Embedded Systems Programming Laboratory

State Machines

Alex Hoffman

Technical University of Munich
Chair of Real-Time Computer Systems (RCS)

November 22, 2022

Lehrstuhl für
Realzeit-Computersysteme

State Machine

You'll thank me later

What are state machines?

A state machine is an abstract model that models a finite number of states in which the system can be in exactly one of.

State Machine

Why do we need them?

Like functions help compartmentalize your code for easier structuring and recycling, state machines help to compartmentalize and structure your code's operational state and the transitioning between these states.

State Machine States

What should a state represent?

A state should represent a certain configuration of the system or that the system's current function is different to other system states. Eg. single player game vs multiplayer game or displaying a menu vs displaying the game screen.

How do we implement a state?

A state can contain or be whatever you want it to be. Eg. if we're working with tasks one could have every state as a task. A simple example could just have us `switch`ing dependant on a `state` variable.

State Machine - Basic Example

```c
#define START_MENU 0
#define SETTINGS_MENU 1
#define SINGLE_PLAYER 2
#define GAME_OVER 3

void my_task(void){
    int state = 0;
    while (1){
        switch (state){
        case START_MENU:
            /* code */
            break;
        case SETTINGS_MENU:
        case SINGLE_PLAYER:
        case GAME_OVER:
        default:
            break;
        }
    }
}
```

State Machine States

How do we handle transitions?

This is where your creativity comes into it.
You should be aware that if you're dealing with multiple tasks then you should make sure you're handling transitions in thread safe manner. Eg. if transition signals must be sent between tasks this should be done using queues etc.

Check demo code **here**.

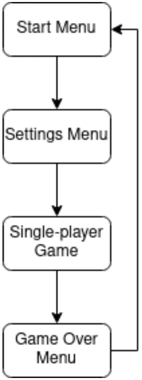
Problem with demo state machine

State transitions only allow for the state machine to cycle through states sequentially

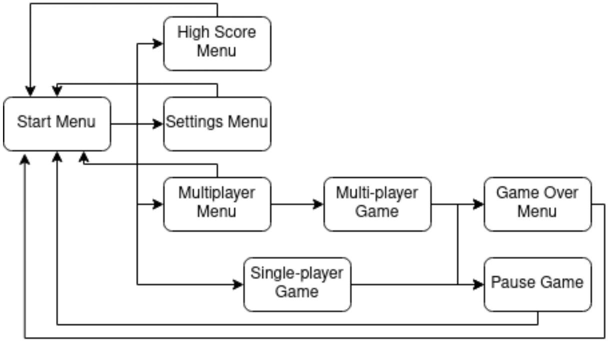
```
void changeState(...){
    switch (forwards) {
        case NEXT_TASK:
            if (*state == STATE_COUNT - 1)
                *state = 0;
            else
                (*state)++;
            break;
        case PREV_TASK:
            if (*state == 0)
                *state = STATE_COUNT - 1;
            else
                (*state)--;
  ...
```

Sequential vs Non-sequential

Sequential



Non-sequential



Transitions

How do we handle transitions?

Transitions handle the exiting of one state and the entering of another specified state. This means that transitions can be used for executing two specific types of code:

- Code to run when a state is exited
- Code to be run when a state is entered

So maybe we should structure our states to have smaller internal states that handle certain parts of transitions.

Transitions

What sub-states can our states have?

- Init: Run only ONCE ever, can be used to initialize anything specific to the state
- Enter: Run when entering the state
- Run: The main part of a state, eg. an internal infinite while loop
- Exit: Run when the state is exited
- Deinit: Run once if the state is permanently exited

If we use these sub-states we can even more easily separate the internal functionality of our code. Eg. entering a menu state? you might want to then reset the variable storing the currently selected menu option.

How do we write this into code?

What sub-states can our states have?

- Init: Run only ONCE ever, can be used to initialize anything specific to the state
- Enter: Run when entering the state
- Run: The main part of a state, eg. an internal infinite while loop
- Exit: Run when the state is exited
- Deinit: Run once if the state is permanently exited

If we use these sub-states we can even more easily separate the internal functionality of our code. Eg. entering a menu state? you might want to then reset the variable storing the currently selected menu option.

How do we write this into code?

You might of picked up on this style of state machine being used in the Git tutorial

See **here**.

But the backbone of the idea is that you have a structure that represents a state, with functions for each sub-state and then a main state task that simply invokes the appropriate function of the current and/or next state at the correct time.

The Git tutorial code keeps track of the state→functions relationships by storing them within the state's struct as function pointers.

Function Pointers

Like everything in C, functions (more specifically their instructions) are located in memory with a specific type known to the compiler, just like regular pointers.

Example

```c
int foo_function(int bar){
  return bar;
    }

int main(null){

  int (*my_function)(int) = &foo_function;

  (*my_function)(20);

  return 0;
}
```

States with Function Pointers

From the Git tutorial code, structure storing each state

```
struct state {
  unsigned char id; /**< The state's ID number */
  void *data; /**< Data stored inside the state */
  char *name; /**< String representation of the state's
      name */
  unsigned char initd; /**< If the state's probe
     function has been run yet */
  void (*probe)(void); /**< The states init function */
  void (*enter)(void); /**< Function that is called
     when going into the state */
  void (*run)(void);   /**< Run function that executes
     while the state is the
                                        current state*/
  void (*exit)(void);  /**< Function run when the state
      is being moved out of*/
};
```

States with Function Pointers

Creating a state

```
unsigned char states_add(void (*probe)(void),
      void (*enter)(void), void (*run)(void),
      void (*exit)(void), int ID, char *name) {

  unsigned char error = 0;
  state_t *ret = calloc(1, sizeof(state_t));
  if (!ret)
    return -ENOMEM;

  (ret->enter) = enter;
  (ret->run) = run;
  (ret->exit) = exit;
  (ret->probe) = probe;
  ...
```

States with Function Pointers

Running states/transitions

```
unsigned char states_run(void) {

  if (state_machine_dev.next_state->id !=
     state_machine_dev.current_state->id) {

    if (state_machine_dev.current_state->exit) /* Exit
     current state */
      (state_machine_dev.current_state->exit)();
    if (state_machine_dev.next_state->enter) /* Enter
     next state */
      (state_machine_dev.next_state->enter)();
    state_machine_dev.current_state =
        state_machine_dev.next_state; /* Change states
     */
  }

  ...
```

States with Function Pointers

Running states/transitions

```
...

  if (state_machine_dev.current_state->run) /* Run
     current state */
    (state_machine_dev.current_state->run)();

  if (state_machine_dev.callback) /** SM callback if
     set */
    (state_machine_dev.callback)();

  return 0;
}
```

A much better example

C file

H file