Algorithm - Embedded State Machine Design for PSoC using ‘C’ Programming  
(Part I of III)

AN2329

Authors: Somsak Sukittanon, Ph.D., Stephen G. Dame, MSEE
Associated Project: Yes
Associated Part Family: CY8C21xxx, CY8C24xxxA, CY8C24794, CY8C27xxx, CY8C29xxx
Software Version: PSoC® Designer™ 4.2
Associated Application Notes: AN2332, AN2333

Application Note Abstract

Finite state machines are the bread and butter of embedded system design for creating production quality firmware. Some sophisticated state machine compilers may be purchased off the shelf and allow designers to express a system as a collection of high-level state behaviors. However, many of these systems require processor resources that are beyond the tight memory requirements of a PSoC. What is needed is a small state machine kernel and efficient and simple means to be able to implement traditional state machine behavior.

Introduction

Embedded state machines are widely used in modeling of application behavior and design of hardware digital systems and embedded controllers. System behavior can be concisely expressed as a logical organization of states, events, and state-transition actions based on system events. In this application, a tightly written C kernel has been developed (less than 300 bytes) that executes a table-driven state machine consisting of state, events and actions. Each state descriptor occupies 5 bytes, which yields an efficient means to expand the system behavior in a table-driven approach. Complex behavior can be represented in a flexible way by coding logical events and actions using freeform ‘C’ code functions.

This Application Note is first in a three-part series that first describes the theory of implementation of a simple state machine that can easily fit in any PSoC design.

Part II (AN2332) describes a code generator and example program to exhibit a simple state machine using the CY3210-PSoCEval1 demonstration platform. This note extends the functionality of the ADC conversion and LCD display example to a state machine-based approach.

Part III (AN2333) of this series gives an example that shows how a simple state machine project can be interfaced to Microsoft Visual Basic® using the PSoC serial port UART running at high speed.

State Machine

Embedded state machine implementation has been used in several projects at Virtual DSP. The main advantage of using a state machine in embedded design, including other platforms, is its flexibility to add, delete, or change the flow of the program without impacting the overall system code structure. This is very important for developers when they try to develop the big project, e.g., ten thousand lines of codes.

In this Application Note, we will discuss implementation of the state machine for use in a PSoC based on C language. The code generator and advanced example project are given in part II of this Application Note series (AN2332).

Before we dive into implementation, let’s create one simple example. Imagine we want to build a counter such that the counting number changes based on turning a digital rotary encoder. A reset button is used to set the counter back to zero. According to this example, the state machine diagram can be depicted in Figure 1. After powering on the system, the state machine enters STATE_IDLE. If the encoder is rotated (EVENT_ENCODER_ROTATE), the state machine leaves the current state to the different state, STATE_NUMBER_0. During this state change, the system initializes registers and buffers (ACTION_COUNTER_INIT), and displays a message on the LCD during the exit of the previous state (SF_PRINT_LCD).
While in STATE_NUMBER_0, the system checks for two possible events that can cause a state change:

1. If the user continues to rotate the encoder, the system still stays in the same state. However, it updates the buffers (ACTION_COUNTER_UPDATE), and displays the new number on the LCD.

2. If the user presses the reset button (EVENT_RESETBUTTON_PUSH), the system prompts the confirmation message (ACTION_RESETCOUNTER_INIT) and the system transitions from STATE_NUMBER_0 to STATE_NUMBER_1.

When the system is in STATE_NUMBER_1, the system monitors for a confirm/cancel answer. Both events change the system back to STATE_NUMBER_0. The only difference is the action function. If the reset process is confirmed (EVENT_RESET_CONFIRM), then the system reinitializes the register and buffer, like at the beginning. If the reset process is cancelled, the system does nothing, and the old number is displayed back on the LCD.

As we see from this example, our state machine is composed of 1) STATES, 2) EVENTS, 3) ACTIONS, and 4) SPECIAL FUNCTIONS (or exit function). Implementation is coded in the form of switch/case and a lookup table. In the next section, we describe how to create each table.

Create State Machine Tables (See main.c, statemachine.h, and statemachine.c)

In the main.c (see Figure 2), the system starts with initializing the PSoC, e.g., UART, PORTS, SPI, or ADC. StateMachine_Init function then sets the state machine to the first state before entering finite polling. Typically, the time that the system spends in one loop (each state) is very short, unless an event and action occur. While the CPU is polling the events, it also accepts the possible interrupt, such as ADC, etc.

StateMachine.h (see Figure 3) contains the list of states (3 total states in this example). The value of STATE_NUMSTATES indicates the total states, 3, and used in assigning the size of state transition matrix. The description of the StateMachine_Init function is described in statemachine.c. Basically, the eCurrentState (an ENUM type) is declared and assigned with the first state value.

Next, the state descriptor is created (see Figure 4). The descriptor is a structure type containing information of the current state, i.e., state number, how many events for this state, and the pointer pointed to the state table. As shown in Figure 4b, StateTransitionMatrix has 3 rows and the order must be identical to state declaration, Figure 3a. The number of events is computed by dividing the size of the current state table by 5. And the last member is the address of the beginning of a state table, e.g., ENL_NUMBER_0 for STATE_NUMBER_0.

As shown in Figure 5, a state table contains information about EVENT, ACTION, NEXT STATE, and EXIT FUNCTION of outgoing direction in each state. From Figure 1, there are 1, 2, and 2 flowing out events for STATE_IDLE, STATE_NUMBER_0, and STATE_NUMBER_1, respectively. To create the table, the structure, called EVENT_NODE_T, is declared and shown in Figure 5a. This structure uses 5 BYTES to describe each list, 1 BYTE for each enum EVENTS_T, STATE_T, and ACTIONS_T and 2 BYTES for pVectorFunc. Using Figure 1, we can fill out the list as shown in Figure 5b. Remember, only the arrow with outgoing direction is listed.

Figure 1. Example of a State Machine with 3 States

Create State Machine Tables (See main.c, statemachine.h, and statemachine.c)
After we create the table, the state machine is ready to run. At this moment, let’s assume we already filled out the list of events and actions. The core of the state machine, as shown in Figure 6, starts with checking if any one event in the current state is activated. If nothing happens, it continually checks the next event until the last event and starts over again. If one event happens in the middle of this event polling, the state machine checks where the next state is and executes the action and exit function corresponding to that event.

**Figure 2. Simple main.c for PSoC State Machine Implementation**

```c
void main()
{
    // Initialize
    // Insert code for UART, PORTS, ADC, or LCD initialization

    StateMachine_Init(); // initialize statemachine

    while (1) // Main loop
    {
        StateMachine(); // core statemachine
    }
}
```

**Figure 3. (a) State Declaration and (b) State Initialization**

```c
//state declaration
typedef enum
{
    STATE_IDLE,
    STATE_NUMBER_0,
    STATE_NUMBER_1,
    STATE_NUMSTATES
} STATE_T;
```

```c
STATE_T eCurrentState;
```

```c
typedef const struct {
    STATE_T eCurrState;
    BYTE bNumEvents;
    EVENT_NODE_T *pEventList;
} STATE_DESCRIPTOR_T;
```

```c
STATE_DESCRIPTOR_T StateTransitionMatrix[STATE_NUMSTATES] = {
    {STATE_IDLE, elements_of(ENL_IDLE), ENL_IDLE},
    {STATE_NUMBER_0, elements_of(ENL_NUMBER_0), ENL_NUMBER_0},
    {STATE_NUMBER_1, elements_of(ENL_NUMBER_1), ENL_NUMBER_1},
};
```

**Figure 4. (a) State Descriptor and (b) State Transition Matrix**

```c
typedef void const (*pVectorFunc)(void);
void StateMachine_Init(void)
{
    eCurrentState = STATE_IDLE;
}
```
Create Events, Actions, and SF_FUNCTION
(See event.c, action.c, and specialfunction.c, also Header Files)

The list of events (see Figure 7) and actions (see Figure 8) are very similar. First, the enumerator type is created. The table is then created using switch/case structure. If the event happens, the event_Check function returns TRUE or FALSE. We should note that for the same event, it cannot be used twice in the same state table. When using EVENT_TRUE, the state machine always executes its action and exit function. Normally, it is placed in the last row when we want the state machine to do something for each state transition.

The action table is very similar to the previous event table. However, the same action can be used twice in the state table. ACTION_NOP does not execute anything. Sometimes, if we find that several functions in action tables are doing the same thing, e.g., clear the key press flag, print to LCD, or reset an idle timer, those redundant operations can be written as the exit function (for example SF_PRINTLCD in this code).
Summary
We have discussed the high-level structure for state machine design within embedded system design for PSoC. 'C' code examples are included. The advantage of this approach is not only its simplicity, which is based on table lookup, but also the ability to add, delete, or change the behavior in each state without significantly affecting other states. The methods discussed in this note are only based on polling. However, it is possible to incorporate an interrupt approach to the state machine. In part II of this Application Note series, we introduce a MATLAB® tool for generating the required state machine files (i.e., the 'C' and header files of event, action, state machine, and special function). After sketching the diagram, these files will be generated and can be used in a PSoC project, or on any 'C' platform with minor modifications.

Figure 6. State Machine Code

```c
void StateMachine(void)
{
    BYTE  bEvents;
    EVENT_NODE_T  *pENL, *pen;
    for(bEvents = 0; bEvents < StateTransitionMatrix[eCurrentState].bNumEvents; bEvents++) {
        pENL = StateTransitionMatrix[eCurrentState].pEventList;
        pen = &pENL[bEvents];
        if(event_Check(pen->Event)) {
            eCurrentState = pen->NextState;
            Action(pen->Action);
            pen->SpecFunc();
            break;
        }
    }
}
```
typedef enum
{
    EVENT_TRUE,
    EVENT_ENCODER_ROTATE,
    EVENT_RESETBUTTON_PUSH,
    EVENT_RESET_CONFIRM,
    EVENT_RESET_CANCEL,
    EVENT_NUMEVENTS
} EVENTS_T;

BYTE event_Check(EVENTS_T event)
{
    switch(event) {
        case EVENT_TRUE:
            return(1);
            break;
        case EVENT_ENCODER_ROTATE:
            //insert code here
            // if (encoder is rotated)
            // return(1);
            break;
        case EVENT_RESETBUTTON_PUSH:
            //insert code here
            // if (resetbutton is pushed)
            // return(1);
            break;
        case EVENT_RESET_CONFIRM:
            //insert code here
            // if (reset is confirmed)
            // return(1);
            break;
        case EVENT_RESET_CANCEL:
            //insert code here
            // if (reset is cancelled)
            // return(1);
            break;
        default:
            return(0);
            break;
    }
    return(0);
}
typedef enum
{
    ACTION_NOP,  
    ACTION_COUNTER_INIT,
    ACTION_COUNTER_UPDATE,  
    ACTION_RESETCOUNTER_INIT,  
    ACTION_NUMACTIONS
} ACTIONS_T;

void Action(ACTIONS_T action)
{
    switch(action)
    {
        case ACTION_NOP:
            break;
        case ACTION_COUNTER_INIT:
            //insert code here
            //initialize counter
            break;
        case ACTION_COUNTER_UPDATE:
            //insert code here
            //update counter
            break;
        case ACTION_RESETCOUNTER_INIT:
            //insert code here
            //initialize counter reset
            break;
        default:
            break;
    }
}

Figure 8. Action Table
About the Authors

Name: Somsak Sukittanon
Title: Principal R&D Engineer
Background: Dr. Sukittanon is a principal R&D engineer at Virtual DSP Corporation. He has led several investigations in the areas of software development of signal processing and embedded systems. He is also a part-time lecturer at University of Washington. He has taught many classes, e.g., circuit analysis and DSP embedded design. He is the recipient of the 2005 outstanding teaching award for Electrical Engineering Department at the University of Washington.

Name: Stephen G. Dame
Title: President and CEO
Background: Mr. Dame is president and founder of Virtual DSP Corporation, which is a research and development and manufacturing company in the areas of digital signal processing, wireless and internet computing devices. He holds a masters degree in electrical engineering and is a successful entrepreneur with more than 25 years of experience in developing products in the medical, aerospace and consumer electronic markets.

Contact: somsak@virtual-dsp.com

Contact: steve@virtual-dsp.com

In March of 2007, Cypress recataloged all of its Application Notes using a new documentation number and revision code. This new documentation number and revision code (001-xxxxx, beginning with rev. **), located in the footer of the document, will be used in all subsequent revisions.

PSoC is a registered trademark of Cypress Semiconductor Corp. "Programmable System-on-Chip," PSoC Designer, and PSoC Express are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.