Programmieren mit der BASIC-Stamp 2

Teil 6: Finite State Machine

In einem Programm werden die Befehle sequentiell bearbeitet, ein Statement kann erst starten, wenn das vorherige vollständig bearbeitet ist. Wenn die Logik einem “roten Faden” folgt, so wird das Programm linear vom Start bis zum Ende ausgeführt. Folgt der logische Ablauf nicht einer, sondern gleichzeitig mehreren Linien, so spricht man von einer gleichzeitigen oder parallelen Programmausführung.

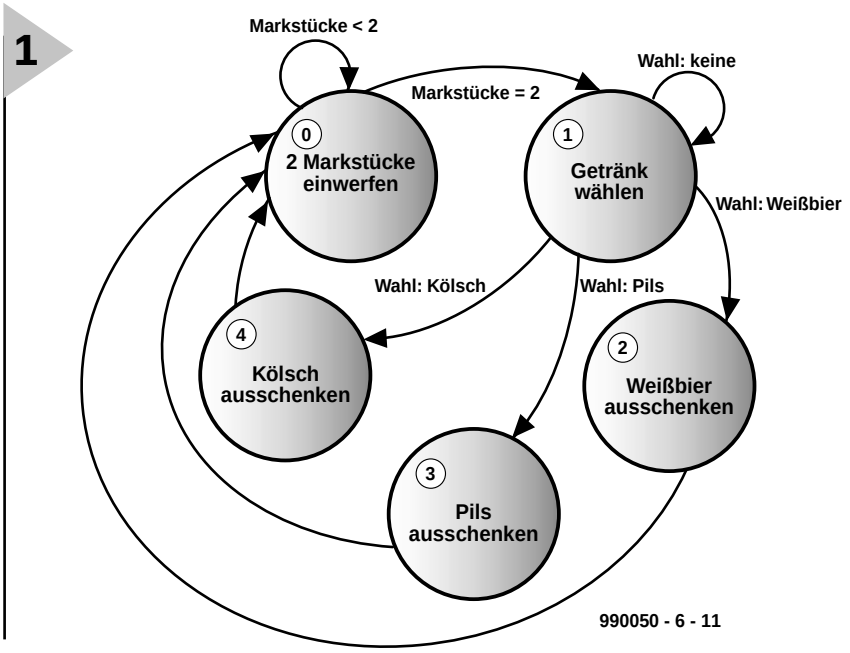

Bild 1. Finite State Machine für einen Getränkeautomat

Parallele Programmabläufe (nicht zu verwechseln mit Multitasking) kann man auch mit der BASIC stamp II realisieren, indem man unabhängige Programmsektionen verwendet, die Module oder (in unserem Fall) Subroutinen genannt werden, die mehrmals aufgerufen werden können, bevor ihre Funktion endgültig ausgeführt ist. Dabei muss stets bekannt sein, wo sich das Programm innerhalb der Routine befindet und an welcher Stelle es nach einem Rücksprung aus der Subroutine fortfährt. Da es nur eine endliche Zahl von verschiedenen Aktionen gibt, die die Subroutine ausführt und die Subroutine an irgendeinem Punkt ihre Aufgabe gelöst hat, nennt man diese Programmabläufe *Finite State Machines* (FSM) oder übersetzt Endliche Ablaufsteuerungen. Aus den verschiedenen Formen von FSMs gebrauchen wir eine Mischform, die speziell für Roboter-Programmierung geeignet ist. Diese “Verhaltens”-FSM produziert keine Aktionen am Ausgang, sondern wechselt lediglich den Zustand aufgrund von Eingangsinformationen und dem gegenwärtigem Zustand. Jede Aktivität ist ein Zustand der *state machine*, einzigartig in seiner Ausführung und unterschiedlich zu allen anderen Zuständen.

Dies ist ohne praktisches Beispiel nur schwer nachzuvollziehen. Das klassische Beispiel einer FSM ist der Getränkeautomat. Um die Erklärung so einfach wie möglich zu halten, haben wir uns dabei auf die Grundfunktionen eines Getränkeautomaten beschränkt: Das Gerät akzeptiert nur Markstücke, wechselt nicht und gibt weder Rest- noch Falschgeld zurück. Alle Erfrischungsgetränke, als da wären Kölsch,

Entwurf von Dennis Clark

Bild 1. Lineare oder parallele Programmierung

Lineare Subroutine zur Servosteuerung

```
act:
    for I = 1 to 10
        pulsout LEFT,750
        Pulsout RIGHT,750
        pause 20
    next
return
```

Subroutine für einen parallelen Programmablauf

```
act:
    if aDur > 0 then aDec
        aDur = 5
        pulsout LEFT,750
        pulsout RIGHT,750
        goto aDone
aDec:
        aDur = aDur - 1

aDone:
return
```

Pils und Weißbier, sind in unbegrenzter Menge vorhanden und kosten jeweils zwei Markstücke.
Wie man sieht, sind alle Möglichkeiten einer Fehlbedienung per definitionem ausgeschlossen, denn schließlich wollen wir FSMs verstehen und nicht Getränkeautomaten konstruieren. Dennoch sind solche Fehler oder Ausnahmen zu berücksichtigen, wenn man eine FSM konkret anwendet. **Bild 1** zeigt eine grafische Darstellung der Getränkeautomat-FSM. Die unterstrichenen Zahlen in jedem Kreis nummerieren die States. Die Zustände sind durch Pfeile miteinander verbunden. Ein Pfeil zeigt einen Übergang (und seine Richtung) von einem in einen anderen Zustand. Wenn einem Pfeil ein Label (Etikett) angeheftet ist, entspringt dieses der Übertragungsfunktion des Zustandes und gibt die Bedingung des Übergangs an. Ein Pfeil ohne Label bezeichnet einfach einen Übergang, nachdem das State seine Aufgabe erfüllt hat. Die Pfeile, die zum gleichen State zurückführen, zeigen, dass die FSM in dem Zustand verharrt und dort irgendetwas unternimmt, um den endgültigen Übergang zu einem anderen State zu ermöglichen. So verharrt der Getränkeautomat im Zustand 0 (*2 Markstücke einwerfen*), da die Bedingung (*Markstücke < 2*) erfüllt ist, bis zwei Markstücke eingeworfen sind. Ist dies geschehen (*Markstücke = 2*), erreicht die *state machine* den Zustand *Getränk wählen*. Hier wartet die *state machine (Wahl:keine)*, bis der Durstige sich für ein Getränk entschieden hat. Dann geht die FSM in einen der drei Zustände 2, 3 oder 4 über und schenkt das gewünschte Getränk aus. Ist diese Aufgabe erfüllt, springt die FSM wieder in den Startzustand 0. In dieser Form wird eine *Finite State Machine* definiert, die auch das Verhalten unseres Roboters bestimmt..
Die Information, an welcher Stelle man sich in der Subroutine befindet, wird als *saving state* bezeichnet und ist essentiell. Jeder Zustand in der FSM wird ausgeführt, wenn die Subroutine aufgerufen ist. Ist der Zustand vollständig erreicht, wird die Subroutine wieder verlassen. Nachfolgende Aufrufe dieser Subroutine führen den nächsten korrekten Zustand aus, der definiert ist. Warum ist das sinnvoll?
Bild 1 zeigt (auch wenn Sie noch nicht alle Befehle kennen) zwei Programmausschnitte, die Servos wie die des BoE-Bots ansteuern können, und zwar mit Impulsen von 1...2 ms Länge, die sich alle 20...30 ms wiederholen. Werden diese beiden Bedingungen nicht eingehalten, arbeitet der Servo nicht korrekt. Erhält der Servo beispielsweise alle 7 ms einen Impuls, so dürfte er nur noch zittern, erfolgt der Steuerimpuls dagegen nur beispielsweise alle 50 ms, so bewegt sich der Servo überhaupt nicht. Auch benötigt der Servo regelmäßige Impulse, mit “Einzelexemplaren” kann er herzlich wenig anfangen.
Der lineare Programmkode erscheint auf den ersten Blick sehr schnell und einfach. Die *Pulsout*-Befehle sorgen für eine Ausgangsimpulsbreite, die die Servos in Rotation versetzt. Die Impulse müssen in Intervallen von 20...30 ms einige Male wiederholt werden, damit sich die Servos sauber drehen. Durch den *Pause*-Befehl wird der Programmablauf für 20 ms unterbrochen, jeder Impuls ist $2\,\mu s \cdot 750 = 1{,}5$ ms lang. So ergibt sich schließlich eine Durchlaufszeit durch die *For-next*-Schleife von 3 ms + 20 ms = 23 ms. Für hier zehn Impulse benötigt die Schleife immerhin 230 ms, fast eine Viertelsekunde, in der die BASIC-stamp zu nichts anderem zu gebrauchen ist!
Werfen wir nun einen Blick auf das Verhalten des zweiten Programmkode-Ausschnitts. Wir haben es dabei mit einer FSM mit zwei Zuständen zu tun. Die Subroutine führt jederzeit eine von zwei Operationen aus. Die erste gibt Impulse an die Servos aus und setzt die *aDur*-Variable, die zweite vermindert auf einfache Weise die Variable *aDur*. In beiden Fällen springt das Programm nach der Operation nach *aDone* und verlässt die Subroutine. Jede der Operationen wird als Zustand im Verhalten der Subroutine *act* definiert.
Gut und schön, aber was unterscheidet am Ende die beiden Subroutinen? Schauen wir uns einmal das “Timing” an. Die BASIC-stamp II führt pro Sekunde etwa 4000 Programmzeilen aus. Jeder Befehl benötigt also ungefähr 250 μs Ausführungszeit. Eine feste Größe sind die beiden Impulslängen für die Servos á 1,5 ms. Im Zustand 1 (Impulsausgabe) zählen zu den insgesamt 3 ms Impulszeit drei Befehle zu je 250 μs, so dass der Zustand 3,75 ms in Anspruch nimmt. Der Zustand 2 (Verminderung von *aDur*) schlägt mit 750 μs zu Buche. Da *aDur* im Zustand 1 auf 5 gesetzt wurde, kommen auf eine Impulsausgabe stets fünf Zustände 2, so dass die gesamte Subroutine 3,75 ms + $(5 \cdot 750\,\mu s)$ = 7,5 ms Prozessorzeit mit Beschlag belegt. Die Subroutine führt die gleiche Aktion aus wie ein Durchlauf der *For-next*-Schleife in der ersten Subroutine mit 23 ms, so dass 15,5 ms Prozessorzeit gewonnen werden.
Warum aber ist das wichtig? Ein Roboter soll schließlich nicht mehr oder weniger blind durch die Gegend fahren, sondern in Kontakt mit seiner Außenwelt bleiben und Aufgaben erfüllen. Ein Feuer suchen und löschen, Müll aufheben oder einen anderen Roboter angreifen ist weitaus wichtiger als nur die Servos in Bewegung zu halten. Gebrauchen wir aber die lineare Subroutine, so konzentriert sich der Roboter für 230 ms nur auf die Ansteuerung der Servos und kann sich weder um Feuer, Müll noch gegnerische Roboter kümmern. Trifft er auf ein Hindernis, so fährt er sich daran fest, bis die Programmschleife beendet ist. Dieses Verhalten muss durch geschicktes Programmieren verhindert werden, denn schließlich ist es nicht gerade sinnvoll, ein Hindernis erst dann zu erkennen, wenn man schon dagegen gefahren ist!
Nehmen wir einen Roboter mit folgenden, von der niedrigsten zur höchsten Priorität geordneten Verhaltenscharakteristika an.

- Gehe nach Norden, bis die Home-Position gefunden ist (wählt eine Fahrtrichtung).
- Vermeide es durch IR-Detektion, an Hindernisse zu stoßen (wenn ein Hindernis detektiert ist, wähle eine

andere Richtung).

- Wenn etwas berührt wird, setze zurück und drehe nach links (wählt noch eine andere Fahrtrichtung)
- Halte an und piepse, wenn die Home-Position wieder erreicht ist (wählt keine Fahrtrichtung, stoppt nur)
- Wähle die Richtung mit der höchsten Priorität und rufe *act* auf, um es auszuführen.

Einige dieser Verhaltensmuster fordern von den Servos eine Aktion: zurücksetzen, nach links drehen und so weiter. Jedes Muster benötigt Informationen aus der Außenwelt, die der eine oder andere Sensor liefert. Jedes der Muster sollte einer *Finite State Machine* entsprechen, die in Form einer Subroutine ausgeführt ist (dazu im nächsten Teil des Kurses mehr). Diese Subroutinen werden aus einem Hauptprogramm aufgerufen.
In die "gesparte" Prozessorzeit von 15,5 ms passen immerhin 62 Befehle, die an anderer Stelle ausgeführt werden können. Zu diesem Zweck springt die Subroutine fünfmal pro Ausführung in das Hauptprogramm zurück, da die Variable *aDur* beim ersten Aufruf von *act* (also unmittelbar vor dem Ausgangsimpuls) auf 5 gesetzt wurde. Die Zahl ist übrigens willkürlich festgelegt und muss später im "Trial-and-error"-Verfahren korrigiert werden.
Während der nächsten fünf Aufrufe der Subroutine werden jeweils nur die drei Befehle (IF...THEN, Verminderung, Return) ausgeführt. Die Ausführungszeit ist minimal. Die für die Servos notwendige Pause von 20...30 ms kann also nahezu völlig genutzt werden, um die Sensoren zu beobachten und die nächste Motoraktion vorzubereiten. Statt also wertvolle Rechenzeit mit einer *Pause* zu verschwenden, nutzen wir die für die Servos notwendige Pause, um die Aufgaben der anderen vier Subroutinen zu erfüllen. Je weniger und schneller zu erfüllende Aufgaben dies sind, desto öfter kann aus der *act*-Routine zurückgesprungen, desto höher der Wert für *aDur* festgelegt werden.
Wenn Sie später einmal mit dem BoE-Bot den Vergleich zwischen linearer und FSM-Programmierung anstellen, werden Sie die Vorteile eines "schnellen" Programmkodes erstaunen. Man hat wirklich den Eindruck, dass alle Aktionen des Roboters gleichzeitig erfolgen anstatt nacheinander.
Bringt man alle Verhaltensmuster in FSMs unter, so kann der Kode, der für die Muster nötig ist, vollständig durchlaufen werden, so dass nicht eine Subroutine warten muss, bis das Ergebnis einer anderen in Form von Motorbewegungen abgearbeitet ist. Dies verbessert die Reaktion des Roboters auf seine Umwelt, im Fall der *act*-Subroutine macht sich dies durch sanftere und "geschicktere" Bewegungen und eine schnellere Reaktion auf Hindernisse und Ziele bemerkbar. Ohne den Einsatz von FSMs zögert der Roboter umso länger bei Entscheidungen, je mehr komplexe Verhaltensmuster programmiert werden. Die FSM-Methode vermeidet ein solch zögerliches Verhalten des Gefährts. Nur wenn eine sehr große Zahl komplexer Muster das Verhalten des Roboters bestimmt, stößt auch diese Programmiermethode an ihre Grenzen. Hier hilft nur noch Assemblerprogrammierung oder ein schnellerer Mikrocontroller weiter. Der FSM-Programmkode ist zweifellos in Entwurf und Praxis komplexer als ein lineares Programm, allein rechtfertigt das Ergebnis die Bemühungen.

(990050-6)rg

Nächsten Monat lernen wir das Subsumptive Programmieren kennen. Damit ist es ein Leichtes, das Roboterverhalten Schritt für Schritt zu planen.

(990050-6)rg