

# Basic für das fischertechnik Robo Interface (Version 0.9.4 Beta) 31.07.2007

## Vorwort

Diese Software entstand aus der Idee heraus, eine Scriptsprache für das Robo Interface zu entwickeln, die auch downloadfähig ist – d.h.: die auch autonom ohne PC im Interface funktioniert. Zwar existiert bereits eine downloadfähige Scriptsprache in Form von C, allerdings ist dies sehr fehleranfällig und für Anfänger gänzlich ungeeignet.

Es ist in Zukunft geplant, die Sourcen zu veröffentlichen – allerdings müssen die erst einmal bereinigt und lesbar gemacht werden ;-)

Ich kann leider nicht garantieren, dass die vorliegende Software völlig fehlerfrei ist. Eine Benutzung geschieht deshalb auf eigenes Risiko, eine Haftung bei Folgeschäden durch die Benutzung dieser Software schließe ich aus.

Momentan hat die Software Beta-Status.

Bitte diese Anleitung komplett durchlesen!

Sollten Teile dieser Beschreibung nicht verständlich sein, bitte ich um Nachricht.

## Historie (Änderungen gegenüber der Vorgängerversion)

- Befehl „edge“ ist nun eine Funktion mit zusätzlichen Parametern
- Ausgaben mit dem Print-Befehl werden nun direkt auf der seriellen Schnittstelle ausgegeben (ohne das Overhead der Firmware-Message-Funktion)
- Hinzugekommen sind put, get() für serielle Kommunikation, len() zur Ermittlung der Länge eines Strings
- Die Ein- und Ausgabefunktionen für die digitalen Ein- und Ausgänge wurden von input() und output auf in () und out geändert, da input() nun für die serielle Eingabe benutzt wird.
- Die Endung der Basic-Dateien wurde von bas auf rib (Robo Interface Basic) geändert, damit die Endung nicht mit anderer Software kollidiert.
- Baud-Befehl ist hinzugekommen
- Der Bytecode kann nun entweder in Flash2 oder in den RAM geladen werden. Der Interpreter verbleibt immer im Flash1 und muss nicht jedesmal mitgeladen werden.

## 1. Wie schreibe und compiliere ich Programme?

Zur Installation des Basic-Pakets die ZIP-Datei einfach in ein Laufwerk entpacken. Es wird ein Verzeichnis „RoboInt\_Basic\_Compiler“ erstellt.

Anschließend die Datei anpassen.exe ausführen. Hierbei werden nur die Verzeichnisse beim „Programmers Notepad“ angemeldet, es wird nichts in der Registrierung des Computers geändert.

**Wird anpassen.exe nicht ausgeführt, sind die Dateipfade nicht richtig definiert und der Aufruf des Compilers aus dem Notepad heraus funktioniert nicht.**

Bei Nutzung eines eigenen Editors ist die Datei anpassen.exe nicht notwendig.

Im Paket ist der kostenlose Editor "Programmers Notepad" enthalten, der auch unter <http://www.pnotepad.org> heruntergeladen werden kann. Er bietet Syntax-Highlighting und Tools für den einfachen Aufruf der benötigten Programme.

Das gesamte Programmpaket benötigt nur User-Rechte zur Installation und kann vollständig entfernt werden, indem der von Programmers Notepad angelegte Ordner „Echo Software“ und der Ordner „RoboInt\_Basic\_Compiler“ gelöscht werden.

An der Dateistruktur des Verzeichnisses "RoboInt\_Basic\_Compiler" sollte nichts verändert werden, da der Editor diese Struktur erwartet und nur dann richtig funktioniert.

### 1.1 Schreiben eines Programms

Starten des Editors mit Doppelklick auf "pn.exe" im Unterverzeichnis „pn“. Beim ersten Aufruf speichert der Editor seine Konfigurationsdaten in "C:\Dokumente und Einstellungen\(\Username)\Anwendungsdaten\Echo Software". Hierbei werden auch die Tools automatisch definiert.

Nun kann das Programm eingegeben oder eines der Beispielprogramme im Ordner "Testprogramme" geladen werden. **Wichtig:** Das Label **main:** muss auf jeden Fall im Programm vorhanden sein, da der Compiler hierdurch mitgeteilt bekommt, wo das Hauptprogramm beginnt.

**Das Programm muss mit der Endung .rib abgespeichert werden. Der Compiler erwartet das Programm als Datei, das Programm muss also unbedingt vor dem Compilieren gespeichert werden.** Wurde das Programm schon einmal gespeichert und wird nur verändert, ist das Programmiers Notepad so voreingestellt, dass es die veränderte Datei bei Druck auf „F5“ automatisch abspeichert. „Programmers Notepad“ ist bereits so installiert, dass der Compiler mit Funktionstasten aufgerufen werden kann, um den schnellen Beginn mit der Basic-Programmierung zu ermöglichen, außerdem werden zur besseren Lesbarkeit die Befehle fett angezeigt („Syntax Highlighting“). Falls man mit Programmers Notepad nicht arbeiten möchte, eignet sich jeder beliebige Texteditor zur Programmeingabe. Hinweise zum Aufruf des Compilers am Ende dieses Kapitels.

## 1.2 Compilieren eines Programms

Mit Klick auf "Tools" und dann auf "[Robolnt] compile" wird das Programm compiliert und gelinkt. Beim Link-Vorgang wird der Interpreter automatisch zum generierten Bytecode hinzugefügt, so dass immer ein lauffähiges Programm entsteht. Dies ist auch der Grund, weshalb die Programme mehr als 32 KByte haben. Diese Datei hat die Endung „.hex“. Im Output-Fenster am unteren Rand wird entweder die erfolgreiche Compilierung oder ein Fehler angezeigt. Bei der Ausgabe "Process Exit Code: 0" ist alles in Ordnung. Der Compiler kann auch mit der Taste "F5" aufgerufen werden. Zusätzliche Optionen (nur für "Fortgeschrittene"):

- compile (Line Numbers) : Fügt Informationen der Zeilennummern hinzu. Bei Runtime-Fehlern wird diese Information benutzt, um die Zeilennummer auszugeben, in der der Fehler passierte.
- compile (Detailed) : Der Compiler gibt alle Zwischenschritte aus, die er beim Übersetzen des Codes benötigt. Dient zum Debuggen des Compilers.

Beim "Process Exit Code: 1" ist ein Fehler beim Compilieren aufgetreten, üblicherweise sollte eine entsprechende Fehlermeldung angezeigt werden, die die Ursache des Fehlers zeigt.

## 1.3 Übertragen eines Programms zum Interface

Während des Compilierens wird ein "hex"-File erzeugt, das den Bytecode enthält. Das Hex-File hat den gleichen Namen, wie der Quelltext.

Durch Aufruf des Tools "Uploader" wird der Bytecode in das RobolInterface hochgeladen. Um so wenige Parameter wie möglich festlegen zu müssen, sucht der Uploader nach dem ersten RobolInterface an der USB-Schnittstelle und benutzt dieses. Sind mehrere Interfaces angeschlossen, kommt es darauf an, welches von Windows als „erstes“ Interface in der USB-Reihenfolge definiert wird. Soll bei mehreren RobolInterface ein bestimmtes angesprochen werden, so müssen die anderen abgeschaltet werden.

Der Bytecode kann entweder in das RAM oder in den Flash-Speicher 2 geladen werden. Während des Upload wird automatisch überprüft, ob sich im Flash-Speicher 1 der Interpreter befindet. Ist dies nicht der Fall, wird er automatisch nachgeladen.

Der RAM-Speicher ist für Versuche und während der Entwicklung eines Programms besser geeignet, da der Flash-Speicher eine begrenzte Anzahl Schreibzyklen hat.

Der Interpreter ist so eingestellt, dass er zuerst prüft, ob sich ein Programm im RAM befindet. Ist dort eines vorhanden, wird es ausgeführt, auch dann, wenn in Flash-Speicher 2 ein Programm ist.

## 1.4 Ausführen des Programms

Das Programm „Control“ verbindet sich mit dem ersten RobolInterface, das auf der USB-Schnittstelle gefunden wird.

**Achtung: Die Firmware löscht den RAM, wenn ein Programm im Flash ausgeführt wird.**

**Programme im RAM haben immer Vorrang, auch wenn Flash2 gestartet wird, überprüft der Interpreter zuerst den RAM und führt ein dort gespeichertes Programm aus.**

Das Control-Programm öffnet sich bei Verwendung des Uploaders automatisch.

Momentan noch nicht möglich, aber geplant, ist der automatische Programmstart für Programme in Flash2.

## 1.5 Anzeige von Meldungen des Interface

Der Basic-Interpreter gibt Status- und Fehlermeldungen über die serielle Schnittstelle des Robo-Interface aus. Diese Funktion kann genutzt werden, indem das mitgelieferte Kabel an den PC angeschlossen wird. Mit einem beliebigen Terminal-Programm (eingestellt auf 38400 Baud, 8 Bit, 1 Stopbit, keine Parity) können dann die Ausgaben auf dem PC angezeigt werden. Ebenfalls werden Daten, die mit dem "print" Befehl ausgegeben werden, angezeigt, auch Runtime-Fehler werden über die serielle Schnittstelle ausgegeben. Beim normalen Programmende blinkt die rote Fehler-LED des Interface 1x. Sollte ein Runtime-Fehler auftauchen, schaltet die rote LED auf Dauerblinken.

## 1.6 Aufruf des Compilers

Falls man den Compiler von Hand oder aus einem anderen Editor heraus benutzen möchte, ist dies ohne weiteres durch den direkten Aufruf möglich.

Der Compiler erwartet bei Aufruf insgesamt 3 Parameter:

compiler [dateiname] [Zeilennummern] [Debug-Option]

|                 |  |
|-----------------|--|
| [dateiname]     | Dateiname des Quelltextes, die Endung .rib darf nicht angegeben werden, sie wird automatisch ergänzt.  |
| [Zeilennummern] | 0 oder 1. Bei 1 werden Zeilennummern in den Bytecode einkodiert, bei einem Runtime-Fehler wird dann die Zeilennummer des Source angegeben, in der der Fehler vorkommt. Der Bytecode wird dadurch allerdings größer.  |
| {Debug-Option}  | 0, 1 oder 2. Bei 0 werden nur die nötigsten Ausgaben während des Compilierens ausgegeben, bei 1 etwas mehr und bei 2 sehr ausführlich. Die Optionen 1 und 2 sind eigentlich nur zum Debuggen des Compilers gedacht, aber wer möchte, kann sich damit genau anschauen, wie der Compiler den Quellcode analysiert. |

Beispiel:

compiler testprogramm 0 0

## 1.7 Alternativer Upload

Alternativ kann der Upload auch mit dem von der Fa. Knobloch bereitgestellten FtLoader erfolgen. Hierbei muss die Datei „interpreter.rom“ geladen werden (ist automatisch für Flash1 adressiert). Anschließend kann der Bytecode als .hex nachgeladen werden, er ist für das RAM adressiert.

## 2. Funktionsprinzip

Das Basic besteht aus zwei Komponenten:

- Einem Compiler, der den Quelltext in einen Bytecode umsetzt
- Einem Interpreter, der den Bytecode ausführt.

Bei der Planung des Basic habe ich viele verschiedene Möglichkeiten geprüft und – meines Erachtens – die bestmögliche verwirklicht. In der Anfangsphase sollte es ein reinrassiger Interpreter werden, dies hätte den Vorteil gehabt, dass man theoretisch ohne einen PC ausgekommen wäre (über die serielle Schnittstelle wäre der Anschluss eines Displays und einer Tastatur durchaus möglich). Allerdings hätte dies bedeutet, dass der Prozessor des Interface sehr stark mit den Übersetzungsaufgaben beschäftigt gewesen wäre. Eine Zwischenlösung ist die Umsetzung von Befehlen in Tokens, d.h. der Klartext der Befehle wird in Bytecode umgesetzt (dies war z.B. das Prinzip des C64-Basic). Bei dieser Methode bleiben aber immer noch: Berechnung der Sprungadressen, Berechnung der Variablenadressen, Syntaxprüfung etc. Gerade die Syntaxprüfung ist ein Punkt, der für einen vom Interpreter getrennten Compiler sprach.

In dem nun vorliegenden Programm sind im Bytecode alle Adressen von Variablen, Funktionen und sonstigen Sprüngen vorberechnet und auch die Konstanten liegen nicht als Ascii, sondern im Binärformat vor, was die Ausführungszeiten positiv beeinflusst.

Theoretisch wäre es auf Grundlage des Bytecodes auch möglich, andere Sprachen (z.B. Pascal o.ä.) zu realisieren.

Beim Übersetzungsvorgang des Compilers wird ein .hex-File erzeugt, das mit dem von ft gelieferten „FtLoader“ in das Robo Interface übertragen werden kann. Das .hex-File beinhaltet den Interpreter und den Bytecode, ist also ohne weitere Downloads funktionstüchtig.

## 2.1 Features

In den letzten Monaten hat sich die Liste der Features von Woche zu Woche geändert – meistens wurde sie verlängert ;-). Mittlerweile bin ich der Überzeugung, dass man mit diesem Basic durchaus ein interessantes Werkzeug zur Verfügung hat. Bei der Realisierung bin ich hauptsächlich von meinen Bedürfnissen ausgegangen – falls jemand ein Feature vermisst, das realisierbar scheint, kann ich nochmal etwas nachlegen.

Eine kurze Liste der realisierten Features (ausführliche Information gibt es in der Befehlsreferenz)

- Multitasking mit bis zu 10 Tasks
- lokale Variablen innerhalb von Tasks und Funktionen

- Funktionen mit Parametern (auch Rückgabe)
- Funktionen rekursiv aufrufbar
- Funktionen mehrfach aus verschiedenen Tasks aufrufbar (reentrant)
- Ausgabe von Zahlen und Texten über die serielle Schnittstelle zum Debuggen
- Arrays mit bis zu zwei Dimensionen (nur global möglich)
- Datentypen Byte, Integer, Long, Float, Double und String
- Trigonometrische Funktionen mit Double-Genauigkeit
- Unterstützung der Hardware des Robo Interface mit entsprechenden Befehlen

## 2.2 Multitasking

Das mit Sicherheit interessanteste Feature ist das Multitasking. Nachdem ich es in C aufgegeben habe, weil es sehr viel Durcheinander auf dem Stack gegeben hat, war die Realisierung innerhalb eines Interpreters geradezu einfach. Die Tasks (bis zu 10 + Hauptprogramm) werden scheinbar gleichzeitig ausgeführt und auch Wartebefehle haben darauf keinen Einfluss.

Die einzige Einschränkung: Für einen Taskwechsel muss der mathematische Stack leer sein. Vor allem bei rekursivem Aufruf von Funktionen kommt es vor, dass ein Berechnungsergebnis auf dem mathematischen Stack verbleibt, dies hat zur Folge, dass die komplette Rekursion erst aufgelöst werden muss, bevor der nächste Task ausgeführt werden kann. Diese Eigenheit lässt sich ohne weiteres durch eine Erweiterung lösen, allerdings sehe ich es eher als Feature, weil man hierdurch eine Art Priorisierung erreichen kann.

## 2.3 Globale und lokale Variablen

Neben den globalen Variablen, die im gesamten Programm sichtbar sind, können für jede Funktion und jeden Task separat lokale Variablen definiert werden. Die lokalen Variablen sind nur innerhalb des Tasks / der Funktion sichtbar, in dem / der sie definiert wurden, nicht darüber hinaus.

## 2.4 Funktionen mit Parametern

Funktionen können mit oder ohne Parameter definiert werden. Werden Parameter definiert, sind diese automatisch als lokale Variable innerhalb der Funktion definiert. Ein Rückgabeparameter wird mit dem return Befehl übergeben.

Dadurch, dass lokale Variablen für jeden Task getrennt gespeichert werden, können Funktionen aus mehreren Task unabhängig voneinander und gleichzeitig aufgerufen werden. Die Aufrufe beeinflussen sich nicht gegenseitig. In diesem Zusammenhang ist aber Vorsicht bei globalen Variablen geboten.

Die restlichen Features sind im Kapitel 2 besprochen.

## 2.5 Bekannte Bugs

Wo Licht ist, ist auch Schatten – es gibt noch einige Bugs, die mir zwar bekannt sind, aber bei denen momentan einfach die Zeit fehlt.

Eine aktuelle Liste ist im Text-File „bekannte\_bugs.txt“ abgelegt. Diese Liste sollte unbedingt gelesen werden, bevor man mit dem Programmieren anfängt. Ansonsten bin ich natürlich immer froh, wenn mir Bugs mitgeteilt werden.

## 3. Programmieren in BASIC

Ich gehe davon aus, dass die grundlegenden BASIC-Begriffe und Programmiertechniken bekannt sind. Die wichtigsten Unterschiede zu einem Standardbasic bestehen in den Elementen für strukturierte Programmierung und darin, dass es keine Zeilennummern gibt. Ferner müssen – BASIC-untypisch – Variablen im Voraus definiert werden. Dies hat seine Begründung darin, dass der Compiler schlichtweg einfacher ist, wenn man Variablen deklariert.

Bei den meisten mir bekannten BASIC-Dialekten konnte man kaum unterschiedliche Datentypen verwenden, bestenfalls kann man zwischen Integer (= ganze Zahlen) und Float (= Gleitkommazahl) wählen. Beim hier vorliegenden BASIC hat man die Wahl zwischen Byte, Int, Long, Float und Double. Warum diese Vielfalt? Ich möchte eine möglichst optimale Ausnutzung von Speicher und Prozessor erreichen. Wenn z.B. ein Array mit 1000 Elementen definiert wird, braucht es als long 4000 Bytes, als byte nur 1000 Bytes.

### 3.1 Datentypen:

**Byte:**

Ganze Zahlen mit einem Wertebereich von -128 bis +127. Benötigt 1 Byte (8 Bit).

**Int:** (Integer)

Ganze Zahlen mit einem Wertebereich von -32768 bis +32767. Benötigt 2 Byte (16 Bit).

**Long:** (long Integer)

Ganze Zahlen mit einem Wertebereich von -2147483648 bis +2147483647. Benötigt 4 Byte (32 Bit).

**Float:** (Gleitkommazahl)

Gleitkommazahlen mit einem Wertebereich von  $(+/-)3,4^{+/-}10^{38}$ . Benötigt 4 Byte (32 Bit). Die Genauigkeit liegt bei max. 7 Ziffern.

**Double:** (Gleitkommazahl)

Gleitkommazahlen mit einem Wertebereich von  $(+/-)1,7^{+/-}10^{308}$ . Benötigt 8 Byte (64 Bit). Die Genauigkeit liegt bei max. 15 Ziffern.

Vor der Programmierung sollte man sich im Klaren sein, welche Variablen welchen Zahlenbereich annehmen können und dementsprechend einen Datentyp wählen. Die Benutzung von float und double sollte man so weit als möglich vermeiden, denn Gleitkommazahlen verbrauchen überdurchschnittlich viel Rechenleistung.

### 3.2 Rechenoperationen und Zahlen

Bei den Rechenoperationen ist (fast) alles erlaubt. Dieser Teil des Compilers hat mich einige graue Haare gekostet, aber scheint nun zu funktionieren.

Es gilt: Punkt vor Strich, Klammern können auch gesetzt werden.

+ Addieren  
- Subtrahieren  
\* Multiplizieren  
/ Dividieren  
^ Potenzieren  
& Bitweise UND (nur für long, bei double ist das Ergebnis immer 0)  
| Bitweise OR s.o.  
! Bitweise XOR s.o.

&, | und ! funktionieren auch bei logischen Operationen (if, while etc.)

Funktionen:

**sin()** sinus  
**cos()** cosinus  
**tan()** tangens  
**cot()** cotangens  
**asin()** Umkehrfunktion zu sin()  
**acos()** Umkehrfunktion zu cos()  
**atan()** Umkehrfunktion zu tan()  
**acot()** Umkehrfunktion zu cot()

Es gibt momentan einen bekannten Fehler im Parser:

Ist innerhalb einer Konstanten ein nicht für Rechenoperationen reserviertes Zeichen (z.B. 4t5), wird kein Fehler ausgegeben. Die Zahl wird bis zum Vorkommen des Zeichens ausgewertet, der Rest wird verworfen. Im obigen Beispiel würde die Zahl also als 4 interpretiert.

Zu beachten ist: Bei den Rechenoperationen müssen die unterschiedlichen Zahlentypen "gecastet", also angepasst werden, sonst kann man long (Ganzzahl) und double (Gleitkomma) nicht miteinander verknüpfen. Dies macht der Interpreter automatisch. Dabei setzt sich immer das genauere (größere) Zahlenformat durch. Die Reihenfolge: Byte – Integer – Long – Float – Double.

Kommt es bei dieser Anpassung zu Bereichsüberschreitungen, so zum Beispiel bei der Konvertierung von Long in Byte, wird der jeweils maximale Wert übernommen, es wird keine Fehlermeldung ausgegeben.

Beispiel:

|      |         |      |
|------|---------|------|
| Long | wird zu | Byte |
| 1374 |         | 127  |
| -462 |         | -128 |

Beispiele für mathematische Ausdrücke:

$a = (b+c)*(d+e)$

$a = 5*(b+8)*(-c-d)$

Der Compiler nimmt dem Interpreter bereits die Arbeit ab, die Zahlen aus dem Klartext in ein dem Rechner verständliches Format umzuwandeln. Konstanten werden direkt im Bytecode abgespeichert, wobei ganze Zahlen 4 Bytes benötigen, Gleitkomma 8 Bytes. Ist in einer Konstanten ein "." (Dezimalpunkt) vorhanden, wird sie als double behandelt, sonst als long. Wenn man von vornherein weiß, dass eine Berechnung in double ausgeführt wird, kann man die Laufzeit optimieren, indem man den Compiler dazu zwingt, eine double abzuspeichern. Dann muss der Interpreter kein Typcasting mehr durchführen.

Es ist geplant, durch Bereichsprüfung bei den Konstanten auch die fehlenden Datentypen byte, integer und float zu implementieren.

1234    long - Konstante

1234.0    double

1234.0E4    double, entspricht 12340000

1234E4    long, das E wird nicht interpretiert, entspricht 1234

### 3.3 Strings

Strings (Texte) sind mit dem RobotInt-Basic auch möglich, allerdings mit der Einschränkung, dass sie nicht dynamisch gespeichert werden, sondern einen festen Speicherbereich zugewiesen bekommen. Man muss also die maximale Größe des Strings angeben.

Ein String wird ähnlich behandelt, wie ein Array, deshalb geschieht auch die Definition des Strings mit dem **DIM** – Befehl:

```
dim text (20) string
```

Dies definiert eine Variable mit dem Namen „text“ als String mit maximal 20 Zeichen Länge. Der String darf während des Programmablaufs auch kürzer sein.

Mit Strings kann nur eine Rechenoperation ausgeführt werden, die Addition +.

Um Strings umzuwandeln, zu verändern stehen Funktionen zur Verfügung:

|   |  |
|---|--|
| <code>zahl = <b>asc</b>(zeichen)</code>                 | Wandelt ein Textzeichen in den entsprechenden Ascii-Code um.     |
| <code>zeichen = <b>chr</b>(zahl)</code>                 | Wandelt einen Ascii – Code in ein entsprechendes Textzeichen.    |
| <code>string = <b>left</b>(string1,anzahl)</code>       | Ergibt einen Teilstring von string1 der Länge anzahl von links.  |
| <code>string = <b>right</b>(string1, anzahl)</code>     | Ergibt einen Teilstring von string1 der Länge anzahl von rechts. |
| <code>string = <b>mid</b>(string1,anfang,anzahl)</code> | Teilstring von string1, beginnend bei anfang, länge = anzahl.    |
| <code>zahl = <b>val</b>(string)</code>                  | Wandelt einen String in eine Zahl                                |
| <code>zahl = <b>valf</b>(string)</code>                 | wie oben, für Gleitkommazahlen                                   |
| <code>string = <b>str</b>(zahl)</code>                  | Wandelt eine Zahl in einen String                                |
| <code>string = <b>strf</b>(zahl)</code>                 | wie oben, für Gleitkommazahlen                                   |
| <code>zahl = <b>len</b>(string)</code>                  | Gibt die tatsächliche Länge des Strings zurück                   |

Um Strings zuzuweisen, wird der Text innerhalb Anführungszeichen angegeben.

```
Text = "Dies ist ein Test"
```

(Gestrichen: Wird die vordefinierte Länge eines Strings überschritten, wird ein Runtime-Fehler ausgelöst.)

Der Stringstack ist 127 Zeichen groß, d.h. Additionen von Strings dürfen nicht über eine Länge von 127 Zeichen gehen.

**Es wird bei Überschreitung kein Runtime-Fehler ausgelöst. Zu lange Strings werden einfach abgeschnitten. Dies gilt auch bei Überschreitung der Länge von Stringvariablen.**

### 3.4 Reihenfolge der Definitionen

Der Compiler akzeptiert die verschiedenen Definitionen nur in einer bestimmten Reihenfolge.

1. Globale Variablen
2. Funktionen (mit eigenen globale Variablen)
3. Tasks (mit eigenen globale Variablen)
4. Hauptprogramm (**main:**)

Grundsätzlich muss eine Funktion definiert sein, bevor sie aufgerufen werden kann. Ein rekursiver Aufruf (die Funktion ruft sich selbst auf) ist deshalb möglich, da die Definition vor dem eigentlichen Aufruf innerhalb der Funktion steht.

Die Definitionen von Tasks und Funktionen können miteinander vertauscht werden, sollen Funktionen aus Tasks heraus aufgerufen werden, müssen die Funktionen wiederum vor dem Task definiert sein.

Globale Variablen müssen direkt nach der Kopfzeile der Funktion / des Tasks definiert werden. Hierzu gibt es im Anhang noch einige Beispiele.

Derzeit muss der gesamte Quellcode in einer einzigen Datei vorliegen, eine Aufsplittung ist geplant, allerdings habe ich noch keine Lösung, wie in diesem Fall Zeilennummern bei Fehlern behandelt werden könnten.

## 4 Befehlsreferenz

### 4.1 Übersicht (nach Funktion sortiert)

#### 4.1.1 Ein- und Ausgabe

|                               |  |
|-------------------------------|--|
| <b>motor</b> nummer, wert     | Motor (nummer) wird mit Stufe (wert) angesteuert               |
| <b>out</b> nummer, wert       | Ausgang (nummer) wird mit Stufe (wert) angesteuert             |
| wert = <b>in</b> (nummer)     | Abfrage des Eingangs (nummer), Rückgabe 0 oder 1               |
| wert = <b>analog</b> (nummer) | Abfrage des analogen Eingangs (nummer)                         |
| wert = <b>ir</b>              | Abfrage des Infrarot-Eingangs                                  |
| <b>print</b> Ausdruck         | Ausgabe auf die serielle Schnittstelle des Robot               |
| <b>put</b> wert               | Ausgabe auf die serielle Schnittstelle des Robot (ein Zeichen) |
| wert = <b>get</b> ()          | Lesen eines Zeichens von der seriellen Schnittstelle           |
| string = <b>input</b> ()      | Lesen einer Zeile von der seriellen Schnittstelle              |
| <b>baud</b> wert              | Einstellen der Baudrate der seriellen Schnittstelle            |

#### 4.1.2 Programmsteuerung

|  |   |
|--|---|
| <b>if</b> (Bedingung) ... <b>else</b> ... <b>end</b> | Verzweigung                             |
| <b>repeat</b> .... <b>until</b> (Bedingung)          | Schleife                                |
| <b>while</b> (Bedingung) <b>end</b>                  | Schleife                                |
| <b>select</b> .... <b>case</b> ... <b>end</b>        | Mehrfache Verzweigung                   |
| <b>edge</b> nummer, typ, time-out                    | wartet auf Ereignis an einem Eingang    |
| <b>for</b> ... <b>next</b>                           | Schleife                                |
| <b>function</b> (....)                               | Definiert eine Funktion (Unterprogramm) |
| <b>waitms</b> zeit                                   | Wartet die Zeit (angegeben in ms) ab.   |

#### 4.1.3 Variablendefinition

|                                       |  |
|---------------------------------------|--|
| <b>var</b> name typ                   | Definiert einen Variable                                 |
| <b>dim</b> name (Anzahl) typ          | Definiert ein eindimensionales Feld, Sonderfall: String. |
| <b>dim</b> name (Anzahl1,Anzahl2) typ | Definiert ein zweidimensionales Feld.                    |

#### 4.1.4 Multitasking

|                   |                         |
|-------------------|-------------------------|
| <b>task</b> name  | Definiert einen Task    |
| <b>start</b> name | Startet den Task „name“ |

**stop** name  
**kill** name

Stoppt den Task „name“ (Pause)  
Beendet den Task „name“

#### 4.1.5 Mathematische Funktionen

|                               |                               |
|-------------------------------|-------------------------------|
| ergebnis = <b>sin</b> (wert)  | Berechnet den Sinus.          |
| ergebnis = <b>cos</b> (wert)  | Berechnet den Cosinus.        |
| ergebnis = <b>tan</b> (wert)  | Berechnet den Tangens.        |
| ergebnis = <b>cot</b> (wert)  | Berechnet den Cotangens.      |
| ergebnis = <b>asin</b> (wert) | Umkehrfunktion für Sinus.     |
| ergebnis = <b>acos</b> (wert) | Umkehrfunktion für Cosinus.   |
| ergebnis = <b>atan</b> (wert) | Umkehrfunktion für Tangens.   |
| ergebnis = <b>acot</b> (wert) | Umkehrfunktion für Cotangens. |

#### 4.1.6 Stringfunktionen

|   |  |
|---|--|
| string = <b>left</b> (text, anzahl)       | Gibt (anzahl) Zeichen von links an zurück      |
| string = <b>right</b> (text, anzahl)      | Gibt (anzahl) Zeichen von rechts an zurück     |
| string = <b>mid</b> (text, start, anzahl) | Gibt (anzahl) Zeichen von (start) an zurück    |
| zahl = <b>val</b> (string)                | Wandelt einen String in eine Zahl (Ganzzahl)   |
| zahl = <b>valf</b> (string)               | Wandelt einen String in eine Zahl (Gleitkomma) |
| string = <b>str</b> (zahl)                | Wandelt eine Zahl in einen String (Ganzzahl)   |
| string = <b>strf</b> (zahl)               | Wandelt eine Zahl in einen String (Gleitkomma) |
| string = <b>chr</b> (zahl)                | Wandelt eine Zahl in einen Buchstaben          |
| zahl = <b>asc</b> (string)                | Wandelt einen Buchstaben in eine Zahl          |

#### 4.2 Ausführliche Referenz (alphabetisch)

##### <Ausdruck>

Ein Ausdruck kann aus Variablen, Konstanten oder Kombinationen davon bestehen (z.B. eine komplette Berechnungsformel). Der jeweilige Befehl verwendet dann das Ergebnis der Berechnung als Parameter. Ebenfalls erlaubt sind Vergleiche, logische Verknüpfungen.

Gültige Ausdrücke:

10      Konstante  
a      Variable.  
a+5      Formel  
(a+5)\*(b+2)  
a = 0  
b > 5  
(a+b) = (c \* d)  
**in** (x) = 0  
**sin**(0) > 1  
b **and** 2

Bei Bedingungen z.B. b>5 ist der Wert des Ausdrucks 1, wenn die Bedingung erfüllt ist, ansonsten 0.  
Zuweisungen wie

a = b>5

sind erlaubt, a wird eine 1 zugewiesen, wenn b > 5 erfüllt ist, ansonsten wird a eine 0 zugewiesen.

##### <variable>

Ein Variablenname kann aus bis zu 15 Zeichen bestehen. Der Variablenname muss mit einem Buchstaben beginnen und darf auch Sonderzeichen (z.B. Umlaute) enthalten. Die Groß- / Kleinschreibung wird bei den Variablenamen nicht berücksichtigt, die Variable „Zähler“ und „zähler“ werden nicht unterschieden.



### **<Ergebnis> = acos(<Ausdruck>)**

Berechnet den Arcuscosinus von <Ausdruck>. Umkehrfunktion von **cos()**.  
Das Ergebnis ist vom Typ double.

### **<Ergebnis> = acot(<Ausdruck>)**

Berechnet den Arcuscotangens von <Ausdruck>. Umkehrfunktion von **cot()**.  
Das Ergebnis ist vom Typ double.

### **<Ergebnis> = analog(<Ausdruck>)**

Fragt einen der analogen Eingänge des Robo Interface ab. <Ausdruck> gibt die Nummer des Analogeingangs an. Um die Programmierung zu vereinfachen, sind Konstanten vordefiniert. Eine Liste der Konstanten finden man im Anhang.

### **<Ergebnis> = asin(<Ausdruck>)**

Berechnet den Arcussinus von <Ausdruck>. Umkehrfunktion von **sin()**.  
Das Ergebnis ist vom Typ double.

### **<Ergebnis> = atan(<Ausdruck>)**

Berechnet den Arcustangens von <Ausdruck>. Umkehrfunktion von **tan()**.  
Das Ergebnis ist vom Typ double.

### **baud <Ausdruck>**

Setzt die Baudrate der seriellen Schnittstelle auf den Wert <Ausdruck>.  
Es sind nahezu beliebige Werte einstellbar, sinnvoll sind Standardwerte:  
1200, 2400, 4800, 9600, 19200, 38400

### **<Ergebnis> = cos(<Ausdruck>)**

Berechnet den Cosinus von <Ausdruck>.  
Das Ergebnis ist vom Typ double.

### **<Ergebnis> = cot(<Ausdruck>)**

Berechnet den Cotangens von <Ausdruck>.  
Das Ergebnis ist vom Typ double.

### **dim <variable> (<Konstante>[,<Konstante>]) <typ>**

Definiert ein Array (Feld). Felder sind immer global und müssen an der entsprechenden Stelle im Programm (direkt am Anfang) definiert werden. Arrays können ein- oder zweidimensional sein. Gerade bei Arrays ist der Speicherbedarf sehr groß, deshalb sollte der Typ des Arrays sorgfältig gewählt werden.  
Beispiele:

```
dim positionen (100,4) int  
dim werte (20) float
```

**Sonderfall: Strings**

**dim text (20) string**

Strings dürfen nur eine Dimension haben.

**edge <Ausdruck1>, <Ausdruck2>[, <Ausdruck3>]**

Ausdruck1 = Nummer des abzufragenden Digital-Eingangs

Ausdruck2 = Typ (-1,0 oder 1, vordefinierte Konstanten #down, #any, #up)

Ausdruck3 = Timeout in ms (optional)

Wartet auf eine Flanke am angegebenen Eingang. Als Typ kann #any (irgendeine), #up (steigend, also 0->1) oder #down (fallend, also 1->0) angegeben werden.

Zusätzlich kann mit Timeout die maximale Wartezeit in Millisekunden auf die Flanke angegeben werden.

Dies ist nützlich, um zum Beispiel einen blockierten Motor zu erkennen.

Der maximale Wert für Timeout ist 65535.

**for <variable> = <Ausdruck 1> to <Ausdruck 2> [step <Ausdruck 3>]**

**<Anweisungsblock>**

**next**

Führt einen Anweisungsblock ein oder mehrfach aus, bis die Endbedingung <variable> = <Ausdruck 2> erreicht ist. „step“ gibt die Schrittweite an, wird „step“ weggelassen, wird +1 angenommen.

For-Next Schleife. Die Variable <variable> wird von <Ausdruck 1> bis <Ausdruck 2> mit der Schrittweite <Ausdruck 3> durchgezählt.

Bei negativen <step> kann auch rückwärts gezählt werden. Die Angabe von <step> ist optional, wird sie weggelassen, wird automatisch 1 angenommen. Hinter next wird nicht, wie teilweise üblich, die Zählvariable angegeben - sie wird automatisch zugewiesen.

(Einschalten der Motoren 1 bis 4)

**for x=1 to 4**

**motor x,8**

**next**

(Zählen von 10 bis 0 rückwärts)

**for x=10 to 0 step -1**

**<Anweisungsblock>**

**next**

(Verschachtelte Schleifen)

**for x=1 to 10 step 2**

**for y=20 to 25**

**<Anweisungsblock>**

**next**

**next**

(Schrittweiten kleiner als 1)

**for x = 1.0 to 1.5 step 0.1**

**<Anweisungsblock>**

**next**

Wichtig: Der Anweisungsblock wird auf jeden Fall mindestens einmal ausgeführt, auch wenn die Schleifenbedingung nicht erfüllt ist.

**function <Funktionsname> [( <variable> <typ> , ...) [typ]]**

Definiert eine Funktion mit dem entsprechenden Namen.

Eine Funktion kann keine oder bis zu vier Übergabeparameter haben und / oder bis zu einen Rückgabeparameter.

Beispiele:

**function** test

**function** test1 (a **long**,b **float**)

**function** test2 (a **long**) **long**

Definiert eine Funktion ohne Parameter

Definiert eine Funktion mit zwei Übergabeparametern

Definiert eine Funktion mit einem Übergabe- und einem Rückgabeparameter

Beispiel für eine Funktion mit Parametern:

**function** summe (a **long**, b **long**) **long**

**return** a+b

Der Aufruf dieser Funktion:

x = summe (10,20)

Beispiel für eine rekursive Funktion (gesamtes Programm)

**var** x **long**

**function** summe (parameter **long**) **long**

**var** a **long**

**var** b **long**

**if** parameter <= 1

    a = 1

**else**

    b = summe (parameter – 1)

    a = parameter + b

**end**

**return** a

**main:**

**print** summe (10)

**Tipp:** Es fällt auf, dass im **else** – Teil des Unterprogramms unnötigerweise die Variable b benutzt wird. Kürzer könnte es heißen:

**else**

  a = parameter + summe(parameter – 1)

**end**

Dies hat allerdings den Nachteil, dass hierdurch ein Taskwechsel verhindert wird, weil die Variable parameter zur Berechnung auf den mathematischen Stack gelegt wird. Erst nach Rückkehr der Funktion (also nach Ausführung aller Rekursionen) ist der mathematische Stack bereinigt. Außerdem hat der mathematische Stack nur 10 Speicherplätze, eine Berechnung von summe (11) wäre nicht möglich. Durch das Speichern des Ergebnisses in der lokalen Variable b werden diese Effekte verhindert.

Diese Eigenart kommt bei normaler Nutzung kaum zu tragen, sollte es zu überdurchschnittlich vielen Problemen führen, kann sie durch eine Änderung am Interpreter eliminiert werden.

**<Ergebnis> = get()**

Liest ein einzelnes Zeichen von der seriellen Schnittstelle. Es wird nicht auf ein Zeichen gewartet, sondern die Funktion gibt den Wert -1 zurück, falls kein Zeichen von der seriellen Schnittstelle empfangen wurde. Das Ergebnis ist vom Typ int.

**If <Vergleich>**

**<Anweisungsblock1>**

**[else**

**<Anweisungsblock2>]  
end**

„Falls ... dann ... sonst ...“

Falls die im Vergleich angegebene Gleichung wahr ist, wird Anweisungsblock 1 ausgeführt, ansonsten (sofern vorhanden) Anweisungsblock 2.

Ein Anweisungsblock kann aus einem oder mehreren Anweisungen bestehen. Es dürfen beliebig viele if-else-end Anweisungen ineinander verschachtelt werden.

Beispiel A:

```
if a = 0
  motor 1,0 // a ist gleich 0
else
  motor 1,8 // a ist ungleich 0
end
```

Falls a = 0, wird der Motor abgeschaltet, bei allen anderen Werten von a wird er eingeschaltet.

Beispiel B:

```
if a = 0
  motor 1,0
end
```

Falls a = 0, wird der Motor abgeschaltet, bei allen anderen Werten von a passiert nichts (es wird der nächste Befehl nach end ausgeführt).

**<Ergebnis> = in (<Ausdruck>)**

Fragt die digitalen Eingänge des Robo Interface ab. <Ausdruck> kann Werte zwischen 1 und 32 annehmen, das Ergebnis ist entweder 0 (Eingang nicht betätigt) oder 1 (Eingang betätigt).

Beispiele:

a = in (8)                      Fragt Eingang 8 ab und speichert das Ergebnis in a

```
if in (1) = 1                      Schaltet den Motor 1 ein, wenn Eingang 1 betätigt wird.
  motor 1,8
end
```

**<Ergebnis> = input()**

Liest einen String von der seriellen Schnittstelle. Der Befehl wartet so lange, bis ein Zeilenvorschub empfangen wird.

**<Ergebnis> = ir**

Fragt den Infrarot-Eingang des Robo Interface ab. Ist keine Taste auf der Fernbedienung gedrückt, ist das Ergebnis 0.

Um Abfragen zu vereinfachen, sind Konstanten vordefiniert. Eine Liste der Konstanten findet man im Anhang.

**kill <taskname>**

Beendet einen Task und initialisiert ihn neu. Bei einem weiteren start <taskname> startet der Task neu.

**motor <Ausdruck 1>,<Ausdruck 2>**

Steuert einen Motorausgang des Robo Interface an. <Ausdruck 1> gibt die Nummer des Motors an (1..16), <Ausdruck 2> die Drehrichtung und Geschwindigkeit. <Ausdruck 2> kann Werte zwischen -8 und 8 annehmen, bei negativen Werten dreht der Motor links herum, bei positiven rechts herum. Bei 0 wird der Motor abgeschaltet. Größere oder kleinere Werte bei den Parametern werden ignoriert.

Beispiele:

|                  |  |
|------------------|--|
| <b>motor</b> 1,8 | Schaltet Motor 1 ein, rechtsdrehend, mit maximaler Geschwindigkeit.          |
| <b>motor</b> a,b | Je nach Inhalt der Variablen a und b wird ein Motor ein- oder ausgeschaltet. |

### **out <Ausdruck 1>,<Ausdruck 2>**

Steuert einen Ausgang des Robo Interface an. <Ausdruck 1> gibt die Nummer des Ausgangs an (1..32), <Ausdruck 2> die Geschwindigkeit. <Ausdruck 2> kann Werte zwischen 0 und 8 annehmen. Bei 0 wird der Ausgang abgeschaltet.

### **print <Ausdruck>**

Zwar hat das Robo Interface keinen Bildschirm, aber eine serielle Schnittstelle. Mit diesem Befehl können Variablenwerte oder Konstanten (z.B. zur Fehlersuche) über die serielle Schnittstelle ausgegeben werden. Die serielle Schnittstelle ist mit 38400 Baud, No Parity, 1 Stopbit vordefiniert.

Beispiele:

|                      |  |
|----------------------|--|
| <b>print</b> a       | Gibt den Inhalt der Variablen a aus.                       |
| <b>print</b> a+5     | Addiert 5 zur Variablen a hinzu und gibt das Ergebnis aus. |
| <b>print</b> 10      | Gibt die Konstante 10 aus.                                 |
| <b>print</b> "Hallo" | Gibt den Text „Hallo“ aus.                                 |

Nach jedem Print-Befehl wird ein Zeilenvorschub gesendet, so dass die nächste Ausgabe in eine neue Zeile geschrieben wird. Mit dem Semikolon am Ende des auszugebenden Textes kann diese Ausgabe unterdrückt werden.

Beispiel:

|  |   |
|--|---|
| <b>print</b> "Hal";<br><b>print</b> "lo" | Gibt den Text „Hallo“ aus (kein Zeilenvorschub zwischen den Print-Befehlen) |
|--|---|

### **put <Ausdruck>**

Gibt ein einzelnes Zeichen auf der seriellen Schnittstelle aus. <Ausdruck> muss ein Zahlenwert sein.

### **repeat <Anweisungsblock> until <Ausdruck>**

Führt den Anweisungsblock so lange aus, bis der Ausdruck wahr (ungleich 0) ist.

Beispiel:

|   |                          |
|---|--------------------------|
| a = 1   | Stoppt die Motoren 1 - 4 |
| <b>repeat</b><br><b>motor</b> a,0<br>a = a + 1<br><b>until</b> a >= 4 |                          |

Der Anweisungsblock wird mindestens ein Mal ausgeführt.

### **Select <Ausdruck>**

```

case <Konstante 1>
  <Anweisungsblock 1>
case <Konstante 2>
  <Anweisungsblock 2>
.
.
end

```

Mit der „select – case“ - Anweisung können Programmverzweigungen elegant gelöst werden. Beispiel: Die Abfrage der IR-Schnittstelle. Hierbei wird der Ausdruck mit den Konstanten verglichen und bei Gleichheit der entsprechende Anweisungsblock ausgeführt. Gleich danach springt die Programmausführung komplett an das Ende der Select-Anweisung, es werden also keine weiteren Vergleiche durchgeführt.

```

select ir                Abfrage des IR - Eingangs
case #m1l1              ist die Taste „Motor 1 Links“ gedrückt ?
  motor 1,-8            Motor 1 linksherum einschalten
  while ir <> 0           warten, bis die Taste wieder losgelassen wird
  end
  motor 1,0             Motor 1 abschalten

case #m1r1              das Gleiche für rechtsherum
  motor 1,8
  while ir <> 0
  end
  motor 1,0

case #m2l1              weitere Abfragen können folgen
.
.
case #m2r1
.
.
end

```

**<Ergebnis> = sin(<Ausdruck>)**

Berechnet den Sinus von <Ausdruck>.  
Das Ergebnis ist vom Typ double.

**start <taskname>**

Startet einen Task. Wurde der Task vorher durch „stop“ unterbrochen, wird er an der gleichen Stelle ausgeführt, an der er unterbrochen wurde. Ein Task, dessen Ausführung beendet ist (keine Schleife), muss erst durch „kill“ neu initialisiert werden.

**stop <taskname>**

Stoppt einen Task. Wird der task mit start <taskname> gestartet, macht er dort weiter, wo er aufgehört hat. stop <taskname> ist so etwas wie eine "pause" - Taste.

**<Ergebnis> = tan(<Ausdruck>)**

Berechnet den Tangens von <Ausdruck>.  
Das Ergebnis ist vom Typ double.

**task <taskname>**

Definiert einen Task. Man kann sich einen Task vorstellen, wie ein Unterprogramm, das zwischendurch immer wieder aufgerufen wird. Die Definition muss immer durch ein "end" beendet werden.

Siehe auch : start, stop, kill

Wird der Programmteil innerhalb des Tasks nicht durch eine Schleife wiederholt (z.B. durch ein goto), wird der Task nur ein einziges mal ausgeführt, kann aber durch „kill“ und anschließendes „start“ neu gestartet werden.

Innerhalb eines Tasks können lokale Variablen definiert werden. Wichtig ist: Die Variablendefinition muss direkt hinter der Taskdefinition stehen.

Beispiele:

Task ohne lokale Variablen, nur einmal ausgeführt:

```
task reset
  motor 1,0
end
```

Sobald bei der Ausführung das „end“ erreicht wird, bleibt der Task an dieser Stelle stehen.

Task ohne lokale Variablen, dauernd ausgeführt:

```
task blinken
  while 1
    output 1,8
    waitms 200
    output 1,0
    waitms 200
  end
end
```

Task mit lokalen Variablen:

```
task test
  var a long
  var b long
  a = 1
  b = 0
  while 1
    motor a,b
    a = a + 1
    if a > 4
      a = 1
    end
  end
end
```

## **var <variable> <typ>**

Definiert (je nach Position innerhalb des Programms) eine globale oder lokale Variable.

Beispiele:

```
var variable1 long
var x int
var y double
```

Variablennamen müssen immer mit einem Buchstaben beginnen und dürfen anschließend nur Buchstaben, Zahlen oder den Unterstrich \_ beinhalten.

Direkt am Programmbeginn (also vor allen Funktionen und Tasks) definierte Variablen sind global für das gesamte Programm, d.h. sie sind überall im Programm gültig.

## **waitms <Ausdruck>**

Wartet die angegebene Zeit in ms. Währenddessen werden andere Tasks weiter ausgeführt.

Beispiele:

```
waitms 1000
```

```
waitms a
```

```
while <Ausdruck>  
<Anweisungsblock>  
end
```

Führt den Anweisungsblock so lange aus, wie der Vergleich wahr ist. Ist der Vergleich beim ersten Durchlauf bereit unwahr, wird der Anweisungsblock überhaupt nicht ausgeführt (im Gegensatz zu repeat – until).

Endlosschleifen (z.B. für tasks) können mit der while-Schleife sehr einfach realisiert werden:

```
while 1  
  <Anweisungsblock>  
end
```

Die 1 bedeutet in diesem Fall, dass der Vergleich immer wahr ist, also wird diese Schleife niemals verlassen.

Momentan ist der zur Unterbrechung von while-Schleifen übliche break-Befehl nicht implementiert, aber über eine zusätzliche Variable kann diese Funktionalität nachgebildet werden:

```
breakvar = 1  
while breakvar  
  <Anweisungen>  
  if <Abbruchbedingung>  
    breakvar = 0  
  end  
end
```

## 5. Beispielprogramme

Dieses Programm schaltet Motor 1 und 2 links oder rechts abhängig vom Zustand der Eingänge 1 und 2. Wird Eingang 8 betätigt, wird das Programm beendet.

-----

```
main:  
if in(1)=1  
  motor 1,8  
else  
  motor 1,-8  
end  
if in(2)=1  
  motor 1,8  
else  
  motor 1,-8  
end  
if in(8)=0  
  goto main  
end
```

-----

Das gleiche Programm, nur mit Multitasking mit einem Task für jeden Motor.

-----

```
task motor1
```



```

while 1
  if in(1)=1
    motor 1,8
  else
    motor 1,-8
  end
end
end

```

```

task motor2
while 1
  if in(2)=1
    motor 2,8
  else
    motor 2,-8
  end
end
end

```

```

main:
start motor1
start motor2
while in(8) = 0
end

```

-----

Für dieses Programm sollte man 8 Lampen an die Ausgänge anschließen. Die Lampen blinken unabhängig voneinander durch die unterschiedlichen Wartezeiten. Dies ist der Funktionstest für die Multitasking-Funktion.

-----

```

task motor1
while 1
  motor 1,8
  waitms 500
  motor 1,-8
  waitms 500
end
end

```

```

task motor2
while 1
  motor 2,8
  waitms 525
  motor 2,-8
  waitms 525
end
end

```

```

task motor3
while 1
  motor 3,8
  waitms 550
  motor 3,-8
  waitms 550
end
end

```

```

task motor4
while 1
  motor 4,8
  waitms 575

```

```

    motor 4,-8
    waitms 575
end
end

```

```

main:
start motor1
start motor2
start motor3
start motor4
while 1
end

```

Weitere Beispiele befinden sich im Dateiordner „Testprogramme“

## ANHANG

### Vordefinierte Konstanten

Konstanten beginnen immer mit #

|                |      |  |
|----------------|------|--|
| <b>#ax</b>     | (0)  | Analog-Eingang AX am Robo-Interface            |
| <b>#ay</b>     | (1)  | Analog-Eingang AY am Robo-Interface            |
| <b>#az</b>     | (2)  | Analog-Eingang AZ am Robo-Interface            |
| <b>#a1</b>     | (3)  | Analog-Eingang A1 am Robo-Interface            |
| <b>#a2</b>     | (4)  | Analog-Eingang A2 am Robo-Interface            |
| <b>#av</b>     | (5)  | Spannungsversorgung des Robo-Interface         |
| <b>#d1</b>     | (6)  | Distanzsensor-Eingang 1                        |
| <b>#d2</b>     | (7)  | Distanzsensor-Eingang 2                        |
| <b>#axs1</b>   | (8)  | Analog-Eingang AX am ersten Extension-Modul    |
| <b>#axs2</b>   | (9)  | Analog-Eingang AX am zweiten Extension-Modul   |
| <b>#axs3</b>   | (10) | Analog-Eingang AX am dritten Extension-Modul   |
| <b>#avs1</b>   | (11) | Spannungsversorgung am ersten Extension-Modul  |
| <b>#avs2</b>   | (12) | Spannungsversorgung am zweiten Extension-Modul |
| <b>#avs3</b>   | (13) | Spannungsversorgung am dritten Extension-Modul |
|                |      |  |
| <b>#m3r1</b>   | (1)  | IR-Fernbedienung Taste Motor 3 rechts          |
| <b>#m3l1</b>   | (2)  | IR-Fernbedienung Taste Motor 3 links           |
| <b>#m1spd1</b> | (3)  | IR-Fernbedienung Taste Geschwindigkeit Motor 1 |
| <b>#m2spd1</b> | (4)  | IR-Fernbedienung Taste Geschwindigkeit Motor 2 |
| <b>#m3spd1</b> | (5)  | IR-Fernbedienung Taste Geschwindigkeit Motor 3 |
| <b>#code2</b>  | (6)  | IR-Fernbedienung Taste Code2                   |
| <b>#m1r1</b>   | (7)  | IR-Fernbedienung Taste Motor 1 rechts          |
| <b>#m1l1</b>   | (8)  | IR-Fernbedienung Taste Motor 1 links           |
| <b>#m2r1</b>   | (9)  | IR-Fernbedienung Taste Motor 2 rechts          |
| <b>#m2l1</b>   | (10) | IR-Fernbedienung Taste Motor 2 links           |
| <b>#code1</b>  | (11) | IR-Fernbedienung Taste Code1                   |
|                |      |  |
| <b>#up</b>     | (1)  | Für Edge-Funktion                              |
| <b>#any</b>    | (0)  |  |
| <b>#down</b>   | (-1) |  |