Efficient State Machines in C

...

José Reyes García Delgado
@Ubidots Hardware and Software Developer

About me

A trained electrical engineer from the Universidad Industrial de Santander, Jose has worked for more than 5 years Integrating and developing solutions for more than 20 building automation and control projects in Colombia. He joined to Ubidots in 2015 to put his two cents in the creation of a new global web IoT platform.

A passionate Colombia fan, Jose loves to see Colombia's football team win and dreams to one day see the label "made in Colombia" printed on technology products sold throughout the world.

In his spare time, Jose passes time with his wife and family, tasting new food and flavors as he travels around the Americas.

Bibliography

Based on three great website posts

- John Santic:

http://johnsantic.com/comp/state.html

- Jacob Beningo:

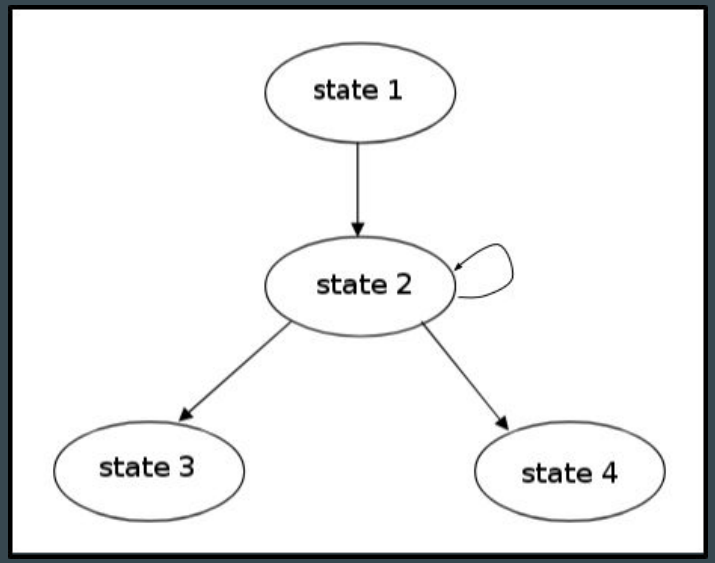
https://www.edn.com/electronics-blogs/embedded-basics/4406821/Function-pointers---Part-3--State-machines

- Joonas Pihlajamaa:

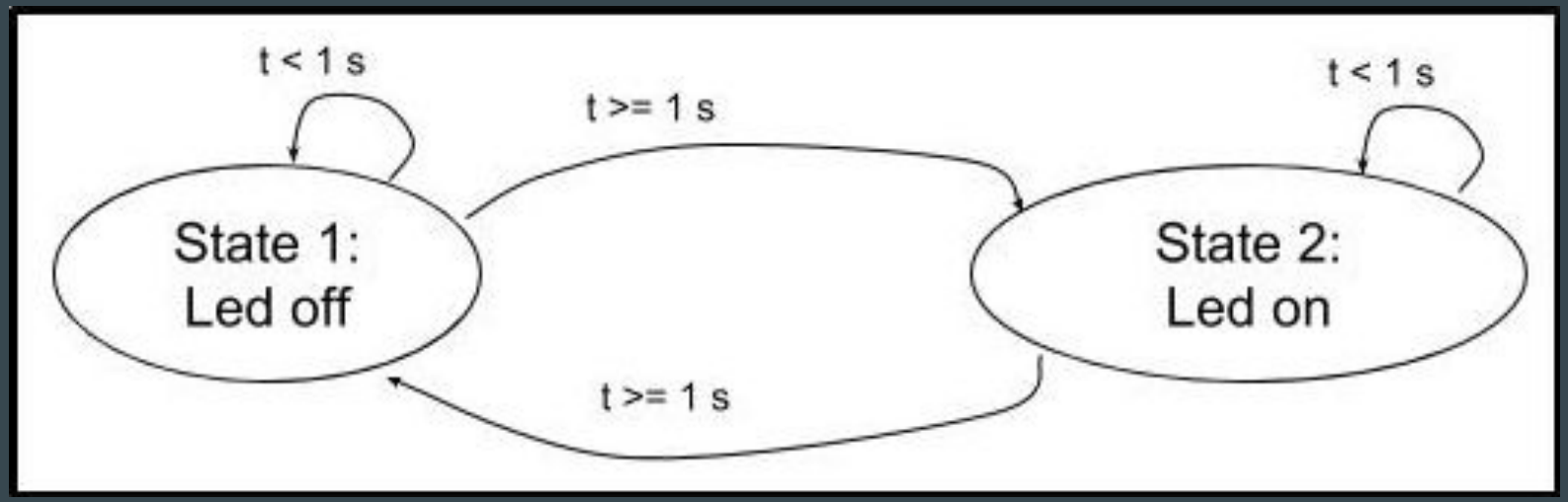
http://codeandlife.com/2013/10/06/tutorial-state-machines-with-c-callback/

Finite State Machine

- Finite State Machines are simply a mathematical computation of a series of cause and events.
- Any FSM must be described before being coded by a state diagram



Finite State Machine: Hello World



Finite State Machine: First try

Create an array with the states and make use of *if-else* statements.

```
/*
   State machine valid states
*/
enum states {
  LED_ON,
  LED_OFF
};

/*
    Initial SM state and functions declaration
*/
enum states state = LED_OFF;

void setup() {
  // put your setup code here, to run once:
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  if(state == LED_OFF) {
    digitalWrite(LED_BUILTIN, HIGH);
    state = LED_ON;
  } else {
    digitalWrite(LED_BUILTIN, LOW);
    state = LED_OFF;
  }
  delay(1000);
}
```

Finite State Machine: Second try

Create an array with the states and make use of *switch-case* statements.

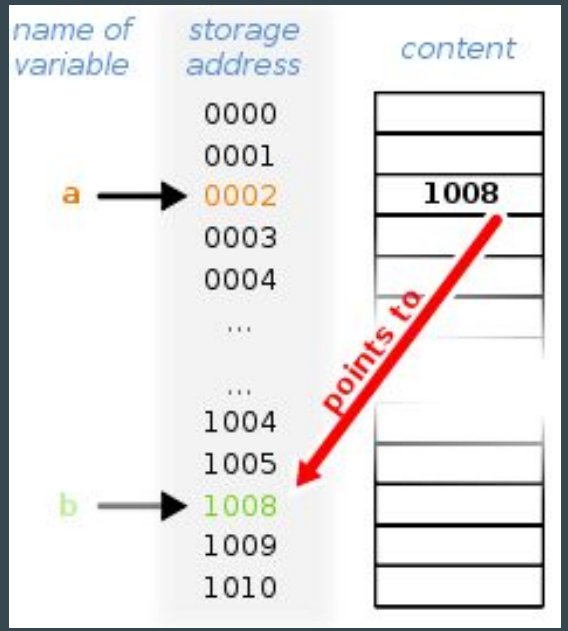
```
/*
   State machine valid states
*/
enum states {
  LED_ON,
  LED_OFF
};

/*
    Initial SM state and functions declaration
*/
enum states state = LED_OFF;

void setup() {
  // put your setup code here, to run once:
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  switch(state) {
    case LED_ON:
      digitalWrite(LED_BUILTIN, HIGH);
      state = LED_OFF;
      break;
    case LED_OFF:
      digitalWrite(LED_BUILTIN, LOW);
      state = LED_ON;
      break;
  }
  delay(1000);
}
```

Problems

- You have to evaluate every *if* or *case* statement until the firmware finds the condition reached.
- Complex firmware routines or FSM will be harder to maintain by a group of developers.
- To implement additional firmware routines, like interruptions, can be really painful.
- A firmware with multiple nested *if* statements is not elegant and professional.

Proposal: Function Pointers

- A pointer references a location in memory, this reference in general is a variable.
- Pointers are basically memory allocation addresses, so they are not limited to reference variables but also set of instructions (functions).



Proposal: Function Pointers

- You can create several functions or methods and just update the pointer address to execute instructions
- Function pointer syntaxis:

```
void (*FuncPtr)(void);
```

Proposal: LookUp Tables

- Additional to the function pointers to avoid the conditionals usage, we also need to store the events and states in a *lookup* table

```
void (*const state_table [MAX_STATES][MAX_EVENTS]) (void) = {
    { action_s1_e1, action_s1_e2 }, /* procedures for state 1 */
    { action_s2_e1, action_s2_e2 }, /* procedures for state 2 */
    { action_s3_e1, action_s3_e2 }  /* procedures for state 3 */
};
```

- With a *lookup table* it will be easier to call the implemented methods or actions.

Finite State Machine: Third try

Create all the available states and define a structure for the lookUp table. Notice that inside the lookUp table will be our function pointer

```
/*************************************
   STATE MACHINE SETUP
 *************************************/

/*
   State machine valid states
*/
typedef enum {
  LED_ON,
  LED_OFF,
  NUM_STATES
} StateType;

/*
   State machine table structure
*/

typedef struct {
  StateType State;

  // Create the function pointer
  void (*function)(void);
} StateMachineType;
```

Finite State Machine: Third try

Declare all the actions or methods and the lookup table. Also, set the initial FSM state.

```
/*
    Initial SM state and functions declaration
*/

StateType SmState = LED_ON;

void Sm_LED_ON();
void Sm_LED_OFF();

/*
   LookUp table with states and functions to execute
*/

StateMachineType StateMachine[] =
{
  {LED_ON, Sm_LED_ON},
  {LED_OFF, Sm_LED_OFF}
};
```

Finite State Machine: Third try

Develop your custom actions methods.
Do not forget to add the states transitions

```
/*
   Custom State Functions routines
*/

void Sm_LED_ON() {
  // Custom Function Code
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);

  // Move to next state
  SmState = LED_OFF;
}

void Sm_LED_OFF() {
  // Custom Function Code
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);

  // Move to next state
  SmState = LED_ON;
}
```

Finite State Machine: Third try

This is the heart of th FSM and where the ‘magic’ happens. Notice that we just call the pointer to the function stored in the lookup table, very easy!!

```
/*
   Main function state change routine
*/

void Sm_Run(void) {
  // Makes sure that the actual state is valid
  if (SmState < NUM_STATES) {
    (*StateMachine[SmState].function) ();
  }
  else {
    // Error exception code
    Serial.println("[ERROR] Not valid state");
  }
}
```

Finite State Machine: Third try

The loop() method now just only need to call our sm_Run() FSM function to perform the FSM logic

```
/************************************
   MAIN ARDUINO FUNCTIONS
 ************************************/

void setup() {
  // put your setup code here, to run once:
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  Sm_Run();
}
```

Finite State Machine: Third try

Advantages:

- If you need to add more states, you have just to declare the new transition method and update the lookup table, the main function will be the same.
- You do not have to perform every if-else statements, the pointer just let's to your firmware to 'go' to the desired set of instructions.
- This is a more C concise and professional way to implement FSM.

Disvatanges:

- You need more statical memory to store the lookup table.

Thanks

...

José Reyes García Delgado
@Ubidots Hardware and Software Developer