Programmieren mit der BASIC-Stamp 2

Teil 7: Subsumtive Programmierung

Von Dennis Clark

1

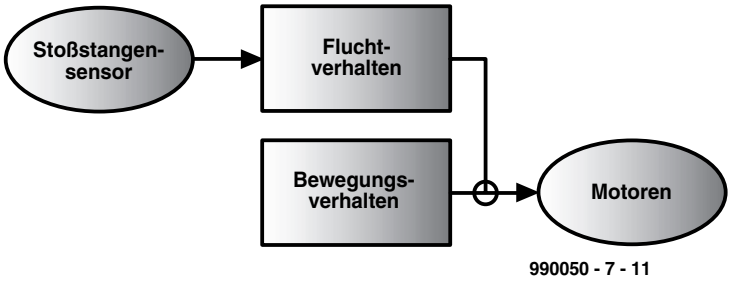


Bild 1. Darstellung des Bewegungs- und Fluchtverhaltens in einem einfachen subsumtiven Netzwerkdiagramm.

Von einer Subsumtion ist in diesem Zusammenhang dann die Rede, wenn ein Task durch einen anderen mit höherer Priorität ersetzt wird. In der Roboterprogrammierung ist das der Fall, wenn ein Verhalten durch ein anderes aufgrund einer Priorität verdrängt wird, die wir zuvor definiert haben. Diese Verhaltensarchitektur wurde zuerst von Dr. Rodney Brooks in seinem Artikel "A robust layered control system for a mobile robot" beschrieben, erschienen im *IEEE Journal of Robotics and Automation*, RA-2, April, 14-23, 1986. Die ideale *Brooks´sche subsumtive Architektur* verlangt nicht, dass ein Verhalten irgendetwas über ein anderes Verhalten weiß. Ein derart programmierter Roboter reagiert reflexartig mit einfachen Verhaltensweisen auf seine Umgebung. Das bedeutet auch, dass ein neues Verhalten hinzugefügt werden kann, ohne irgendetwas an einem anderen Verhalten zu ändern. Bei einer subsumtiven Programmierung unseres kleinen BoE-Bot haben wir zum Beispiel ein Verhalten, das einfach zufällig eine Fahrtrichtung und eine bestimmte Fahrtzeit in dieser Richtung auswählt. Nach Ablauf dieser Zeit legt diese Verhaltensroutine wieder zufallsbestimmt eine neue Richtung und Fahrtzeit fest. Wenn wir aber nicht wollen, dass BoE-Bot sich an einer Wand oder einem Tischbein festfährt, fügen wir ein Verhalten hinzu, das ein Anstoßen (z.B. mit einem Stoßstangensensor) wahrnimmt und unseren Roboter zurückstoßen und zur Seite drehen lässt. Wir geben diesem *"Fluchtverhalten"* natürlich Priorität gegenüber dem eingangs beschriebenen *Bewegungsverhalten*, so dass das Fluchtverhalten das Bewegungsverhalten subsumiert und die Kontrolle über die Radmotoren übernimmt, um den BoE-Bot ausweichen zu lassen. Weil das Fluchtverhalten eine höhere Priorität hat als das Bewegungsverhalten, können wir uns darauf verlassen, dass BoE-Bot in der Lage ist, sich von Hindernissen zu entfernen, die ihm im Weg stehen. Wenn das Bewegungsverhalten die letzte Laufrichtung mit einer langen Laufzeit versehen hatte, wird das Bewegungsverhalten nach dem Ende des Fluchtverhaltens wieder die Kontrolle übernehmen und BoE-Bot unbeirrt in die alte Richtung weiterfahren lassen, als ob nichts geschehen wäre.

Weil jedes Verhalten unabhängig von den anderen ist und viele einfach auf die Umgebung des Roboters reagieren, können wir eine Reihe von Verhalten installieren, deren Interaktionen *nicht* vorprogrammiert sind, so dass wir nicht vorhersehbare Verhaltenskombinationen feststellen werden. Das nennt man *emergent behaviour,* also *"auftauchende"* Verhaltensweisen, die wir nicht geplant haben und die sich aus der Wechselwirkung zwischen dem Roboter und seiner Umwelt ergeben. Wenn wir solches *sich ergebendes Verhalten* zulassen, scheinen unsere Roboter ein Eigenleben zu entwickeln. Es sieht so aus, als ob sie aufhören würden, immer auf die gleiche Weise zu reagieren. Wichtiger für die *reale* Welt ist, dass manches Verhalten programmiert werden kann, um eine bestimmte Aufgabe zu erfüllen, ohne dass man sich um ständige Aufgaben wie der Hindernisvermeidung kümmern muss.

Die beschriebenen einfachen Verhaltensweisen können in einer Grafik dargestellt werden, einem *subsumtiven Netzwerk-Diagramm*. Ein Beispiel für unser einfaches Bewegungs- und Fluchtverhalten ist in **Bild 1** zu sehen. Ellipsen links sind die Eingänge, die Rechtecke in der Mitte die Verhaltensweisen und Ellipsen rechts die Ausgänge. Wenn eine Linie von einem Verhalten mit höherer Priorität auf eine Linie von einem Verhalten mit niedrigerer Priorität auf dem Weg zum Ausgangs-Aktuator (hier Motoren) trifft, sagt man, dass das Verhalten höherer Priorität die Kontrolle vom Verhalten niedrigerer Priorität *subsumiert* (abzieht, übernimmt). Das wird im Diagramm durch den Kreis rund um den Kreuzungspunkt der Linien angedeutet. Es gibt viele Möglichkeiten, ein solches Netzwerkdiagramm zu konzipieren, es kann viele Kreuzungspunkte an vielen Stellen der Richtungslinien geben.

Nächste Schritte

Die beschrieben Diagramme und Denkweisen werden in der Folge regelmäßig und auch in erweiterten Netzwerken verwendet. In jedem Abschnitt werden die Aufgaben in kleine, überschaubare Teile zerlegt, wie benötigte Variable, I/O-Ports und neue Befehle. Gezeigt wird auch der Prozess zur Definition von *State Machines* (*SM*). Das Subsumtions-Netzwerkdiagramm trägt dann dazu bei, dass die Verhaltens-Prioritäten verständlich werden.

Listing 1.

```
'Servo-Routine Konstanten und Variablen
LEFT con 15    'Port 15,linker Motor
RIGHT con 3    'Port 3,rechter Motor
SACT con 5     'Zahl der Schleifendurchläufe
drive var word 'Beide Seite in Variable
ldrive var drive.byte1  'Linke Seite ist hier
rdrive var drive.byte0  'Rechte Seite ist hier

aDur var byte           'Dauer Zähler
```

```
act: 'Servo Controller Subroutine
   if aDur > 0 then aDec
      aDur = SACT         'Status 1 ausführen
      pulsout LEFT,lmotor * 10
      pulsout RIGHT,rmotor * 10
      goto aDone          'Status 1 ausgeführt
aDec:
      aDur = aDur - 1     'Status 2 ausführen
aDone:
return
```

PHASE 1: BOE-BOTS BEWEGUNGSVERHALTEN UND FINITE STATE MACHINE (FSM)

Ein untätiger Roboter ist nicht besonders interessant. Es ist an der Zeit, dem BoE-Bot ein (zufallsgesteuertes) Bewegungsverhalten zu geben.
Dafür brauchen wir zwei Module: Eines, das die Richtung und die Dauer der Fahrt bestimmt, und ein zweites, das für die Steuerung der Räder sorgt. Wir nennen diese Module *wander* bzw. *act*. *Act* steuert einfach die Motoren und ist eigentlich kein Verhalten, sondern ein Ausgang (output), der im Diagramm mit *motor*s bzw. Motoren bezeichnet wird. *Wander* ist hingegen ein Verhalten und wird auch im Subsumtions-Diagramm als solches dargestellt.

Die *State Machine* und Stamp II-Progammkode für *Act*

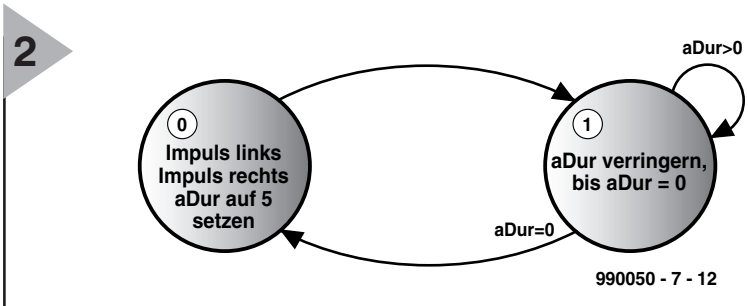
Wie wir bereits wissen, benötigen die Rad-Servos einen Impuls mit einer Dauer zwischen etwa 1 und 2 ms, wobei die Stopp-Position (Stillstand) bei etwa 1,5 ms liegt. Dieser Impuls muss alle 20 bis 30 ms wiederholt werden, damit sich die Räder weiter drehen. Es sind also zwei Operationen zu programmieren, eine für *'output the pulse'*, die andere für *'wait for 20 ms'* und gehe danach zurück zu *state 1*. Es sind zwei Servos, also haben wir drei Operationen: *output pulse* für linken Servo, *output pulse* für rechten Servo und *wait*. Um das Modul einfach zu halten, beschränken wir es auf zwei *states*, einen für die Ausgabe der Impulse und einen für das *delay* bis zur nächsten Impulszeit. Da wir wissen, dass der Impuls alle 20 bis 30 ms zu wiederholen ist und wir nicht die ganze Zeit im *act*-Modul darauf warten wollen, wissen wir auch, dass dieser Programmkodeblock einige Male durchlaufen wird, bevor der ganze Satz von *state transitions* auftritt. Da jeder Befehl der Stamp II zur Ausführung etwa 250 μs braucht, können wir abschätzen, wie lange jedes der aufgerufenen Module läuft, indem wir die Befehle zählen und die Anzahl mit 250 μs multiplizieren. Wir schätzen jetzt und nehmen an, dass wir 5 Durchläufe durch *state* 2 für eine Dauer von 20 ms brauchen. Wenn wir andere Module schreiben, werden wir diese Schätzung revidieren, aber für den Moment ist es ein guter Ausgangspunkt. Wenn wir jetzt die *state machine* grafisch darstellen, können wir verstehen, worum es geht und vielleicht schon Fehler in unserer Logik erkennen.

Bevor wir unsere *state machines* bauen können, ist es hilfreich, alle Aktionen zu definieren, die das Verhalten oder die Ausgangsfunktion ausführen soll. Jede dieser Funktionen kann nach den Aktivitäten einem bestimmten *state* zugeordnet werden. Hier ein Beispiel für die Definition einer Motorsteuerfunktion, die eingangs als Subroutine *act* erwähnt wurde:

```
State 0
 Output left servo pulse value
 Output right servo pulse value
 Set the number of iterations
(aDur) = 5
State 1
 Decrement aDur
 If aDur = 0 then go to state 0
```

Wir haben jetzt eine detaillierte Liste der Aktionen, die in unserer *state machine* für *act* zu passieren haben. Wir können jetzt ein ausreichend detailliertes Zustandsdiagramm (*state diagram*) zeichnen (siehe **Bild 2**).
Dieses Diagramm zeigt uns, wann eine Änderung (*transition*) stattfindet, wir sehen genau, was in den einzelnen *states* passiert. Der Übergang von state 0 (links) nach *state 1* tritt immer auf, es gibt dafür keine Bedingung. Hingegen passiert der Wechsel von *state 1* (rechts) nach *state 0* nur, wenn wenn *aDur* = 0 ist. Zu beachten ist, daß *state 1* eine Schleife aufweist, die austritt und wieder auf sich selbst zurück gerichtet ist. Das zeigt an, daß dieser *state* auf sich selbst *iteriert*, was bedeutet, daß dieser Teil des Moduls mehrmals durchlaufen wird, bevor er abgeschlossen ist. Das Diagramm sollte eingehend betrachtet werden. Es zeigt uns genau, was das *act*-Modul zu tun hat, und in welcher Reihenfolge. Wie würde es zum Beispiel aussehen, wenn der Ausgangspuls für den linken Motor in einem anderen *state* als der für den rechten Motor erfolgen soll? Für unsere Zwecke können wir sagen, daß jeder Kreis (*state*) ein Ort ist, an dem wir in unserem Programm in das betreffende Modul eintreten und entscheiden, was als nächstes zu tun ist. Jetzt sollte man eigentlich bemerken, wie brauchbar dieses Konzept für unsere Zwecke ist!
Der Programmkode, der unser *act*-Modul implementiert, ist in Listing 1 angegeben.
Damit ein Programm leichter zu lesen und zu modifizieren ist, sollen die Konstanten-Definitionen (*con-statements)* reichlich verwendet werden, "magische Zahlen" sollten im Programmkode möglichst nicht auftauchen. Alles rechts von einem ´ ist ein Kommentar und kein Befehl. Warum die *lmotor*- und *rmotor*-Werte mit 10 multipliziert werden, wird im nächsten Abschnitt erklärt.
Beim Schreiben größerer Programme muß man auf Platz für Variable achten, was mit der Sprache PBASIC auf recht intelligente Weise möglich ist. Der *pulsout*-Befehl gibt

Bild 2. Zustandsdiagramm für die "act"-Prozedur.



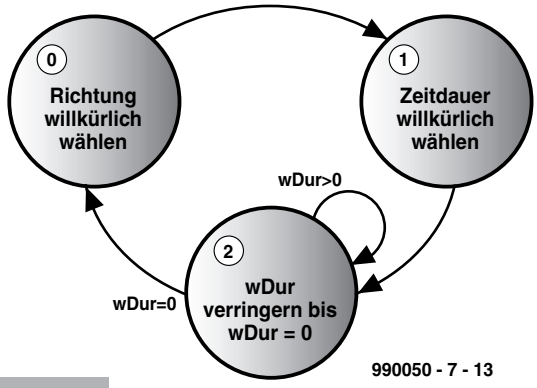


Bild 3. Zustandsdiagramm für die "wander"-Prozedur.

einen Impuls mit der richtigen Pulsbreite für die gewünschte Geschwindigkeit und Richtung an die Servomotoren aus. Die *drive*-Variable ist ein Wort mit zwei Bytes, einem *Byte0* und einem *Byte1*. Beim Schreiben des Kodes für das *wander*-Modul im nächsten Abschnitt werden wir sehen, daß diese Darstellung der Motor-Antriebs-Variablen recht nützlich ist.

PHASE 2: BOE-BOT AUF WANDERSCHAFT

Für das Modul *wander*, das das Bewegungsverhalten definiert, brauchen wir nur zwei Aktionen: Zufallsauswahl der Fahrtrichtung und der Fahrtdauer in dieser Richtung. Dieses Modul ist damit so einfach gestrickt wie das *act*-Modul. Außerdem zeigt sich, wie leicht sich neue Verhaltensweisen hinzufügen lassen und wie sich diese Verhalten "merken", was sie tun. Der erste Schritt der Entwicklung unserer *finite state machine* besteht in der Niederschrift der auszuführenden Schritte und der zu testenden Übergangsfunktionen. Nach Aufstellung dieser Liste kann die *state machine* gezeichnet werden. Hier ist die Aktionsliste für *wander*:

State 0
  Wähle eine Fahrtrichtung mit Hilfe der *random*-Funktion (wDir)
Go to State 1
State 1
  Wähle eine Fahrtdauer mit Hilfe der *random*-Funktion (wDur)
  Füge eine Mindestdauer hinzu, damit die Fahrtdauer nicht zu kurz wird
Go to state 2
State 2
Decrement wDur
If wDur = 0 then go to state 0

Das Diagramm der *state machine* für das Bewegungsverhalten *(wander)* ist in **Bild 3** zu sehen. Nicht alle Pfeile sind dabei mit der Übergangsfunktion bezeichnet. Der Grund: Wenn eine state machine automatisch von einem state zum anderen wechselt und somit keine Entscheidung zu treffen braucht, erübrigt sich auch die Angabe einer Übergangsfunktion. Da beim *wander*-Modul Fahrtrichtung und Fahrtzeit zufallsgesteuert sind, spielt der Befehl *random()* für die Zufallsfunktion eine wichtige Rolle in der Logik des Bewegungsverhaltens. *Random* verwendet eine Variable vom *Wort*-Typ und braucht einen Ausgangswert als "Saat" ("*random seed*"), um einen *return*-Wert zu etablieren. Wir nehmen dabei einfach den letzten *return*-Wert als Ausgangswert für den nächsten. Außerdem verwenden wir noch eine Maske, um nur Werte in einem bestimmten Bereich zu erhalten - einem kleinen Bereich für die Richtungsänderungen und einem wesentlich größeren für die Fahrtdauer. Ebenfalls interessante Möglichkeiten bietet der Befehl *lookup*, der ebenfalls eine Variable von der Größe eines Worts verwendet. Weil wir die *drive*-Variable in eine *byte0*- und eine *byte1*-Variable aufbrechen, können wir die Werte für den rechten und den linken Motor mit einem einzigen *lookup*-Befehl erhalten.
Motorsteuerwerte sind in den Listings mit einem $-Zeichen vor dem Wert aufgeführt, was bedeutet, daß es sich um hexadezimale Werte handelt. Das macht es einfach, die linke und rechte Hälfte (byte1 und byte0) zu erkennen, die wir dem linken bzw. dem rechten Motor zugeordnet haben. Das setzt natürlich voraus, daß man die hexadezimale Schreibweise kennt. Jede Stelle kann dabei mit den Ziffern 0 bis 9 und den Buchstaben von A bis F Dezimalzahlen von 0 bis 15 darstellen. Bei einer mehrstelligen hexadezimalen Zahl ist die Stelle ganz rechts die Einer-Stelle, mit der dann die Zahlen 0 bis 15 dargestellt werden können. Die zweite Stelle von rechts ist die "Sechzehner-Stelle". Bei einer zweistelligen hexadezimalen Zahl multipliziert man daher die zweite Stelle mit 16 und addiert den Wert der ersten Stelle hinzu. Die Zahlen hinter dem $-Zeichen in den Listings haben vier Stellen, weil sie die Werte für die beiden Motoren zusammenfassen. Die beiden Zahlen links stellen damit einen der beiden Werte dar, z.B. hexadezimal $1C, was dezimal 16 + 12 = 28 ergibt.
Jetzt ist es auch Zeit zu erklären, warum *act* die Werte von *ldrive* und *rdrive* mit 10 multipliziert. Ein Byte kann nur Werte von 0 bis 255 darstellen, die Servos benötigen aber Zahlen von 500 bis 1000. Deshalb verwenden wir für ldrive und rdrive nur Werte

Listing 2.

```
'Servo drive commands
fd       con      $6432      'vorwärts
rv       con      $3264      'rückwärts
st       con      $4b4b      'stopp
tr       con      $644b      'nach rechts drehen
tl       con      $4b32      'nach links drehen
rr       con      $6464      'rechts rotieren
rl       con      $3232      'links rotieren
                             'Bewegungswerte
wstate   var byt  e          'FSM Status
wDir     var      word       'Bewegungswert
wDur     var      byte       'Bewegungsdauer


wander:
branch wstate,[wcDir,wcDur]
'This is state 2
wDur = wDur – 1
if wDur > 0 then wDone1
         drive = wDir  'Richtung ermitteln
         wstate = 0    'Status zurücksetzen
wDone1: 'completed
         return
wcDir:                  'Richtung wählen
         random seed    'Zufallsrichtung
         i = seed & %111     'Maskieren für 0-7
         lookup i,[tr,fd,fd,fd,rr,fd,fd,tl],wDir
                                'Richtung wählen
         wstate = 1          'nächster Status
         return
wcDur:                          'Wählen Dauer
         random seed            'Zufallsgenerator
         wDur = (seed & %111111) + 20
                                'Maske für 64 und
20 addieren für mehr Zeit
         wstate = 2          'nächster Status
         return
```

von 50 bis 100 (was kleiner als 255 ist) und multiplizieren diese Werte mit 10, um in den Bereich von 500 bis 1000 zu kommen. Damit nutzen wir den mit einem Byte möglichen Zahlenbereich zwar nur zum Teil, das macht aber nichts, da wir für unseren Antrieb mit modifizierten Servomotoren keine höhere Auflösung brauchen.
Wir haben jetzt schon fast alles, um unseren Roboter ziellos herumwandern zu lassen - aber eben nur fast. Die Funktionsmodule *wander* und *act* sind Subroutinen, das bedeutet, wir brauchen noch etwas, um sie aufzurufen. Dieses etwas besteht aus einer durch *main* und *goto main* definierten Schleife, die alle Verhaltensweisen aufruft. **Listing 3** ist das vollständige Programm, das alle Variablen und Subroutinen ebenso enthält wie den Setup und die Hauptschleife. Der Setup definiert den Status (*state*), mit dem das Bewegungsverhalten (*wander*-Routine) startet. Für den regelmäßigen Aufruf von *wander* und *act* sorgt dann die Hauptschleife, die bei *main* startet.
Wenn wir das Listing mit dem Stamp-2-Programmer programmieren, erhalten wir einen Roboter, der auf einem hindernisfreien Parcour herumwandert. Das Ausweichen, sprich Fluchtverhalten, soll er in der nächsten Folge noch erlernen. Dabei wird die Hauptschleife eine wichtige Rolle spielen.

(990050-7e)

Listing 3.

```
'Allgemeine Werte
I        var      byte                          'Zähler für diverse Schleifen
Tmp      var      word                          'Temporärer Speicher
Seed     var      word                          'Zufallszahl gesetzt
                                                'Dies gilt den Servo-Routinen
LEFT     con      15                            'Port linkes Rad
RIGHT    con      3                             'Port rechtes Rad
SACT     con      5                             'Anzahl act-Durchlauf-Routine
Drive    var      word                          'Rad-Befehl combo
Ldrive   var      drive.byte1                   'Befehl linkes Rad
Rdrive   var      drive.byte0                   'Befehl rechtes Rad
ADur     var      byte                          'Dauer des Impulses
                                                'Servo Fahrbefehle
fd       con      $6432                         'vorwärts
rv       con      $3264                         'rückwärts
st       con      $4b4b                         'stopp
tr       con      $644b                         'nach links drehen
tl       con      $4b32                         'nach rechts drehen
rr       con      $6464                         'rechts rotieren
rl       con      $3232                         'links rotieren
                                                'Bewegungswerte
wstate   var      byte                          'geteiltes Byte
wDir     var      word                          'Bewegungswert
wDur     var      byte                          'Bewegungsdauer
                                                'Einrichten für Bewegung
wstate =0                                       'anfänglicher Bewegungs-Status
main:                                           'dies ist die Haupt-Aktivitäts-Schleife
         gosub wander
         gosub act
goto main
                                                '=======================================
                                                'Verhalten folgen
                                                '=======================================
wander:                                         'Zufälliges Herumwandern
         branch   wstate,[wcDir,wcDur]          'Status 2 folgt unmittelbar
 wDur = wDur - 1
         if wDur > 0 then wDone1
                  drive = wDir                  'Richtung erfassen
                  wstate = 0                    'Status zurücksetzen
wDone1:                                         'vollständig
         return
wcDir: 'choose direction
         random seed                            'Zufallsrichtung
         i = seed & %111                        'Maske off nur für 0-7
         lookup i,[tr,fd,fd,fd,rr,fd,fd,tl],wDir 'Richtung wählen
         wstate = 1                             'nächster Status
         return
wcDur: 'choose duration
         random seed                            'Zufallsrichtung und -dauer
         wDur = (seed & %111111) + 20           'Maske für 64 Möglichkeiten
         wstate = 2                             'nächster Status
         return
act:                                            'Bewegt Servo-Motoren
         if aDur > 0 then aDec                  'wenn schon erledigt, zur aDec-Routine
                  aDur = SACT                   'Durchlaufzahl durch diese Routine
                  pulsout LEFT,ldrive * 10
                  pulsout RIGHT,rdrive * 10
aDec:                                           'Abziehen
         aDur = aDur - 1
aDone:
return
```