# The StateMachine Library

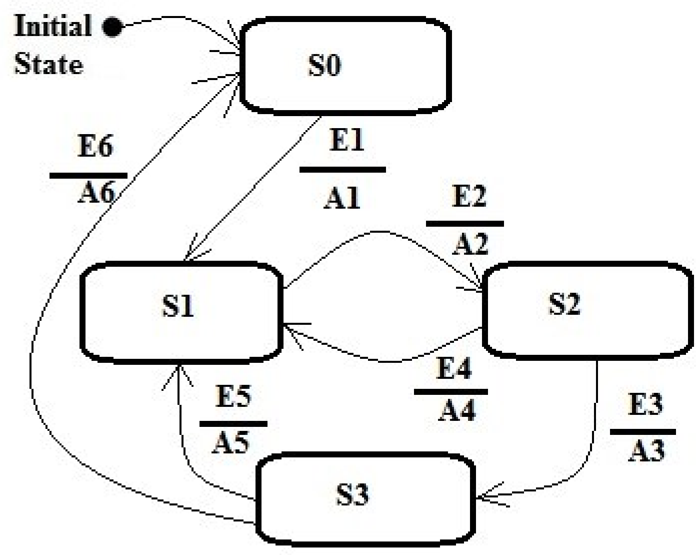
16-03-2014

## Content

The library ‘StateMachine’ provides a simple mechanism to build a state machine in the program. It does not support “SuperStates”, ”, “entry”, “exit” or “Do” actions, only transition related actions (it is a “Mealy” machine). It also has no event queue.

# 1 The StateMachine

What is a state machine? It implements a “state transition diagram” like the following:



In above state diagram

- S0..S3 represents the different **states** the system can be in (S0 being the initial state),
- E1..E6 represent the **events** that cause state transitions
- A1..A6 represent the **actions** to take place when a state transition occurs. This action is optional.

A State Transition diagram is a good way to describe the behaviour of a system, and is easy to implement.

## 2 Using the Library

For this library the user has to provide a number of items in the program:

- Constants for **Events** and **States**, see Events and States Constants.
- All "**Action**" procedures that are mentioned in the "StateMachine_TransitionTable" below, see The "Action" procedures.
- A constant named "**StateMachine_NrTransitions**", holding the number of state transitions in the "StateMachine_TransitionTable" below, see The "StateMachine_NrTransitions" constant.
- A constant named "**StateMachine_TransitionTable**", representing the State transition diagram to be implemented, see The "StateMachine_TransitionTable".
- A function named "**StateMachine_GetEvent**" of type "byte" which returns the event that occurred in the system (zero means "no event"), see The "StateMachine_GetEvent".
- Running the state machine, see Running the State machine.

### 2.1 Events and States Constants

Example:

```
// ------------ Event Constants ------------------------------------------------
const E1 = '1';
      E2 = '2';
      E3 = '3';
      E4 = '4';
      E5 = '5';
      E6 = '6';

// ------------ State Constants ------------------------------------------------
const S0 = 0; // initial state (always zero)
      S1 = 1;
      S2 = 2;
      S3 = 3;
```

The constant names can be chosen freely.

In above example the event values are characters (e.g. '1'), but they can be chosen freely, as long as they are mutually exclusive.

The constant values (0..254 for states, 1..255 for events) can be chosen freely except in 3 cases:

- Events can not have the value zero ('zero' is 'no event'),
- The **initial state** (here S0) has always the value zero.
- A value of 255 means "**any state**". This value can only be used as "fromstate" in The "StateMachine_TransitionTable". AnyState can be used together with "general" events as .e.g safety or alarm events, where the transition towards a certain state always has to occur.

## 2.2 The Action procedures

They have the following signature:

```
procedure Action1(From, ToWards, Event: byte);
```

As one can see the Action routines receive all the information they need to be able to do the correct thing: the current state (From), the future state (ToWards) and the event that causes the state transition (Event).

An example:

```
procedure A1(From, ToWards, Event: byte);
begin
  Uart1_write_text('From: ');
  Uart_write_UnsInt(From);
  Uart1_write_text(', To: ');
  Uart_write_UnsInt(ToWards);
  Uart1_write_text(', Event: ');
  Uart_write_UnsInt(Event);
  Uart_write_Line(', Action1');
end;
```

## 2.3 The StateMachine_NrTransitions constant

Equal to the number of State transitions in the State Transition Table below. The name can NOT be chosen freely.

Example:

```
const StateMachine_NrTransitions: byte = 6; // number of transitions in the table below
```

## 2.4 The StateMachine_TransitionTable

This is an array with elements of the following format:

```
fromstate, tostate, event, action
```

```
Example (according the above state diagram):

const StateMachine_TransitionTable: array[StateMachine_NrTransitions] of TStateTransition
= // obligatory table
      // format: fromstate, tostate, event, action
      (
      (S0, S1, E1, @A1), // leave initial state (zero)
      (S1, S2, E2, @A2),
      (S2, S3, E3, @A3),
      (S2, S1, E4, @A4),
      (S3, S1, E5, @A5),
      (S3, S0, E6, @A6)
      );
```

The table is scanned in the order stated in the table.

Once a matching entry is found (*fromstate* and *event* match) the *action* is executed and the transition is made, the rest of the table is not scanned further.

The order of the entries in the table is of no importance (normally the entries in the table are mutually exclusive) except when the "AnyState" value (255) is used as a "fromstate". The latter should be placed below more strong transitions (that is: where the fromstate does not equal "anystate" for a certain event).

e.g.:

```
const StateMachine_TransitionTable: array[StateMachine_NrTransitions] of TStateTransition
= // obligatory table
      // format: fromstate, tostate, event, action
      (
      ...
      (S2, S3, E3, @A3), // first the occurance of E3 will be checked in state S2
      (S2, S1, E4, @A4),
      ...
      (255, Sx, E3, @Ax) // goto Sx from any other state than S2 when E3 occurs
      );
```

If no action is needed during a state transition "nil" should be specified as action. Actions may be used for different transitions.

The name of the table can NOT be chosen freely.

## 2.5 The StateMachine_GetEvent function

This function has the following signature:

*function StateMachine_GetEvent: byte;*

An example:

```
function StateMachine_GetEvent: byte; // oligatory function
begin
  Result := Event; // Fetch the pending Event (if some) that occured in the system
  Event  := 0;     // clear the pending Event
end;
```

As one can see the (pending) event is returned by the function and should subsequently be cleared.
The variable "Event" (name can be freely chosen) has to be given the correct value by the process. A value of 0 means "no event", a value of >0 means "something happened" (e.g. a button was pushed or a temperature rised above a limit). The process can of course use more than one "Event" variable to gather multiple events. They must all be represented by a different number however.

The name of the function can NOT be chosen freely.

## 2.6 Running the State machine

This is quite simple.

2 methods exist:

- Call the "**StateMachine_Run**" routine. No code should appear below this call, it will not be executed ("StateMachine_Run" calls "StateMachine_Init" and after that calls "StateMachine_Step" in an endless loop) , **OR**

- Call the “**StateMachine_Init**” routine once, and after that call “**StateMachine_Step**” in a loop of your own. The latter method will allow also to execute code outside the StateMachine.

Examples:

```
// Method 1
  StateMachine_Run;  // start the state machine
  // no code allowed below this line (it will not be executed). All actions have to be
done in the transition "Action" routines
```

OR

```
// Method 2
  StateMachine_Init;
  while true do
  begin
    Statemachine_Step; // make the statemachine do one step
    // do other stuff here
  end;
```

The “StateMachine_Step” routine does the following:

- It calls its “….GetEvent” function to fetch (if any) a pending event,
- If compares all entries in its “State Transition Table”. If it finds an entry with its “from state” matching the current state and with its “Event”  matching  the newly fetched “event” then
  - It calls the appropriate “Action” routine with as parameters : “From” (the current state), “Towards” (the next state), and the “event” that causes the transition.
  - After that the actual state transition will occur: the current state of the stateMachine is set to the new state
  - It skips the remaining entries in the table (i.e. only the first matching entry is executed)
- If it does not find an entry in the table then no action and no transition takes place.

## 3 The interface part of the “StateMachine” library

```
unit StateMachine;

// interface

type
     TActionProc = procedure(From, Towards, Event: byte);
     TStateTransition =
     record
       Fromstate, ToState, Event: byte;
       Action: ^TActionProc;
     end;
     TStateMachineTable = array[1] of TStateTransition; // dummy size

// published variables
var CurrentState_: byte; // can be used to observe the current state (if necessary)

// published routines
procedure StateMachine_Run;
procedure StateMachine_Init;
procedure StateMachine_Step;
```

```
// external items
function StateMachine_GetEvent: byte; external;
const StateMachine_NrTransitions: byte; external;
const StateMachine_TransitionTable: array[1] of TStateTransition; external; // dummy size


implementation
```

[end of document]