt wird der Block zwischen *if* und *then* ausgeführt und der Block zwischen *else* und *end* übersprungen.

## 2.8 Elseif

Mit der Anweisung *elseif* können weitere Bedingungen für eine Programverzweigung programmiert werden. Der auf *elseif* folgende Ausdruck wird nur ausgewertet, wenn die vorherigen *if* und *elseif* Ausdrücke den Wert Null (*FALSE*) ergeben. Wenn der Ausdruck einen Wert ungleich Null (*True*) ergibt, werden die auf *then* folgenden Anweisungen (statements) bis zur nächsten *elseif* oder *else* Anweisung ausgeführt. Dann wird das Programm nach *end* ausgeführt.

Die Syntax der *Elseif* Anweisung ist:

```
if expression then
    statements....
elseif
    statements.
elseif
    statements.
else
    statements.
end
```

## 2.9 Function

Die *function* Anweisung definiert eine Funktion. Eine Funktion ist ein benannter Block von Anweisungen. Sie kann vom Arbeitsblatt, aus anderen Funktionen des Programms, oder aus jedem anderen Programm-Modul namentlich aufgerufen werden. Eine Funktion liefert als Resultat einen einzelnen Wert, ein Datenfeld oder einen Text-String.

Die Syntax der *function* Anweisung ist:

```
function Name (argument, argument.....)
    statements...
end
```

Wenn keine Argumente an eine Funktion übergeben werden, muß in der Deklaration und im Funktionsaufruf eine leere Klammer hinter dem Namen stehen.

Beispiel: **function** Name()

## 2.10 Forward

In einem Programm können Funktionen nur aufgerufen werden, wenn sie vorher deklariert sind. Der Zweck einer *forward*-Deklaration ist es, eine Funktion zu deklarieren die erst weiter unten im Quellcode implementiert ist. Dadurch können andere Funktionen, die vorwärts-deklarierte Routine aufrufen, bevor sie tatsächlich definiert ist. Dieses ist zum Beispiel notwendig, wenn Funktionen sich gegenseitig rekursiv aufrufen.

Die Syntax der *forward*-Anweisung lautet:

```
forward Name
```

## 2.11 Call

Die *call*-Anweisung ruft eine Funktion auf, ohne ein Ergebnis auszuwerten.

Die Syntax der *call*-Anweisung lautet:

```
Call FunctionName (argument, argument...)
```

## 2.12 Result

Die *result* Anweisung gibt einen Wert an die aufrufende Routine zurück. Der Rückgabewert kann eine Zahl, ein boolescher Wert, ein Text-String, ein Datenfeld oder das Resultat einer anderen Funktion sein. Es können konstante Werte oder Variable zugewiesen werden.

Die Syntax der *result* Anweisung lautet:

```
Result = Value
```

## 2.13 Next

Mit der *Next* Anweisung werden einzelnen Elementen von Datenfeldern Werte zugewiesen. Die Besonderheit von *Next* ist, daß kein Index angegeben wird. Der Index wird in der Feldvariablen selbst verwaltet. Mit jeder *Next* Zuweisung wird der interne Index um +1 inkrementiert.

Beispiel: `define x = [1..20]`

```
next x  
next x = 23  
next x = 5.6
```

In dem Beispiel oben wird der Index durch die Anweisung *Next x* initialisiert. Dann wird **23** an **x[0]** und **5.6** an **x[1]** zugewiesen.

Wenn die initialisierte Variable einer anderen Variable zugewiesen oder als Parameter an eine Funktion übergeben wird, wird der Index auch übernommen.

Eine weitere Besonderheit ist, daß bei der Verwendung von *Next* kein Fehler durch Bereichsüberschreitung auftreten kann, weil *Next* automatisch das Datenfeld verlängert, wenn der Index das Ende erreicht hat. Bei sehr großen Datenmengen ist es aber sinnvoller, das Datenfeld vor der Verwendung ausreichend groß zu dimensionieren, weil die nachträgliche Verlängerung des Feldes zusätzliche Rechenzeit in Anspruch nimmt. Bei kleinen Datenmengen bis zu einigen tausend Records ist das meistens unerheblich. In jedem Fall ist es wichtig, daß mindestens

eine Zeile des Datenfelds mit der erforderlichen Anzahl von Spalten und Dimensionen definiert wird.

*Next* prüft, ob die Daten in der Zuweisung mit dem Datenfeld kompatibel sind. Die Anzahl der Dimensionen der zuzuweisenden Daten muß um eins geringer sein als die Dimension des Datenfeldes. In dem Beispiel oben, wurde ein einfacher Wert an ein Element eines eindimensionalen Felds zugewiesen. In dem Beispiel unten, wird ein eindimensionales Feld an ein Element eines zweidimensionalen Felds zugewiesen.

Die Anzahl der Spalten kann voneinander abweichen. In dem Beispiel unten ist *x* als ein zweidimensionales Datenfeld mit 3 Spalten definiert. In der vierten Zeile wird ein einspaltiges Feld mit dem Wert 99 zugewiesen. Die restlichen Elemente werden mit Nullen gefüllt.

In der fünften Zeile wird ein vierspaltiges Feld zugewiesen. Dort wird das vierte Element ignoriert.

Beispiel:

```
define x[] = [1..6]*[1..3] fill 1
next x
next x = [10, 20, 30]
next x = [99]
next x = [22, 33, 44, 55]
```

Inhalt von x:

10	20	30
99	0	0
22	33	44
1	1	1
1	1	1
1	1	1

## 2.14 Index

*Index* ist eine Funktion die als Resultat den aktuellen Index einer Variable liefert (Index der letzten Zuweisung per *Next*).

Beispiel: `i = index(x)`

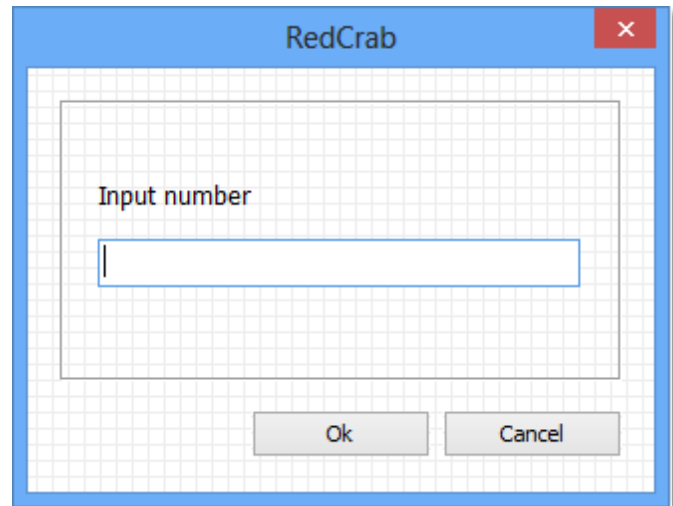


## 3.0 Ein- und Ausgabe

### 3.1 Input

Das Kommando *input* öffnet ein Fenster zur Daten-Eingabe. Dem *input* Statement folgen zwei, oder drei Parameter:

1. Einen Text-String der den Text enthält, der im Fenster über der Eingabezeile angezeigt wird.
2. Den Namen der Variable, der die Eingabe zugewiesen wird.
3. Optional kann als dritter Parameter in einem Text-String ein Wert vorgegeben werden, der in der Eingabezeile angezeigt wird.



Beispiel 1: `input "Input number", x, "123"`

Beispiel 2: `let a = "Input number"`  
`let b = "123"`  
`Input a, x, b`

### 3.2 Display

Das Kommando *display* öffnet ein Fenster zur Anzeige eines numerischen Resultats. Dem *display* Statement folgen zwei Parameter:

1. Der Name der Variable deren Wert angezeigt werden soll.
2. Einen Text-String mit optionaler Formatierungsanweisung.

Beispiel 1: `Let x = 471.2`

`Display x, "Das Resultat lautet: "`

Anzeige im Fenster: „Das Resultat lautet: 471,2“

Beispiel 2: `Let x = 471.2`

`Display x, "Das Resultat lautet: #.##"`

Anzeige im Fenster: „Das Resultat lautet: 471,20“

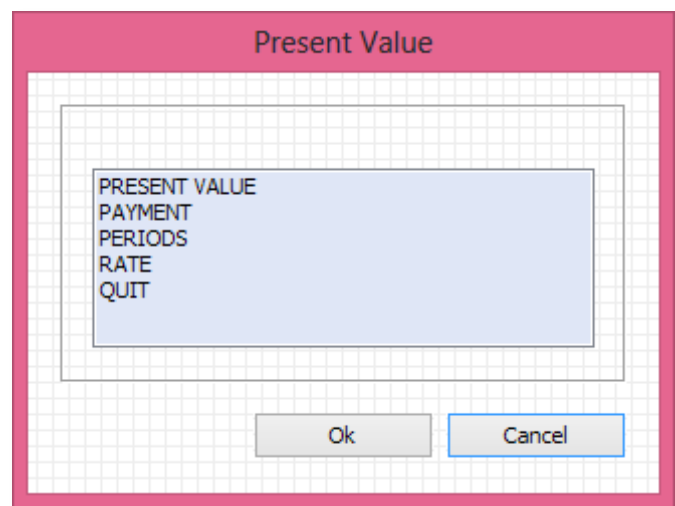
Die Formartierung ist identisch mit der von Result Boxen. Lesen Sie dazu die Beschreibung im User Manual: ***Result Box / Formatierung***.

## 3.3 Menu

Das Kommando ***menu*** öffnet ein Fenster zur Anzeige eines Menüs. Dem ***menu*** Statement folgen eine unterschiedliche Anzahl String Parameter.

Der erste String ist der Titel des Menüs, dann folgt der Name des ersten Menü Item und der Name der Funktion, die von diesem Item aufgerufen wird. Es folgt der Name des zweiten Menü Item und der Name der aufzurufenden Funktion, u.s.w.

Alle Parameter werden durch Komma getrennt und in Anführungszeichen geschrieben.



Beispiel:   Menu "Present Value",  
              "PRESENT VALUE", "PresentVal",  
              "PAYMENT", "PresentPayment",  
              "PERIODS", "PresentPeriods",  
              "RATE", "PresentRate",  
              "QUIT", "Quit"

# 4.0 Debugger

**RedCrab** enthält einen integrierten Debugger der die Fehlersuche und –Behebung in selbstprogrammierten Funktionen unterstützt.

Zur Kontrolle des Debuggers stehen die drei Kommandos **Trace**, **Break** und **Watch** zur Verfügung.

## 4.1 Trace

Mit der **Trace** Anweisung kann der Wert einer Variable in einem laufenden Programm überwacht werden.

Syntax:     **trace** x

**Trace** kann in jeder beliebigen Zeile des Programms eingefügt werden. Nach dem Aufruf von **Trace** wird der Wert der Variable im **Trace**-Fenster der Debugger Leiste angezeigt und bei laufend Program mit jeder Änderung aktualisiert.

! Die Anzeige der Variable verlangsamt das Programm. Die Überwachung einer Variable in einer Schleife kann die Berechnung erheblich verzögern.

! Die **Trace** Anweisung ist nur wirksam, wenn die Debugger Leiste eingeschaltet ist. Bei geschlossenem Debugger wird **Trace** ignoriert und verbraucht keine Rechenzeit.

Tip: Bei einer lange andauernden Berechnung ist es möglich, das Program mit eingeschalteter Debugger Leiste zu starten, damit die **Trace** Funktion initialisiert wird. Während der laufenden Berechnung kann der Debugger geschlossen werden. Der Debugger kann dann von Zeit zu Zeit geöffnet werden, um sich über den Fortschritt der Berechnungen zu informieren.

Lesen Sie hierzu auch die Beschreibung im User-Manual unter **Debugger** Menü.

## 4.2 Break

Mit der **Break** Anweisung kann der Programm Ablauf unterbrochen werden. **Break** kann in jeder beliebigen Zeile des Programms eingefügt werden.

Syntax:     **break**

Das Programm stoppt an der entsprechenden Position und die Werte der lokalen und globalen Variablen werden im Debugger Fenster angezeigt. Die Zeile an der das Programm unterbrochen ist, wird rot markiert. Das Programm kann unter dem Debugger-Menü mit *Step\_Over*, *Step\_Into*, *Run* oder *Ignore\_Break*, oder alternativ mit den Funktionstasten fortgesetzt werden.

F10	-	Step into
F11	-	Step over
F12	-	Run
F12 Shift	-	Run and ignore Break

### *Step Into*

Das Programm führt die nächste Programmzeile aus. Wenn die Programmzeile einen Funktionsaufruf enthält, wird die Funktion aufgerufen und das Programm stoppt in der ersten Zeile der Funktion.

### *Step Over*

Das Programm führt die nächste Programmzeile aus. Wenn die Programmzeile einen Funktionsaufruf enthält, wird die Funktion ausgeführt und das Programm stoppt in der Zeile hinter dem Funktionsaufruf.

### *Run*

Das Programm wird bis zum nächsten Break, oder bis zum Programmende ausgeführt.

### *Ignore Break*

Das Programm wird fortgeführt wie unter **Run** beschrieben, aber der aktuelle **Break Point** wird im weiteren Programmverlauf nicht mehr berücksichtigt. Dieses kann nützlich sein, wenn ein **Break** in einer Schleife programmiert ist und keine weiteren Stops an dieser Position mehr gewünscht werden.

! Die **Break** Anweisung ist nur wirksam, wenn die Debugger Leiste eingeschaltet ist. Bei geschlossenem Debugger wird **Break** ignoriert.

Tip: Wenn nach einem **Break** keine weiteren Programm Unterbrechungen mehr gewünscht werden obwohl noch weitere **Break** folgen, kann die Debugger Leiste geschlossen werden. Das Program wird dann mit dem Menü **Debugger -> Run (F12)** gestartet und läuft ohne Unterbrechung bis zum Ende.

Lesen Sie hierzu auch die Beschreibung im User-Manual unter **Debugger** Menü.

## 4.3 Watch

Mit der **Watch** Anweisung kann der Wert einer Variable überwacht werden.

Syntax: **watch** x

Der angezeigte Wert der Variable wird aktualisiert wenn das Programm per **Break** unterbrochen wird. Die globalen und lokalen Variablen im Sichtbarkeits Bereich der Programm Unterbrechung werden automatisch angezeigt. Eine per Watch überwachte Variable kann außerhalb des Sichtbereichs, zum Beispiel auch in einem anderen Programm Modul definiert sein.

## 5.1 PHP Programm Schnittstelle

*RedCrab* unterstützt die Entwicklung von *PHP* Programmen mit einer integrierten Entwicklungs Umgebung. Mehr Informationen zur Installation und Konfiguration finden Sie im *RedCrab<sup>PLUS</sup>* Handbuch. Dieses Kapitel beschreibt die Programm Schnittstelle zum Daten Transfer zwischen *RedCrab* und *PHP*. Die Schnittstelle verwendet zum Transfer die Standard Ein- und Ausgabe von *PHP* Programmen.

## 5.2 PHP Input

*ReadCrab* sendet Daten zur Standard Eingabe des *PHP* Programms mit der Methode *POST*. Die Daten enthalten ein oder mehrere benannte Argumente. Das erste Argument hat immer den Namen *func*, es enthält den Namen der Funktion die aufgerufen wird. Die nächsten Argumente enthalten die Parameter der Funktion; sie sind benannt mit *arg1*, *arg2*, *arg3* usw.

Das folgende Beispiel zeigt einen Aufruf der Funktion *Add*, mit zwei Parametern, in dem *PHP* Program *math*.

Beispiel:

Funktion Aufruf in *RedCrab*:

```
x = math.Add(a,b)
```

Handhabung des Aufrufs in *PHP*:

```
$fname = $_POST["func"]; // Funktion Name in $fname speichern  
$a1     = $_POST["arg1"]; // Erstes Argument in $a1 speichern  
$a2     = $_POST["arg2"]; // Zweites Argument in $a2 speichern  
$fname($a1,$a2);         // Aufruf der Funktion Add
```

Das Beispiel oben zeigt einen Funktions Aufruf mit zwei numerischen Parametern. Wenn die Argumente Datenfelder anhalten, werden diese von *RedCrab* in *PHP* Format konvertiert.

Das nächste Beispiel zeigt die Handhabung von Argumenten mit Datenfeldern im *PHP* Program.

Beispiel:

#### Eindimensionales Datenfeld

```
1.) $data = $_POST["arg1"];  
    $x    = $data[1];  
    $y    = $data[2];
```

```
2.) $x = $_POST["arg1"][1];  
    $y = $_POST["arg1"][2];
```

#### Zweidimensionales Datenfeld

```
1.) $data = $_POST["arg1"];  
    $x    = $data[1][0];  
    $y    = $data[2][0];
```

```
2.) $x = $_POST["arg1"][0][1];  
    $y = $_POST["arg1"][1][1];
```

## 5.3 PHP Output

Resultate werden von den *PHP* Funktionen über die Standart Ausgabe per *echo* Befehl an *RedCrab* gesendet.

Beispiel: `echo $val;`

Wenn eine Funktion ein Datenfeld als Resultat sendet, muß dieses als String formatiert werden. Das Format ist identisch mit dem Format importierter Textdateien. Das bedeutet, die Tabelle wird Zeile für Zeile gesendet. Spalten werden mit Komma, Zeilen werden mit Semikolon getrennt.

Die *PHP* Datei *RCFieldEcho.php* enthält eine Funktion, die ein Datenfeld zur Ausgabe formatiert. Sie können diese Datei in Ihr Programm importieren oder den Code kopieren.



```
1 <?PHP
2
3 function RCFieldEcho($a)
4 {
5     $cnt = count($a);
6     $mcnt = count($a[0]);
7     for ($i = 0; $i < $cnt; $i++)
8     {
9         if ($mcnt > 1 )
10         {
11             if ($i > 0) echo ' ';
12             RCFieldEcho($a[$i]);
13         }
14         else
15         {
16             if($i > 0) echo ',';
17             echo $a[$i];
18         }
19     }
20 }
21
22 ?>
```