The StateMachine_2 Library

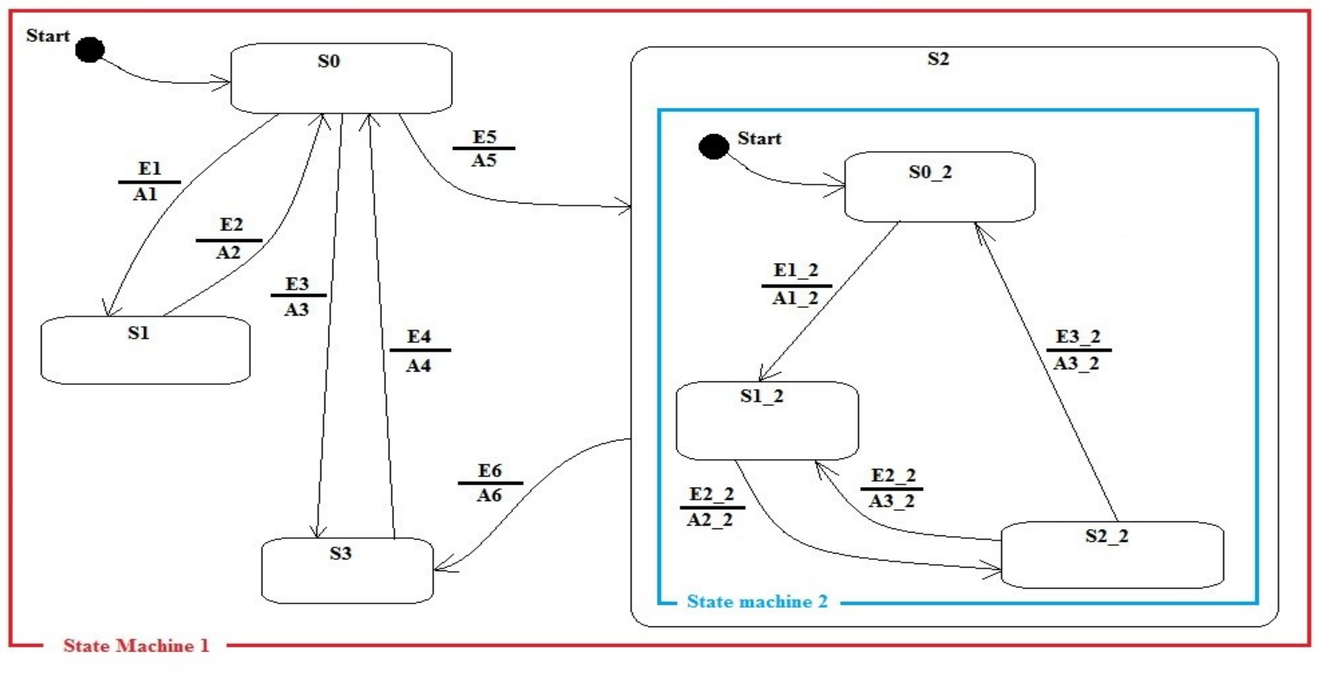
16-03-2014

Content

The library ‘StateMachine_2’ provides a simple mechanism to build one or more state machines in the program. It does support “SuperStates”. It does not provide state “entry”, “exit” or “Do” actions, only transition related actions (it is a “Mealy” machine). It also has no event queue.

1 A State TransitionDiagram and a State Machine

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



In above state transition diagram (**STD**)

- **S**x represent the different **states** the state machine can be in,
- The arrows show the transitions that can take place between states,
- **E**x represent the **events** that cause such state transitions
- **A**x represent the **actions** to take place when a state transition occurs. This action is optional.

In the STD above the state “S2” is a socalled “super state”, which means it has an internal state machine of its own. This means 2 state machines will be needed: the main one (called “State Machine 1” in the diagram), and the one for state S2 (called “State Machine 2” in the diagram). All items in state machine 2 have the “_2” suffix to make things clear.

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:

1. For each State Machine needed in the program: a variable of the type “TstateMachine” should be created, see here.
2. For each state machine: make constants for Events and States, see here.
3. All “**Action**” routines that are mentioned in the State Transition Tables (see pt 2). Make sure that these actions, if needed, start up or stop additional state machines as needed, see here.
4. For each State Machine: a constant table (a State Transition Table) representing the State transition diagram of that state machine. Additionally a constant holding the number of state transitions in the above State Transition Table, see here.
5. For each State Machine: a function of type “byte” which returns the event that occurred in the system (zero means “no event”) meant to be executed by the state machine. Make sure all events for that state machine are returned by this function, see here.
6. For each statemachine: initialise the state machine (routine “StateMachine_Init”) , see here.
7. Start all state machines that need startup (routine “StateMachine_Cold_Start”), see here.
8. In an endless loop: make ALL statemachines (started or not) do a “step” (routine “StateMachine_Step”), see here.
9. Stop and restart a state machine if necessary, see here and here.

2.1 The StateMachine Variables

For each State Machine a variable of type “TstateMachine” is needed.

For example as in the STD:

```
var StateMachine1: TStateMachine; // 2 state machines needed
    StateMachine2: TStateMachine;
```

2.2 Events and States Constants

For each State machine constants have to be defined for its States and for its events. The initial state is always state 0, a “no event” is always value 0.

Example for State machine 1:

```
// ------------ 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;
```

Example for State Machine 2:

```
// ------------ Event Constants ------------------------------------------------
const E1_2 = 'a';
      E2_2 = 'b';
      E3_2 = 'c';

// ------------ State Constants ------------------------------------------------
const S0_2 = 0; // initial state (always zero)
      S1_2 = 1;
      S2_2 = 2;
```

As one can see the "Events" are characters in this example.

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 and S0_2) has always the value zero.
- A value of 255 means "**any state**". This value can only be used as "fromstate" in the State Transition Table. 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.

2.3 The Action routines

In these "actions" all necessary code is executed that should when the state of a State Machine is changing.

They have the following signature:

procedure Action1(Id, 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 Identification of the state machine calling the Action (Id), the current state (From), the future state (ToWards) and the event that causes the state transition (Event).

Examples (refer to the STD above):

```
procedure A1(Id, From, Towards, Event: byte);
begin
  display1(Id, From, Towards, Event, 'A1');
end;

procedure A5(Id, From, Towards, Event: byte);
begin
  display1(Id, From, Towards, Event, 'A5');
  PendingEvent2 := 0;                      // clear pending event for its state machine
  StateMachine_Cold_Start(Statemachine2); // entering substate, start its StateMachine
end;

procedure A6(Id, From, Towards, Event: byte);
begin
  display1(Id, From, Towards, Event, 'A6');
  StateMachine_Stop(Statemachine2);       // leaving substate, stop its StateMachine
end;
```

As can be seen the actions A5 and A6 respectively start and stop State Machine 2. In above examples no real work is done, except “display” the transition…

2.4 The State Transition Table

This is an array with elements of the following format:

fromstate, tostate, event, action

An example for statemachine 1 as in the above STD:

```
const StateMachine1_NrTransitions: byte = 6;
      StateMachine1_TransitionTable: array[StateMachine1_NrTransitions] of
TStateTransition =
      // format: fromstate, tostate, event, action
      (
      (S0, S1, E1, @A1),
      (S1, S0, E2, @A2),
      (S0, S3, E3, @A3),
      (S3, S0, E4, @A4),
      (S0, S2, E5, @A5),
      (S2, S3, E6, @A6)
      );
```

An example for statemachine 2 as in the above STD:

```
const StateMachine2_NrTransitions: byte = 4;
      StateMachine2_TransitionTable: array[StateMachine2_NrTransitions] of
TStateTransition =
      // format: fromstate, tostate, event, action
      (
      (S0_2, S1_2, E1_2, @A1_2),
      (S1_2, S2_2, E2_2, @A2_2),
      (S2_2, S1_2, E2_2, @A3_2),
      (S2_2, S0_2, E3_2, @A4_2)
      );
```

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 StateMachine2_NrTransitions: byte = 5;
      StateMachine2_TransitionTable: array[StateMachine2_NrTransitions] of
TStateTransition =
 // format: fromstate, tostate, event, action
 (
 (S0_2, S1_2, E1_2, @A1_2),
 (S1_2, S2_2, E2_2, @A2_2), // first the occurance of E2_2 will be checked in state S2_2
 (S2_2, S1_2, E2_2, @A3_2),
 (S2_2, S0_2, E3_2, @A4_2),
 ( 255, Sx_2, E2_2, @Ax_2)  // goto Sx_2 from any other state than S2_2 when E2_2 occurs
 );
```

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

2.5 The GetEvent function

Each State Machine should have one.

This function has the following signature:

```
function StateMachine_GetEvent: byte;
```

An example for statemachine 1 as in the above diagram:

```
function StateMachine1_GetEvent: byte;
begin
  Result := PendingEvent1; // Fetch the Event (if some) that occured in the system
  PendingEvent1 := 0;      // clear the pending Event
end;
```

An example for statemachine 2 as in the above diagram:

```
function StateMachine2_GetEvent: byte;
begin
  Result := PendingEvent2; // Fetch the Event (if some) that occured in the system
  PendingEvent2 := 0;      // clear the pending Event
end;
```

As one can see the (pending) event variable is returned by the function and subsequently cleared.
The variable "PendingEventn" (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 of a temperature rised above a limit). The process can of course use more than one "PendingEventn" variable to gather multiple events. They must all be represented by a different number however.

2.6 Initialisation of a State Machine

Each state machine should be initialised . During initialisation the following is assigned to the state machine:

- An Identification, any number between 0 and 255. This number will be the first parameter when an "Action" routine is called.
- The state Machine Variable involved,
- The appropriate State Transition Table
- The appropriate Nr of Transitions in above table
- The appropriate "Get_Event" routine

Example for State Machine 1:

```
StateMachine_Init(1, StateMachine1, @StateMachine1_TransitionTable,
StateMachine1_NrTransitions, @StateMachine1_GetEvent);

Example for State Machine 2:
StateMachine_Init(2, StateMachine2, @StateMachine2_TransitionTable,
StateMachine2_NrTransitions, @StateMachine2_GetEvent);
```

2.7 Starting a State machine

This is quite simple.

2 methods exist:

- Call the "**StateMachine_Cold_Start**" routine. A cold start always resets the current State of the machine to zero before actually switching it on
  OR
- Call the "**StateMachine_Warm_Start**" routine. A warm start routine only swithes the state machine on, it keeps the state it was in.

Example for State Machine 1:

```
PendingEvent1 := 0; // clear pending events for State Machine 1 (optionally)
StateMachine_Cold_Start(StateMachine1); // start state machine 1
```

Example for State Machine 2:

```
PendingEvent2 := 0;    // clear pending event for its state machine (optionally)
StateMachine_Cold_Start(Statemachine2);// entering substate, start its StateMachine
```

2.8 Making a State Machine do its work

Initialisation and Starting a State Machine is not enough to let it do its work. One should call in an endless loop the routine "StateMachine_Step" after Initialisation of the State Machine.

This "step" routine will call the "GetEvent" routine, evaluate the event and the current State against the state transition table, call the appropriate "Action" routine, and set the Current State to a new value.

The "StateMachine_Step" routine can be called always, also when a state machine is stopped (it has be be initialised though).

Example:

```
while true do
  begin

    // gather events occuring in the system (can be done in any routine)
    …

    Statemachine_Step(StateMachine1); // make the statemachine 1 do one step
    Statemachine_Step(StateMachine2); // make the statemachine 2 do one step

    // do other stuff

  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: “Id” (the calling state machine identity), “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.

2.9 Stopping a State Machine

A State Machine can always be stopped (it does not react any longer to its events), and Cold or Warm started again.

Example for State Machine 1:

```
StateMachine_Stop(StateMachine1);
…
StateMachine_Warm_Start(StateMachine1); // proceed in the last state the machine was in
```

3 The interface part of the “StateMachine” library

```
unit StateMachine_2;

// interface

uses StrngUtils;

type
     TGetEventProc = function: byte;

     TActionProc = procedure(Id, From, Towards, Event: byte);

     TStateTransition =
     record
       Fromstate, ToState, Event: byte;
       Action: ^TActionProc;
     end;

     TStateMachineTable = array[1] of TStateTransition; // dummy size

     TStateMachine =
     record
       Ident            : byte;
       TransitionTable : ^const TStateMachineTable;
       NrTransitions    : byte;
       GetEventProc     : ^TGetEventProc;
       Running          : boolean;  // running or stopped
       CurrentState     : byte;     // the current state of the state machine
       CurrentEvent     : byte;     // the current event to react upon
       NextState        : byte;     // the future state of the state machine
     end;

// published routines
procedure StateMachine_Init(ID: byte; var Machine: TStateMachine; Table: ^const
TStateMachineTable; Nr: byte; EventProc: ^TGetEventProc );
procedure StateMachine_Cold_Start(var Machine: TStateMachine);
procedure StateMachine_Warm_Start(var Machine: TStateMachine);
procedure StateMachine_Stop(var Machine: TStateMachine);
procedure StateMachine_Step(var Machine: TStateMachine);


implementation
                                                                   [end of document]
```