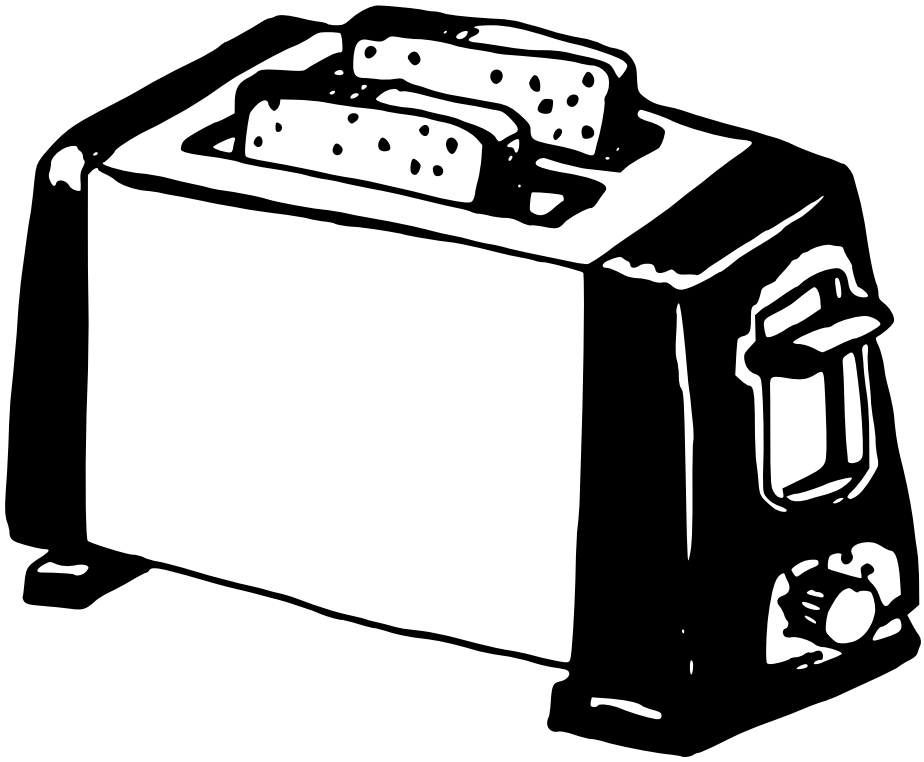
Implementierung einer Finite State Machine

Philipp Kälin

5. März 2012



Urheber

Philipp Kälin
kaelinphilipp@gmail.com

Lizenzen

Dokumentation

Programmcode

Titelbild

Source Dateien

Alle Source Dateien dieses Dokuments und des dazugehörigen Beispielcodes sind bei mikrocontroller.net im Artikel Statemachine verfügbar.

`http://www.mikrocontroller.net/articles/Statemachine#Implementierung_einer_objektorientiert_Finite_State_Machine_in_C.2B.2B`

Dieses Dokument wurde mit LaTeX gesetzt. Die Abbildungen wurden mit DIA und Inkscape erzeugt.

Inhaltsverzeichnis

1 Einleitung

Dieser Artikel hat zum Ziel, eine Möglichkeit zu zeigen wie kleinere bis mittlere endliche Automaten einfach implementiert werden können.
Folgende Sachverhalte werden in diesem Artikel beschrieben:

- Grundlagen was ein endlicher Automat ist und wie er funktioniert
- Notation von Endlichen Automaten in UML[1]
- Praktisches Beispiel, anhand dessen die Funktionsweise eines Toasters erklärt wird. Dazu wird die Notation in UML verwendet.
- Implementation des Beispiels in C++ auf einem AVR-Controller

Die Grundlagen sind in diesem Artikel knapp gehalten und setzen ein grundlegendes Wissen voraus. Um die Grundlagen genauer zu verstehen wird der Artikel *Statemachine* unter
`http://www.mikrocontroller.net/articles/Statemachine` empfohlen.

2 Allgemeines über Endliche Automaten

Ein endlicher Automat beschreibt eine Maschine oder eine Software welche eine definierte Anzahl Zustände hat, in denen sie sich befinden kann. Wozu braucht man nun so etwas?
Mit endlichen Automaten ist es sehr einfach das Verhalten von Systemen zu beschreiben.

3 Funktionsweise eines Moore Automaten

Mit endlichen Automaten kann man auf abstrakter Ebene das Verhalten von Systemen beschreiben, insbesondere auch das von Digitalen Schaltwerken.
Endliche Automaten werden jedoch nicht nur in der Digitaltechnik verwendet, sondern werden auch in der Softwareentwicklung angewendet. Der in der Softwareentwicklung verwendete Automat hat am meisten mit einem Moore Automat gemeinsam, daher wird in diesem Kapitel das grundsätzliche Verhalten eines solchen Moore-Automaten erklärt, zu einem späteren Zeitpunkt folgt dann die Umsetzung in einer Programmiersprache.

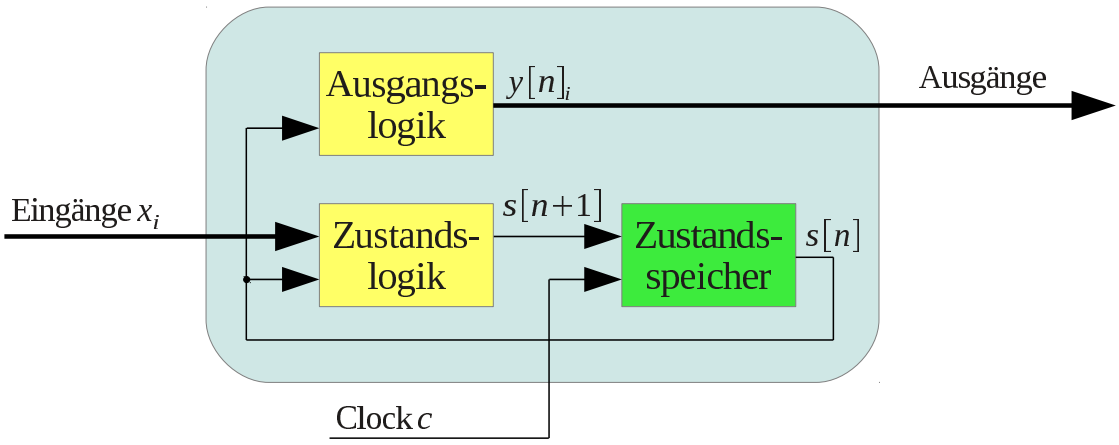


Abbildung 3.1: *Schematische Darstellung eines Moore-Automaten*

Ein Moore Automat besteht im wesentlichen aus drei Teilen. Der wichtigste ist der *Zustandsspeicher*. Dieser speichert den aktuellen Zustand $s[n]$.

[1] Unified Modeling Language

Der nächste Zustand $s[n+1]$ wird von der *Zustandslogik* berechnet. Sobald ein Clock c auftritt wird der Zustand $s[n+1]$ als aktueller Zustand $s[n]$ übernommen.

Die *Ausgangslogik* berechnet gleichzeitig aufgrund des neuen Zustandes $s[n]$ den neuen Ausgänge y_i.

Der neue Zustand $s[n+1]$ wird jeweils aufgrund des aktuellen Zustandes $s[n]$ und den Eingängen x_i berechnet.

Dieses Modell kann fast identisch in der Softwareentwicklung angewendet werden. Die Eingänge werden dabei zu Events, welche von verschiedenen Quellen z.B. Taster, Timer oder auch Schnittstellen kommen.
Der Clock entspricht einem Verarbeitungsaufruf der State Machine. Dabei ist es nicht zwingend, dass dieser Aufruf in einem festen zeitlichen Abstand erfolgt. Der Aufruf kann auch so oft wie möglich erfolgen.
Die Ausgänge können wiederum Hardware, Anzeigen, Kommunikation oder I/O sein, oder auch nur Funktionsaufrufe.

4 UML Notation von Endlichen Automaten

Die UML ist eine graphische Modellierungssprache zur standardisierten Beschreibung von Softwaresystemen. Der hier beschriebene Teil ist nur ein kleiner Ausschnitt aus der ganzen UML, der jedoch für die Beschreibung von Endlichen Automaten ausreicht.

4.1 Auswahl einiger Elemente in UML

Die Abbildung 4.2 zeigt die wichtigsten Elemente die zur Beschreibung eines Endlichen Automaten nötig sind.

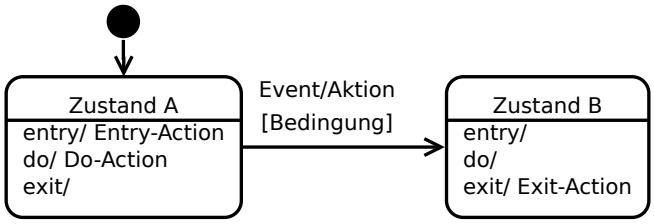


Abbildung 4.2: *UML-Elemente eines Endlichen Automaten*

Zur Verdeutlichung soll ein kleines Beispiel helfen.

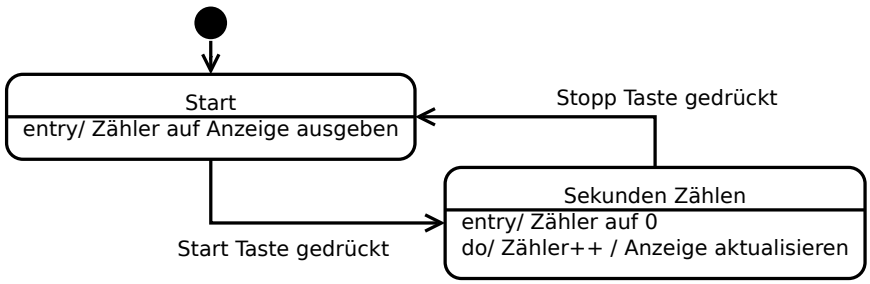


Abbildung 4.3: *Beispiel eines einfachen Zählers*

Nach dem Start befindet sich der Zähler im Zustand *Start*. Dies ist durch den schwarzen Kreis, welcher mit dem Zustand verbunden ist, signalisiert. Bei *Start* wird durch die entry-Action gekennzeichnet, dass der Zähler auf eine nicht näher spezifizierte Anzeige ausgegeben wird.

Ist die Bedingung der Transition zum Zustand *Sekunden zählen* erfüllt, also die *Start Taste* gedrückt, so findet ein Zustandswechsel statt. Im Zustand *Sekunden zählen* wird zu Beginn die Zählervariable auf 0 zurückgesetzt. Die do-Action zeigt an, dass die Zählervariable bei jedem Tick um 1 erhöht wird und die Anzeige mit dem neuen

Wert aktualisiert wird. Aus dem Diagramm ist nicht ersichtlich um welche Periodenzeit es sich bei der tick-Action handelt. Dies muss gesondert angegeben werden.
Im UML-Diagramm spielt es keine Rolle ob die Beschreibungen in "Deutsch" oder Pseudocode hingeschrieben werden. Grundsätzlich sollte ein UML-Diagramm aber unabhängig von der Programmiersprache sein, wieweit man das jedoch durchziehen möchte ist ebenfalls frei.

Ob im Diagramm alle Actions gezeichnet werden, so wie in Abbildung 4.2, oder ob die nicht verwendeten weggelassen werden wie in Abbildung 4.3 ist Geschmackssache. Bei der später beschriebenen Implementation wird es jedoch so sein, dass alle Actions vorhanden sein müssen und gegebenenfalls leer bleiben, da es effizienter ist, eine leere Action aufzurufen als zu prüfen, ob diese überhaupt vorhanden ist.

4.1.1 Zustände

Die Zustände werden in einem Rechteck mit abgerundeten Ecken gezeichnet. Der Name sollte beschreiben, was das System in diesem Zustand macht. Was der Zustand macht wird mit drei verschiedenen Action-Attributen dargestellt.

- Die **Entry Action** beschreibt, was beim Eintreten in einen Zustand gemacht wird. Also dann wenn von einem anderen Zustand in diesen gewechselt wird.

- Die **Do-Action** beschreibt, was passiert wenn ein Event eintritt, der aktuelle Zustand aber nicht gewechselt wird. Ein Beispiel dafür ist ein Tick Event welcher in bestimmten Zeitabständen auftritt. Daher wird zum Teil auch der Name *Tick-Action* verwendet.

- Die **Exit Action** beschreibt, was beim Austreten aus einem Zustand geschieht. Also dann wenn man den aktuellen Zustand verlässt und in einen anderen wechselt.

Der Start eines endlichen Automaten wird mit einem schwarzen Kreis markiert, welcher durch eine Transition zum Startzustand verbunden ist.

4.1.2 Transitionen

Ein endlicher Automat macht natürlich nur wenig Sinn wenn er immer im gleichen Zustand bleibt. Übergänge von einem zum nächsten Zustand nennt man *Transitionen*. Transitionen werden grundsätzlich immer von einem Event ausgelöst. Ein Event kann z.B. sein, dass eine Taste gedrückt wird. Optional an einer Transition ist die *[Bedingung]* welche zusätzlich erfüllt sein muss, damit eine Transition ausgeführt wird. Optional kann bei einer Transition auch eine zusätzliche *\Aktion* stehen welche während des Übergangs ausgeführt wird.
Für einfachere Zustandsautomaten recht es jedoch aus, nur den Event zu definieren.

4.1.3 Events

Events sind Ereignisse welche dem endlichen Automaten mitgeteilt werden, damit dieser gegebenenfalls darauf reagieren kann. Dabei wird grundsätzlich zwischen zwei Arten von Events unterschieden:

- **User-Events** sind Events welche von der Umwelt um das System ausgelöst werden. Beispiele hierfür sind Tasten, Sensoren etc.

- **System Events** sind Events welche vom System selber generiert werden. Das können z.B. zeitlich gebundene Events sein, Timer Ticks, Fehlermeldungen oder Timeouts etc.

Alle Events die auftreten werden an den Endlichen Automaten übergeben. Ob auf ein Event reagiert wird ist durch die Transitionen des aktuellen Zustandes zu den anderen Zuständen festgelegt.

5 Toaster als Beispiel für einen endlichen Automaten

Als anschauliches Beispiel soll ein futuristischer Toaster verwendet werden. Der Toaster besitzt zwei Tasten, eine *Start* und eine *Fertig* Taste, mit denen der Toastvorgang gestartet und beendet werden kann. Um die Sache Komfortabler zu machen hat der Toaster eine automatische Brotschublade ähnlich einem CD-Laufwerk. Um zu erkennen ob die Schublade ganz ein- oder ausgefahren ist, sind zusätzlich zwei Endschalter (*Endschalter drinnen* und *Endschalter leer*) vorhanden. Zur Bereitschaftsanzeige ist eine LED *Ready* vorhanden. Ist die Heizung eingeschaltet leuchtet die LED *Heating*. Sollte ein Fehler auftreten, z.B. wenn die Brotschublade verklemmt ist, so leuchtet die LED *Error*.

5.1 UML-Diagramm des Toasters

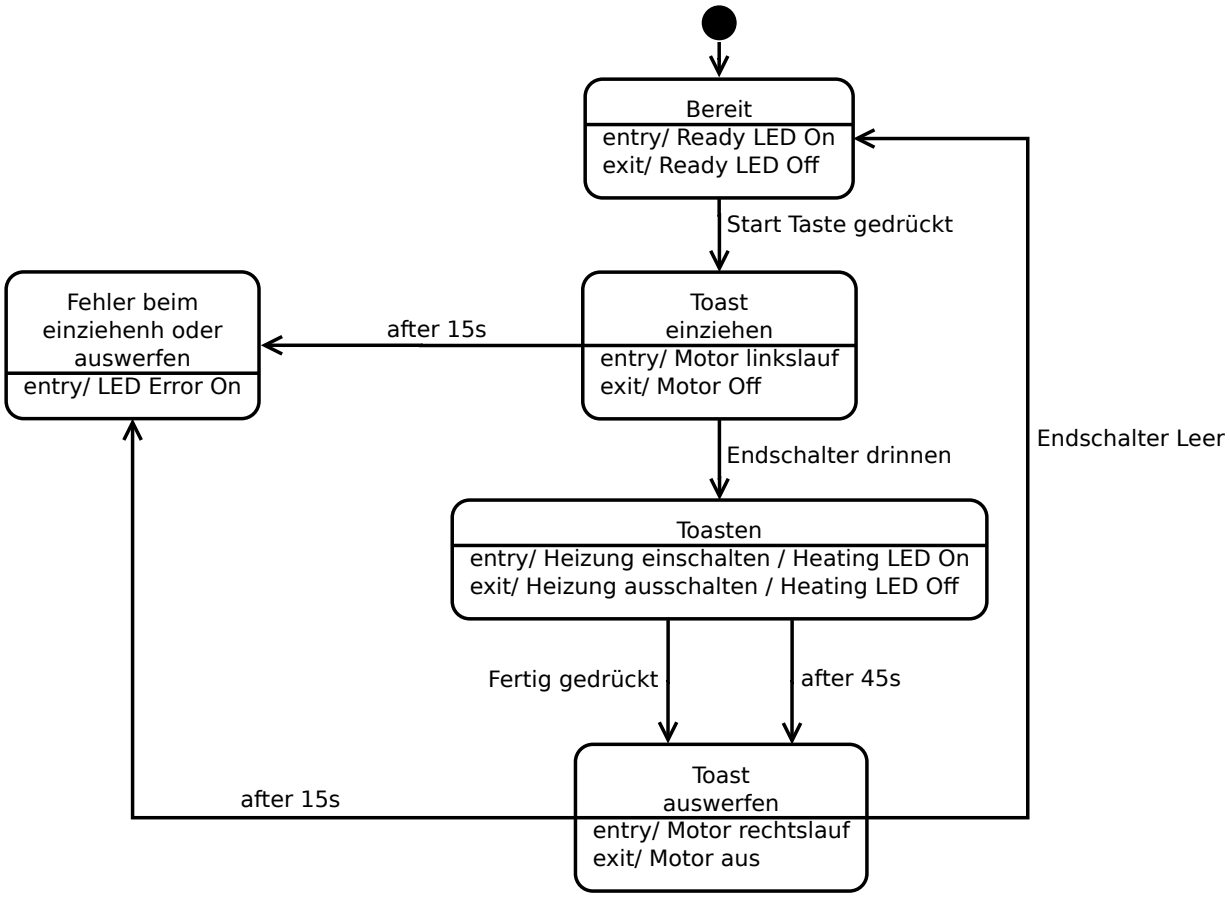


Abbildung 5.4: *Modellierung eines Toasters in UML*

5.2 Funktionsbeschreibung

Der Startzustand ist *Bereit*. Die *Entry* und *Exit* Action bewirken, dass die LED Ready nur leuchtet wenn sich der Toaster im Zustand *Bereit* befindet.

Sobald man im Zustand *Bereit* die *Start Taste* drückt wechselt der Toaster in den Zustand *Toast einziehen*. Um den Toast einzuziehen muss der Motor für die Schublade drehen, im Beispiel Linkslauf. Wenn alles funktioniert, sollte die Schublade innert nützlicher Zeit eingezogen sein.
Wenn die Schublade eingezogen ist wird der *Endschalter drinnen* gedrückt und der Toaster wechselt in den Zustand *Toasten*. Im Zustand Toasten muss natürlich die *Heizung* eingeschaltet werden. Ebenfalls soll die *Heating* LED leuchten.

Um den Toastvorgang zu beenden gibt es zwei Möglichkeiten. Entweder man drückt die *Fertig Taste* oder die 45 Sekunden Maximaldauer sind abgelaufen. In beiden Fällen wird in den Zustand *Toast auswerfen* gewechselt.

Bein ein- und ausfahren der Schublade kann ein Fehler auftreten, wenn die Schublade verklemmt ist. Ist dies der Fall, so kann die Schublade nicht ganz ein- oder ausgefahren werden, demzufolge spricht der entsprechende Endschalter nicht an. Nach dem verstrichenen Timeout, welches mit *after 15s* angegeben ist, wird in den Zustand *Fehler* gewechselt.

Beim Toast auswerfen gibt es wieder die zwei Fälle wie beim einziehen. Entweder der Toast wird korrekt ausgeworfen und der Toaster ist wieder im Zustand *Bereit*, oder es wird in den *Fehler* Zustand gewechselt.

5.3 Implementierung mittels einer Case Struktur

Für die Implementierung gibt es mehrere Möglichkeiten. Die meist verbreitete ist wahrscheinlich die *Case-Struktur*. Ein Beispiel einer solchen Case Struktur ist in Listing 5.1 gezeigt.

```
1  enum {
2    Bereit,
3    ToastEinziehen,
4    ...
5  } ToasterState;
6
7  ToasterState state;              // Variable um den aktuellen Zustand zu speichern
8
9  switch(state) {
10   case Bereit:
11     ...                          // Etwas im Zustand machen
12     state = ToastEinziehen;      // Aktueller Zustand wechseln
13     break;
14
15   case ToastEinziehen:
16     ...
17     state = Toasten;
18     break;
19
20   ...
21 }
```

Listing 5.1: *Beispiel eines Endlichen Automaten mittels Case Struktur*

Eine solche Case Struktur ist gut geeignet um kleinere endliche Automaten abzubilden. Für das Toasterbeispiel gibt es zwei Merkmale, die einem das Leben bei einer solchen Case Struktur schwer machen. Zum einen sind dies die Entry und Exit Actions, wenn in der Case Struktur also in einen Zustand gewechselt wird, so müsste immer abgefragt werden, ob dies der erste Eintritt ist oder nicht. Als zweite Hürde sind die Timeout Bedingungen welche nicht so einfach Realisiert werden können.

5.4 Implementierung mittels einer Tabelle

Das UML-Diagramm kann direkt in eine Tabelle übernommen werden welche die gleichen Informationen enthält wie das Diagramm.

Aktueller Zustand	Event	Timeout [s]	Nächster Zustand
Bereit	Taster Start		Toast einziehen
Toast einziehen	Endschalter drinnen		Toasten
Toast einziehen	Tick	15	Fehler
Toasten	Taster Fertig		Toast auswerfen
Toasten	Tick	45	Toast auswerfen
Toast auswerfen	Endschalter Leer		Bereit
Toast auswerfen	Tick	15	Fehler

Tabelle 1: *Transisionen des Toasters in einer Tabelle dargestellt*

	Entry Action	Exit Action
Bereit	Ready LED On	Ready LED Off
Toast einziehen	Motor Linkslauf	Motor Off
Toast auswerfen	Motor Rechtslauf	Motor Off
Toasten	Heizung einschalten	Heizung ausschalten
	Heating LED On	Heating LED Off
Fehler	LED Error On	

Tabelle 2: *Actions der Zustände in einer Tabelle dargestellt*

Die Darstellung mittels Tabellen hat für die Implementation zwei Vorteile, verglichen mit dem UML-Diaramm. Erstens ist nun eine klare Trennung zwischen Transitionen und Actions sichtbar, und zweitens kann eine solche Tabelle ohne grossen Aufwand in Code umgesetzt werden.
Die Variante mit der Case Struktur wie oben gezeigt hat allgemein das Problem, dass die "Intelligenz" zur Umschaltung zwischen den Zuständen verteilt ist und nicht an einem Zentralen Ort. Dieses Problem soll mit der Tabellenversion gelöst werden.

6 Implementation des Toaster Beispiels auf einem AVR

Die Implementation auf einem Atmel ATmega128 in C++ ist so ausgelegt, dass diese gut im AVR-Studio mit dem Simulator nachvollzogen werden kann. Als Sprache kommt C++ zum Einsatz, da erstens die Implementation übersichtlicher und besser nachvollziehbar ist als in C, zweitens sind heute C++ Compiler so gut dass diese durchaus auf auf Embedded Systemen zum Einsatz kommen. Als Compiler für das Beispiel kommt `avr-g++` zum Zuge.

6.1 Definition der Zustände

Für jeden Zustand wird eine eigene Klasse verwendet. Jeder Zustand wird von der abstrakten Klasse *State* abgeleitet.

```
1 class State {
2   public:
3     virtual void entryAction() = 0;
4     virtual void exitAction() = 0;
5     virtual void tickAcktion() = 0;
6 };
```

Listing 6.2: *Abstrakte Klasse einer Zustandes*

Für jeden Zustand müssen nun die Informationen aus der Tabelle **??** implementiert werden. Als Beispiel soll hier der Zustand *Bereit* verwendet werden. Dazu wird die Klasse *StateBereit* von der Klasse *State* abgeleitet, und die drei Actions implementiert.

```
class StateBereit : public State {
  public:
    void entryAction();
    void exitAction();
    void tickAcktion();
};

//------------------------------------------------------------------------------
void StateBereit::entryAction() {
  Led::ready(true);
}

//------------------------------------------------------------------------------
void StateBereit::exitAction() {
  Led::ready(false);
}

//------------------------------------------------------------------------------
void StateBereit::tickAcktion() { }
```

Listing 6.3: *Implementation des Zustandes Bereit*

6.2 Definition der Transitionen

Eine Transition verbindet zwei Zustände und muss daher zwei Pointer auf die Zustände haben. Ausserdem ist die Information wichtig, bei welchem Event die Transition ausgelöst wird. Die letzte Angabe wird für das Timeout verwendet und besagt wie lange sich der endliche Automat in einem Zustand befinden darf, bevor die Transition ausgelöst wird.

```
struct Transition {
  State*      currentState;
  Event       event;
  uint16_t    maxTimeBeforeTransition;
  State*      nextState;
};
```

Listing 6.4: *Definition der Transition*

6.3 Abbildung der Tabelle in C++

Nun sind alle Einzelteile bekannt um diese in einer Tabelle abzubilden und somit den identischen Informationsgehalt wie in Abbildung 5.4 zu erhalten. In C++ umgesetzt sieht das folgendermassen aus:

```
Toaster::Transition fsmApp[] = {
  // actualState                    MaxTimeBeforeTransition
  //             Event                              Next State
  {&bereit,      TasterStart,       0,              &einziehen },

  {&einziehen,   EndschalterDrinnen, 0,              &toasten   },
  {&einziehen,   Tick,              Seconds(15),     &fehler    },

  {&toasten,     TasterFertig,      0,              &auswerfen },
  {&toasten,     Tick,              Seconds(45),     &auswerfen },

  {&auswerfen,   EndschalterLeer,   0,              &bereit    },
  {&auswerfen,   Tick,              Seconds(15),     &fehler    },
};
```

Listing 6.5: *Abbildung der Tabelle in C++*

6.4 Verwendung des endlichen Automaten

Die Verwendung des endlichen Automaten kann in zwei Jobs unterteilt werden. Zum Einen müssen natürlich die auftretenden Events gesendet werden. Da es vorkommen kann, dass mehrere Events auftreten bevor diese abgearbeitet werden, werden die Events in eine Queue eingereiht, deren Länge mit `EventQueueSize` festgelegt werden kann.

```
1 Toaster::sendEvent(Toaster::EndschalterDrinnen);
```

Listing 6.6: *Event an den endlichen Automaten senden*

Falls Timeouts verwendet werden, so können *Tick* Events ganz einfach innerhalb eines Timer-Interrupt gesendet werden.

```
1 ISR( TIMER0_OVF_vect ) {
2   Toaster::sendEvent(Toaster::Tick);
3 }
```

Listing 6.7: *Tick Event an den endlichen Automaten senden*

Die Verarbeitung der Events erfolgt am besten im Hauptprogramm in einer Endlosschleife und wird so oft wie möglich aufgerufen.

```
1 Toaster::process();
```

Listing 6.8: *Process Aufruf um einen Event zu verarbeiten*

6.5 Verarbeitung der Events

Wie oben erwähnt werden die Events in einer Queue gesammelt und mit einem `process()` verarbeitet. Die Funktion process durchsucht dabei die Tabelle nach einer Zeile, in welcher der aktuelle Zustand und der Event übereinstimmen. Wird eine solche Zeile gefunden, wird ein Wechsel des Zustandes vorgenommen.

Der Grosse Vorteil dieser Tabellen Methode gegenüber des bereits genannten Case Konstrukts ist, dass die gesamte Intelligenz des endlichen Automaten in einer Funktion steckt.

```
1  bool Toaster::process() {
2    if (evQueueIsEmpty()) {
3      return false;
4    }
5    Event e = evQueue[evQueueHead];
6    evQueueHead = (evQueueHead + 1) & (evQueueSize - 1);
7
8    for ( uint8_t i=0; i<sizeof(fsmApp)/sizeof(Transition); i++ ) {
9      if (    ( (currentState == fsmApp[i].currentState) && (e == fsmApp[i].event) )
10         || (fsmApp[i].event == 0)    )
11     {
12       if (e == Tick) {
13         // System Tick
14         fsmApp[i].currentState->tickAcktion();
15
16         timeInState++;
17         if (    (fsmApp[i].maxTimeBeforeTransition != 0)
18             && (timeInState < fsmApp[i].maxTimeBeforeTransition) )
19         {
20           // Normal Tick
21           fsmApp[i].currentState->tickAcktion();
22         } else {
23           // Force Transition because of an Timeout
24           fsmApp[i].currentState->exitAction();
25           currentState = fsmApp[i].nextState;
```

```
26            currentState->entryAction();
27            timeInState = 0;
28          }
29        } else {
30          timeInState = 0;
31          fsmApp[i].currentState->exitAction();
32          currentState = fsmApp[i].nextState;
33          currentState->entryAction();
34        }
35        break;
36      }
37    }
38    return true;
39  }
```

Listing 6.9: *Gesamte Intelligenz des endlichen Automaten*

7 Schlusswort

Das Beispiel ist so ausgeführt, dass es im Simulator des AVR-Studio 4 nachvollzogen werden kann. Als C++ Compiler wird `avr-g++` verwendet, welcher unter anderem in `WinAVR` vorhanden ist.

Die Idee der Tabellenvariante kann natürlich auch in C umgesetzt werden. Dazu wird in der Tabelle jeweils der Pointer auf das Objekt durch drei Funktionspointer ersetzt, welche jeweils direkt auf die Action-Funktionen zeigen.

Die gezeigte Möglichkeit bzw. das Beispiel sollte als Denkanstoss verstanden werden und nicht als Referenzimplementation. Es wurden bewusst bestimmte Feinheiten von endlichen Automaten verzichtet, um das Beispiel auf verständlichem Niveau zu halten.

7.1 Links und Source

Ausführlichere Informationen und den Source dieses Artikels und der Beispielcode sind im Artikel *Statemachine* im mikrocontoller.net Wiki zu finden. http://www.mikrocontroller.net/articles/Statemachine#Implementierung_einer_objektorientiert_Finite_State_Machine_in_C.2B.2B