Bern University
of Applied Sciences

Exercise - 26

State Machine

Andreas Habegger | Adrian Steiner
BTS4230 | Version 1.0.0 of 17.03.2025

A State-Event driven Embedded System is a design paradigm used in the development of embedded systems, where the behavior of the system is driven by both its current state and external events. In essence, a State-Event driven Embedded System combines the principles of state machines (where the system's behavior is determined by its current state) with event-driven programming (where the system responds to external events). This makes embedded systems easier to understand, maintain and extend by providing a structured and modular approach to system design.

This exercise describes several possible implementations using an MCU. It complements the discussion in the lecture.

Objectives

- Using a cyclic read and process
- Finite state machine design and implementation
- Redesign of an implementation using a different method

Outcomes

- Understand finite state machine design
- Understand finite state machine implementation
- Understand C typedefs and enumerations
- Understand the advantages and disadvantages of each approach

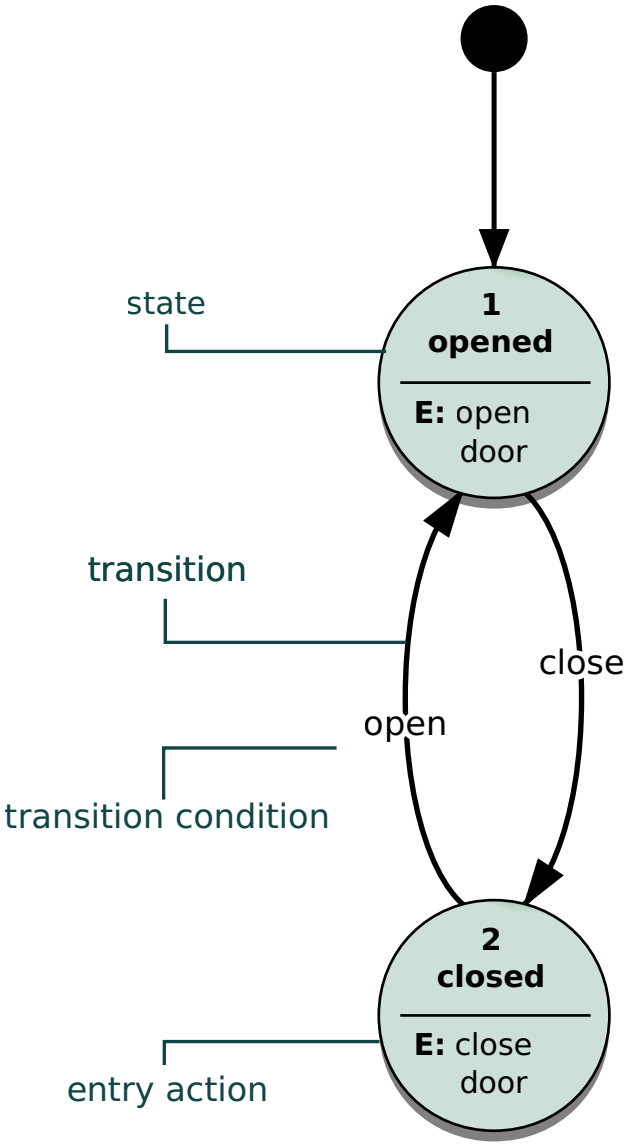
Description

Example of an FSM

Finite-state machines (FSMs) are a subset of state machines where the number of states is finite, which makes them particularly useful for modeling systems with a limited number of discrete states and deterministic behavior.

Based on the current state and a given event, the FSM performs state transitions and produces outputs. This is best explained with a simple example.

State Diagram Example



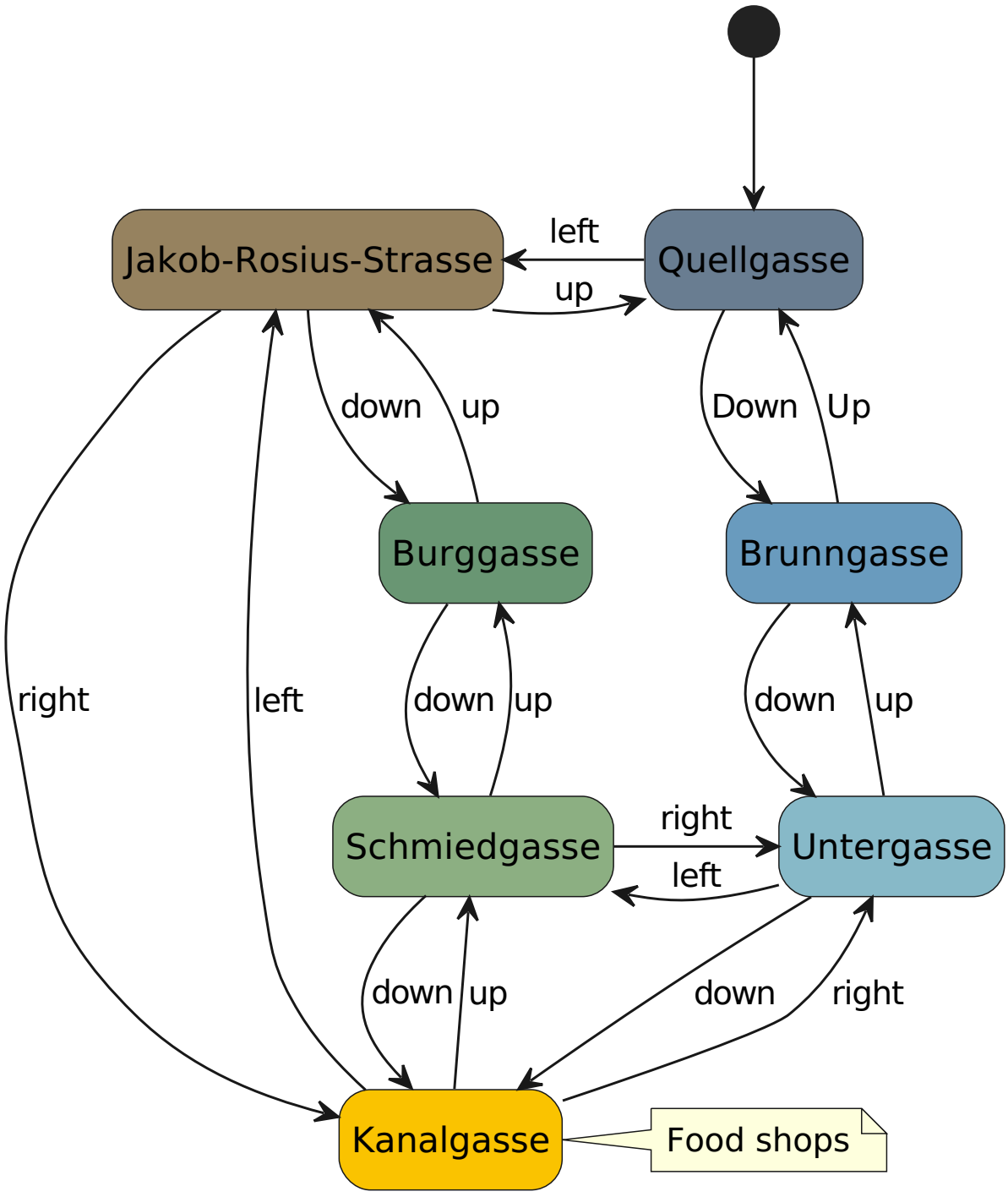
State Diagram Example, source: Wikipedia/Finite-state-machine

An example door FSM can have two states: open and closed. These states are linked by transitions. The transition conditions are the events to open/close the door. The system is then fully defined, as each state has at least one input and one output transition.

Exercise description

The aim of this exercise is to implement a state machine using different methods. The state machine represents several roads to travel from the school in Quellgasse to the various grocery stores in Kanalgasse. This can be seen in the figure below.

Path State Machine Diagram



Path Description

The events are the rising edges of the corresponding joystick position on the mbed application shield. At each state change, the current state is transmitted to the computer via the UART together with the corresponding state number and the name of the street. If no state change occurs, the current state information is retransmitted periodically (every second). Define a message format that you find useful.

Tasks

As described above, this exercise will be solved using several methods. Each method of implementing a state machine has its own advantages and disadvantages. The methods will be explained in this section and then implemented in the subsequent section.

The switch-case method

A rather naive implementation of an FSM is to implement the state machine using the switch statement. The switch statement checks a variable which holds the current state. This variable can be changed accordingly in each state. An enumeration is used to name the states without the cost of a string compare.

Typedef enumeration

```
typedef enum { DOOR_OPENED = 0, DOOR_CLOSED, DOOR_MAX_STATES } fsm_statesType;
```

At each state, an evaluation of the transition has to be implemented, either with an if/else or with a nested switch case.

Switch case

```
switch (curState) {
case DOOR_OPENED:
  /* ... */
  if (<EVENT>) {
    curState = DOOR_CLOSED;
  }
  break;
case DOOR_CLOSED:
  /* ... */
  if (<EVENT>) {
    curState = DOOR_OPENED;
  }
  break;
default:
  curState = DOOR_OPENED; // Undefined state, switch to initial state
  break;
}
```

The table-based method

Another way of implementing a state machine is to create a state transition table. The idea is to create a table where the rows represent all the states and the columns represent all the events. For our door control, a state transition table might look like this.

State transition table example

State Transition Table Door

Current States	Transition open	Transition close
OPENED	OPENED	CLOSED
CLOSED	OPENED	CLOSED

In order to implement this table in C, it is possible to create a two-dimensional array. To define the hole table without magic numbers, it is recommended to create an event enumeration.

Event enumeration

```
typedef enum { EVENT_OPEN = 0, EVENT_CLOSE, MAX_EVENTS } fsm_eventsType;
```

By enumerating the States and Events, the table is fully defined and can be implemented as shown in the following example.

Transition table implementation

```
const fsm_statesType fsm_stateTransitionTable[DOOR_MAX_STATES][MAX_EVENTS] = {
    [DOOR_OPENED] = {[EVENT_OPEN] = DOOR_OPENED, [EVENT_CLOSE] = DOOR_CLOSED},
    [DOOR_CLOSED] = {[EVENT_OPEN] = DOOR_OPENED, [EVENT_CLOSE] = DOOR_CLOSED}};
```

Implementation

The first step is to initialise the project and the peripherals used.

Project setup

1. Create a new project.
2. Using the schematics, set all joystick GPIO pins as an input.
3. Read the joystick using polling with a frequency of $20\,\mathrm{Hz}$.
4. Send a message to the computer every second using the UART. There is no specification on how to implement this.
5. Test your implementation.

Having initialised the project and peripherals, the next step is to implement the basic definitions and functions to be used for both methods.

State machine utils

6. Add a header and source file for a state machine module.
7. Add an enumeration where each state has its own element. You can specify the number for each state (street name).
8. Add an enumeration for all required events (joystick positions).
9. Implement a function to send the current state number and street name via UART to your computer. The current state can be passed as an argument.

   Possible Message Format

   ```
   "State <#>: <STREETNAME>\n"
   Example:
   State 0: Quellgasse
   ```

10. Test this implementation. Go through each state. Send the information string to the computer.

The basic definitions are implemented and can be used for both methods. The first method is to implement the state machine using the Switch-Case method.

The switch-case method

11. Add a variable to store the current state.

12. Add an init function and set the variable to the initial state (Quellgasse).
13. Add a state machine process function that changes the state variable with the current joystick position with the switch-case method.
14. Call this function on every event change.
15. Send the current state to your computer on every state change and every second.
16. Test your implementation and answer following questions:

Question

a. What are the advantages/disadvantages of this implementation method?

b. Is it easy to change?

c. Is it easy to extend?

d. How would the change and extension be implemented?

Another option is to implement the state machine using the table-based method.

The table-base method

17. Create the transition table as shown in the task description.
18. Add a variable to store the current state.
19. Initialise this variable in an init function.
20. Implement a process function to change the current state based on the transition table.
21. Send the current state to your computer at every state change and every second.
22. Test your implementation and answer following questions:

Question

a. What are the advantages/disadvantages of this implementation method?

b. Is it easy to change?

c. Is it easy to extend?

d. How would it be implemented?