Event Driven Programming for Embedded Systems - A Finite State Machine Based Approach

Nrusingh Prasad Dash*, Ranjan Dasgupta†, Jayakar Chepada‡ and Arindam Halder§
Innovation Lab, Tata Consultancy Services Ltd. Kolkata, 700091, India
*E-Mail:*nrusingh.dash@tcs.com,†ranjan.dasgupta@tcs.com, ‡jayakar.ch@tcs.com, §arindam.halder@tcs.com*

***Abstract*—The paper gives a brief overview of event driven program and its relationship with finite state machine (FSM). It proposes a FSM-based framework implemented in C programming language. In Microcontroller Unit (MCU) based tiny embedded system, FSM based software design and event-driven programming techniques are efficient in handling concurrent and asynchronous events usually occur. Finally, the paper states and demonstrates a solution of a system power sequence problem using the same framework as a case study.**

Keywords - Event Driven Programming; FSM; States; State Transitions; MCU.

I. Introduction

Most of the tiny embedded systems respond to external or internal events in some or other way. The external event can be an interrupt, or change of signal level at I/O pins, a message packet coming from other part of the system through some interface, e.g., serial peripheral interface(SPI), inter-integrated circuit(I2C), two wire interface(TWI), or simply an expiry of internal timer. This paper initially discusses the prior art on software implementation of FSM in Section II, subsequently provides the theoretical background of event driven programming paradigm and how the event driven programming problems can be solved using finite state machines(FSM) in Sections III, and IV respectively. In Section V, the paper proposes a FSM framework and narrates a case study where, the same framework has been used to implement an event driven application efficiently and easily on a TI MSP430F1232 MCU based system. Finally, the paper discusses the performance figures of the case study in Section VI

II. State of The Art

There are various approaches taken for the software implementation of finite state machines (FSM). The works presented in [1] and [3] are switch-case driven FSM implementations where, several comparisons are required before execution of the event handler. The number of comparison increases with the number of states and events. The more is the number of comparisons the more CPU cycle is consumed. In the work [4] a table driven event handler hashing approach has been taken to implement the FSMs, but does not separate out the FSM framework and FSM implementation, therefore lacks re-usability. The works [2] and [5], emphasize on model driven FSM generation techniques, but, to maintain the genericness and re-usability, generated code for FSM would require high memory foot-print as well as more CPU cycles and therefore may not be suitable for the embedded systems with very tight memory and cpu horse-power budget. The current work aims at an re-usable, simple and compact FSM framework, which takes minimal CPU cycles and less memory foot-print to implement an FSM problem.

III. Event Driven Programming

The events are mostly generated when user actions are done on a system. The user actions can be a press of a push button or a key pad, touch a touch screen, move or click of a mouse. The events can also be generated from the sensors or devices connected to a system (may be through interrupts or may be form of message packets through a physical interface). Sometimes event may be generated internally, e.g., timeout event or a software exception. Irrespective of the source or type of events, the event driven programming talks about a programming paradigm in which the flow of the program is determined by the events. The actual implementation of event driven programming can be done with any programming language, like C/C++ etc. Broadly these implementations have following sections of programs.

- Event Capture Section
- Event Despatch Section
- Event Handlers Section

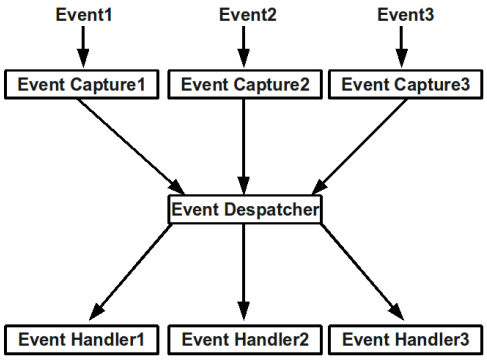

Figure 1. Sections of Event Driven Program

 ISBN:978-1-61208-114-4

A. Event Capture Section

This section of program is responsible for capturing the events, do pre-processing and identifying the event type. The event capture section can be at one place or may be distributed across several interrupt handlers and the main background loop.

B. Event Despatch Section

The responsibility of the dispatcher is to map the events with the respective handler and calls the handler. If there is no associated handler with an event, dispatcher either drops the event or raises an exception.

C. Event Handlers Section

The event handlers implement the activities; those should take place on occurrence of an event.

Many event driven programs are state less, which means when any application finishes processing an event, the application does not require to maintain its earlier event. When the event occurs, the respective handler is just executed. It means it is a state less event driven program where the execution flow is not dependent on the earlier events. On the contrary the other category of event driven programs, where the execution flow is dependent on not only the current event but also the sequence of prior events, called as state full event driven programs. This article discusses about the latter category and how FSM can be used to solve state full event driven programming problems.

IV. Finite State Machine (FSM)

Finite State Machine (FSM) is a model behavior composed of a finite number of states, transitions between those states, and actions.

Finite state machines (FSM) consist of 4 main elements:

- **States** - Define behavior and may produce actions
- **State Transitions** - Switching of state from one to another
- **Conditions** - Set of rules which must be met to allow a state transition
- **Input Events** - Triggers which are either externally or internally generated, which may possibly invoke conditions and upon fulfilling the conditions lead to state transition.

Every FSM has an initial state, which is the starting point. The input events act as triggers, which cause an evaluation of the conditions imposed. On fulfilling those, the current state of the system switches to some other state, which is called as state transition. State transitions, may happen along with the associated actions in most of the cases. The actions can happen, before entering to a state or at exiting the state or while being in the state itself.

V. Case Study - A System Power ON/OFF Sequence

Problem Statement - There is a power key in a system. Initially the system is assumed to be off. When the system is off, if the power key is pressed for 2 seconds, it switches on. When the system is on, if power key is pressed for 2 seconds it switches off. But if the key press time is less than 2 seconds while system is either on or off state, it remains in same state which means there is no state transition. The Fig. 2 is the unified modeling language(UML) state chart representation of the problem stated above.

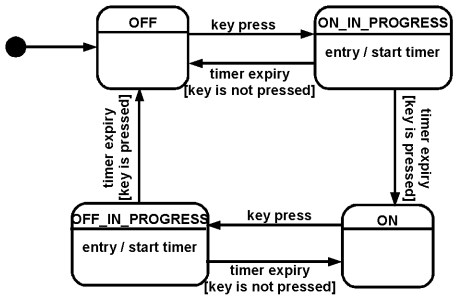

Figure 2. UML State Chart

The problem stated above can be very easily implemented using a FSM framework, that has been proposed in this work.

A. The FSM Framework Implementation

The C program Listing 1 in the appendix is a state transition table-driven implementation of FSM framework, which can be used for a quick and easy implementation of a FSM problem. The framework exposes following interfaces for programmer.

sm_declare_states(fsm_name, list_of_states) - Interface for declaring states. The first parameter is the name of the state machine and the rest of the parameters are the list of states separated by coma.

sm_declare_events(fsm_name, list_of_events) - Interface for declaring events. The first parameter is the name of the state machine and the rest of the parameters are are the list of events separated by coma.

sm_declare_state_machine(fsm_name, initial_state, list_of_state_handlers) - Interface for declaring the state transition table and initializing the state to initial_state.

sm_handle_event(fsm_name, event) - Interface is used for handling the event after the event is captured.

sm_set_state(fsm_context, state) - Interface is used inside the event handlers to set the next state.

sm_set_private_data(fsm_context, private_data) - Interface is used inside the event handlers to set the problem specific private data.

 ISBN:978-1-61208-114-4

sm_get_private_data(fsm_context) - Interface is used inside the event handlers to obtain the reference to the problem specific private data.

sm_define_handle_event(fsm_name) - This is not an interface for the programmer. This macro defines the state machine handling function for the events. Note: It is customary to use this macro without putting a semicolon after it.

sm_declare_handle_event(fsm_name) - This is not an interface for the programmer. This macro declares the state machine handling function. It is customary to use this macro in the header file of the actual state machine implementation ending with a semicolon.

B. FSM Implementation of System Power ON/OFF Sequence

The FSM framework described in previous subsection is used to implement the FSM problem of system power on/off sequence described earlier.

The program Listing 2 in the appendix demonstrates how easily the states and the events can be declared. It is just a matter of using two macros, i. e., **sm_declare_states(fsm_name, list_of_states)** and **sm_declare_events(fsm_name, list_of_events)**. In this case the states are **OFF**, **ON_IN_PROGRESS**, **ON**, **OFF_IN_PROGRESS** and the events are **KEY_PRESS**, and **TIMER_EXPIRY**.

The program Listing 3 in the appendix demonstrates the implementation of the state transition table and the event handlers. The initial state setting and the state transition table definition is done with **sm_declare_state_machine(fsm_name, initial_state, list_of_state_handlers)** interface. The guard conditions, e.g., the check for the key is still pressed at the timer expiry or not is implemented as a condition check within the event handlers. The transition to next state is also done inside event handler using **sm_set_state(fsm_context, state)** interface.

The event capture sections are distributed. In the current problem the event **KEY_PRESS** is captured as polling of the respective pin in the main background loop and the event **TIMER_EXPIRY** is captured in timer interrupt context as timer expiry callback function. The event capture and despatch is demonstrated in program Listing 4 in the appendix. After the events are captured they are despatched to respective event handles using **sm_handle_event(fsm_name, event)**.

Note: The timer implementation and the power key state check program listings are not included or described to maintain the focus on the FSM implementation.

The FSM framework presented in this paper has theoretical commonality with the FSMs presented in [1] and [3] but has novelty in its implementation. It uses a pre-hashed event handling approach where, the current state and event are used as hash-keys to fetch the event handler from table, without requiring comparisons and saves the CPU cycles consumed. Unlike [4], the work presented in this paper separates out the FSM framework and FSM implementation, enhancing the re-usability of the framework. The Listing 1 in the appendix implements a reusable framework, which can be reused for implementing other FSM problems. The CPU cycle consumption and memory foot-print figures are very minimal as discussed in Section VI which proves its suitability for embedded systems applications.

However, the framework presented in this paper is suitable for the FSM problems where, the most of the state and event combinations are handled. Otherwise, the respective table entry consumes memory without doing any useful activity.

VI. CONCLUSION

The FSM framework discussed is not only a very quick and easy way to implement a FSM problem, but also the memory footprint of the generated code is very less and being a function table driven event handling implementation, the execution is pretty fast. These characteristics make the framework very much suitable for the tiny embedded systems application where the memory and processor resources are very scarce. The framework is used for the power sequence problem described in the paper on a MSP430F1232 MCU and the memory footprint is as below.

- 164 bytes of code memory
- 8 bytes of data memory
- 24 bytes of constant memory

The FSM consumes approximately 18 instruction cycles between the event despatch and the event handler is called. The above figures are reported using IAR Workbench [6] v4.

REFERENCES

[1] Miro Samek, *Practical UML Statecharts in C/C++*, 2nd Edition, Newnes.

[2] Ilija Basicevic, Miroslav Popovic, and Ivan Velikic *"Use of Finite State Machine Based Framework in Implementation of Communication Protocols A Case Study"* Sixth Advanced International Conference on Telecommunications, May 9 - 15, 2010

[3] Andrei Drumea and Camelia Popescu, *"Finite State Machines and their applications in software for industrial control"*, 27th International Spring Seminar on Electronics Technology: Meeting the Challenges of Electronics Technology Progress, May 13 -16, 2004

[4] Johannes Weidl, Ren6 R. Klosch, Georg Trausmuth, and Harald Gall, *"Facilitating program comprehension via generic components for state machines"*, Fifth Iternational Workshop on Program Comprehension, March 28 - 30, 1997

[5] Chung-Shyan Liu, and Kuo-Hua Su, *"An FSM-Based Program Generator for Communication Protocol Software"*, Eighteenth Annual International Computer Software and Applications Conference, November 9-11, 1994

 ISBN:978-1-61208-114-4

[6] http://www.iar.com/website1/1.0.1.0/220/1/

APPENDIX

Listing 1. FSM framework

```
#ifndef _SM_FRAMEWORK_H_
#define _SM_FRAMEWORK_H_
/*
File:sm_framework.h
Description: A framework for finite state
    machine implementations
*/
typedef struct sm_context sm_context_t;
typedef void (*sm_handler_t) (sm_context_t *);

#define sm_declare_states(name, ...)     \
typedef enum {                           \
    __VA_ARGS__,                         \
    name##_STATE_COUNT                   \
} name##_state_e

#define sm_declare_events(name, ...)     \
typedef enum {                           \
    __VA_ARGS__,                         \
    name##_EVENT_COUNT                   \
} name##_event_e


#define sm_declare_state_machine(name, st_init,
    ...) \
struct sm_context { \
    void                    *priv; \
    name##_state_e          state; \
    name##_event_e          event; \
    sm_handler_t            *handler; \
}; \
static const unsigned int \
name##_handler[name##_EVENT_COUNT][name##
    _STATE_COUNT] = {__VA_ARGS__}; \
static sm_context_t name##_context \
= {0, st_init, (name##_event_e) 0, (
    sm_handler_t *)name##_handler}

#define sm_set_state(c, s) (c)->state  = (s)
#define sm_set_private_data(c, p) (c)->priv = (
    p)
#define sm_get_private_data(c) (c)->priv

#define sm_define_handle_event(name) \
void name##_handle_event(name##_event_e ev) \
{   sm_context_t *c = &name##_context; \
    c->event = (ev); \
    c->handler[c->event * name##_STATE_COUNT +
        c->state](c); \
    return; \
}

#define sm_declare_handle_event(name) \
void name##_handle_event(name##_event_e ev)

#define sm_handle_event(name, ev) name##
    _handle_event(ev)

#endif /*_SM_FRAMEWORK_H_*/
```

Listing 2. Events and states declaration

```
#ifndef __POWER_STATE_MACHINE__
#define __POWER_STATE_MACHINE__
#include
/*
* File: power_state_machine.h
* Description: States and events declared
*/
sm_declare_states(power,
                  OFF,
                  ON_IN_PROGRESS,
                  ON,
                  OFF_IN_PROGRESS
                  );

sm_declare_events(power,
                  KEY_PRESS,
                  TIMER_EXPIRY
                  );

sm_declare_handle_event(power);

#endif /*__POWER_STATE_MACHINE__*/
```

Listing 3. Power state machine

```
#include <power_state_machine.h>
static void no_action_handler(sm_context_t *
    context);
static void on_key_press_when_off(sm_context_t
    *context);
static void on_timer_expiry_when_on_in_progress
    (sm_context_t *context);
static void on_key_press_when_on(sm_context_t *
    context);
static void
    on_timer_expiry_when_off_in_progress(
    sm_context_t *context);
sm_declare_state_machine(power,
/* initial state */
OFF,
/* event = KEY_PRESS state = OFF */
(unsigned int) on_key_press_when_off,
/* event = KEY_PRESS state = ON_IN_PROGRESS */
(unsigned int) no_action_handler,
/* event = KEY_PRESS state = ON */
(unsigned int) on_key_press_when_on,
/* event = KEY_PRESS state = OFF_IN_PROGRESS */
(unsigned int) no_action_handler,
/*
* event = TIMER_EXPIRY
* state = OFF
*/
(unsigned int) no_action_handler,
/*
* event = TIMER_EXPIRY
* state = ON_IN_PROGRESS
*/
(unsigned int)
    on_timer_expiry_when_on_in_progress,
/*
* event = TIMER_EXPIRY
* state = ON
*/
(unsigned int) no_action_handler,
/*
* event = TIMER_EXPIRY
* state = OFF_IN_PROGRESS
*/
(unsigned int)
    on_timer_expiry_when_off_in_progress);

/*don't put semicolon at the end of the line
    bellow*/
sm_define_handle_event(power)
```

```
41
42 static void no_action_handler(sm_context_t *
       context)
43 {
44     return;
45 }
46 static void on_key_press_when_off(sm_context_t
       *context)
47 {
48     timer_start(TIMER_ID);
49     sm_set_state(context, ON_IN_PROGRESS);
50     return;
51 }
52 static void on_timer_expiry_when_on_in_progress
       (sm_context_t *context)
53 {
54     if(is_power_key_pressed())
55     {
56       sm_set_state(context, ON);
57     }
58     else
59     {
60       sm_set_state(context, OFF);
61     }
62     return;
63 }
64 static void on_key_press_when_on(sm_context_t *
       context)
65 {
66     timer_start(TIMER_ID);
67     sm_set_state(context, OFF_IN_PROGRESS);
68     return;
69 }
70 static void
       on_timer_expiry_when_off_in_progress(
       sm_context_t *context)
71 {
72     if(is_power_key_pressed())
73     {
74         sm_set_state(context, OFF);
75     }
76     else
77     {
78         sm_set_state(context, ON);
79     }
80     return;
81 }
```

Listing 4. Event capture and dispatch

```
1  #include <power_state_machine.h>
2  /*
3  * File: main.c
4  */
5  #define TIMER_ID            1
6  #define TIMER_DURATION      2
7  /*
8  * Timer handler called in interrupt context
9  * captures the timer expiry event
10 */
11 void power_key_timer(void *data)
12 {
13      sm_handle_event(power, TIMER_EXPIRY);
14 }
15 /*
16 * Background loop for capturing key press event
17 */
18 int main(void)
19 {
20     timer_init(TIMER_ID,
21                TIMER_DURATION,
22                TIMER_ONE_SHOT,
23                power_key_timer,
24                NULL);
25     while(1) {
26         if(is_power_key_pressed()) {
27             sm_handle_event(power, KEY_PRESS);
28         }
29     }
30 }
```