BULETINUL INSTITUTULUI POLITEHNIC DIN IAŞI
Publicat de
Universitatea Tehnică „Gheorghe Asachi” din Iaşi
Tomul LV (LIX), Fasc. 2, 2009
Secţia
AUTOMATICĂ şi CALCULATOARE

A FINITE STATE MACHINE MODEL USED IN EMBEDDED SYSTEMS SOFTWARE DEVELOPMENT

BY

ANDREI STAN, NICOLAE BOTEZATU, LUCIAN PANDURU and ROBERT GABRIEL LUPU

Abstract. Finite state machines are among the most used building blocks in the embedded systems design for creating quality firmware. The custom implementation of finite state machines consumes time and is prone to error. A better approach is to design an abstract engine (kernel or core) that runs a finite state machine based on an appropriate description. By using such an engine, an embedded designer would have only to specify the actions, checks and transitions in terms of simple functions and tables and let the engine to take care of all the housekeeping work needed to run the finite state machine.

Key words: embedded systems, finite state machines, microcontroller.

2000 Mathematics Subject Classification: 68N19, 68P99.

1. Introduction

This paper presents and discuses the implementation and the evaluation stages of a software finite state machine (FSM) engine (core or kernel) in order to use it in embedded systems software development process.

The main advantage of using a state machine in embedded design consists in its flexibility to add, delete or change the flow of the program without impacting the overall system code structure [1].

2. Implementation

The proposed model for a finite state machine engine is defined by states, events or conditions, transitions and action functions. Each of these components and their implementation modes are described as follows.

Each state is described by the FSME_STATE structure type, presented in Fig. 1, which contains information about the action function, number of transitions from this state and a reference to the transitions table. The action function information is represented by a pointer to function, taking into account that this is a Moore model of a finite state machine.

```
// function pointer type – for action (output)
functions
typedef void (*FSME_PF) (void);
// state type
// describes a FSM state using the following fields:
// Action- a pointer to an action (output) function;
// the outputs of the FSM that are activated in this
state
// TransNO - the number of transitions from the
state
// Trans - an array that contains the actual
transitions infromation: pairs (event, next state)
typedef  struct {
          FSME_PF Action;
          uint8 TransNO;
          FSME_TRANS  * Trans;
} FSME_STATE;
```

Fig. 1 − State structure type.

The transitions are represented by the FSME_TRANS structure type presented below in Fig. 2. It contains a pointer to a function that returns a Boolean value associated with the evaluation of a condition or the occurrence of an event. This structure also contains the code of the state that the finite state machine goes into if the condition is true or the event has occurred.

```
// function pointer type – for event update functions
typedef uint8 (*FSME_PF_EV) (void);
// transition type
// describes a transition using the following fields:
// Event - a pointer to an event (input) function
// NextState - the next state for the transition
typedef struct  {
          FSME_PF_EV Event;
          uint8 NextState;
} FSME_TRANS;
```

Fig. 2 − Transition structure type.

The last specific structure type for this model is FSME_FSM structure presented in Fig. 3. The structure holds data that describes an instance of a finite state machine that runs on an embedded system. It consists of a flag that indicates if the finite state machine is stopped or it is running, a flag that indicates a change in the state of the finite state machine and fields that hold the current state, the number of states of the finite state machine and a reference to an array that contains the states and transitions of the finite state machine.

Also, the structure contains a field for the number of transitions from the current state of the finite state machine and a reference to an array with the transitions form current state. The last two fields are added in order to optimize the code size and also the execution speed. By keeping this information in the finite state machine structure it is avoided the complex access to these fields that should be done by a sequence of instructions. This optimization decision reduces the code size and the execution speed but uses some extra memory space for these fields.

```
// FSM type
// describes a FSM using:
// Enable - a flag that indicates the state of FSM: enabled
or disabled
// CurrentState - the current state of the FSM
// StatesNO - the number of states of the FSM
// StateChanged - flag that indicates a state change
// States   - an array that contains the states and
transitions of the FSM
// TransNO - the number of transitions from current state
// Trans - a reference to an array with the transitions form
current state
typedef struct {
            uint8 Enable;
            uint8 CurrentState;
            uint8 StatesNO;
            uint8 StateChanged;
            FSME_STATE  * States;
            uint8 TransNO;
            FSME_TRANS  * Trans;
} FSME_FSM;
```

Fig. 3 − State machine structure.

A finite state machine is formed from a set of FSME_TRANS arrays that contain information about the transitions from every state and a FSME_STATE array that contains all the states of the finite state machine and references to the transition arrays. Also a FSME_FSM variable is used to wrap all the data and all the references needed by the FSM core. The function for state update of the finite state machine is presented in Fig. 4.

```
static FSME_TRANS  *  _t;
static FSME_STATE  *  _s;
static void FSME_UpdateState( FSME_FSM * F )
{
    uint8 _i = 0;
    uint8 _n;
// set a variable to point to current transition table
    _t = F->Trans;
// set a variable to current value of transitions count for
current state
    _n = F->TransNO;
// loop for all possible transitions from current state
    for ( ; _i < _n ; _i++ )
    {
// check if the events have occured (conditions are true
for a transition to take place)
        if ( FSME_EVENT_TRUE ==  _t[ _i ].Event() )
        {
// update current state
            F->CurrentState = _t[ _i ].NextState;
// get a pointer to current state
            _s =  & ( F->States[ F->CurrentState ] );
// update current transition table according to current
state
            F->Trans = _s->Trans;
// update current transitions number according to current
state
            F->TransNO = _s->TransNO;
// set state changed flag
            F->StateChanged = FSME_STATE_CHANGED;
// leave the for loop and function
            break;
        }
    }
}
```

Fig. 4 − Update state function.

The function is called with a reference to the FSME_FSM instance of the finite state machine. It cycles trough the transitions array associated with the current state and checks for the occurrence of the events associated to the transitions. If an event arises, the state of the FSM is updated according to the state array and the change in state is signaled by asserting the StateChanged flag. Once an event is processed, the function exits even if there are other events pending. This feature can be used in complex finite state machines to prioritize the transitions based on application requirements.

In Fig. 5, the main function of the FSM engine and the action function are presented. FSME_Action executes the associated action function. FSM_Run calls the two functions presented above only if the Enable flag of the pointer F parameter is set.

```
void FSM_Run( FSME_FSM * F  )
{
    if ( FSME_DISABLE == F->Enable )
    {
// TODO: may reset the FSM into initial state and
deactivate outputs
        return;
    }

    FSME_UpdateState( F );
    FSME_Action( F );
}

static void FSME_Action( FSME_FSM * F )
{
    F->States[F->CurrentState].Action();
}
```

Fig. 5 − Run and action functions code.

The FSM_Run function can be pooled in an loop for the occurrence of the events associated to the current state or can be integrated in a task switching context based on the needs of the application.

Fig. 6 presents two auxiliary functions to the engine: Enable and Disable. A call to these functions sets or resets the Enable flag in the F structure accordingly. This feature offers the possibility of enabling or disabling individual finite state machines in an embedded software design. In this way it is enabled the efficient design with multiple finite state machines, allowing, for example, the design of hierarchical state machines.

```
void FSM_Enable( FSME_FSM * F  )
{
    F->Enable = FSME_ENABLE;
}

void FSM_Disable( FSME_FSM * F  )
{
    F->Enable = FSME_DISABLE;
}
```

Fig. 6 − Enabling and disabling FSM code.

The advantage of this approach is not only its simplicity and program flow organization, which is based on table lookup, but also the ability to add, remove or update the behavior in each state without significantly affecting other states [1].

The developed finite state machine engine occupies only 230 bytes of code memory. Each state occupies 5 bytes of memory and each transition consumes 3 bytes of memory. The finite state machine structure occupies 9 bytes of memory.

3. Model Evaluation

In order to evaluate the proposed model for a finite state machine an environment was set up for this purpose. The evaluation environment is targeted to 8-bit embedded microcontrollers, specifically the ATMega family of microcontrollers from Atmel [2].

The evaluation environment uses the following components: a finite state machine module built using the proposed model, a C compiler [3] for the ATMega microcontrollers – IAR Embedded Workbench, and the simulator [4] integrated in the IAR IDE.

A sequence detector is implemented using the proposed model for a finite state machine. This type of sequencial system is an abstraction for many practical problems that may be solved using the finite state machine approach. This finite state machine is very simple: its implementation using the proposed model has very simple functions for input checking and output generation. These are simpler functions compared to the main function FSM_Run of proposed model. This fact allows a better estimation of the complexity (measured using cycles count taken for its execution) of the FSM_Run function compared to the rest of the code. A block diagram of the sequence detector system is presented in Fig. 7.

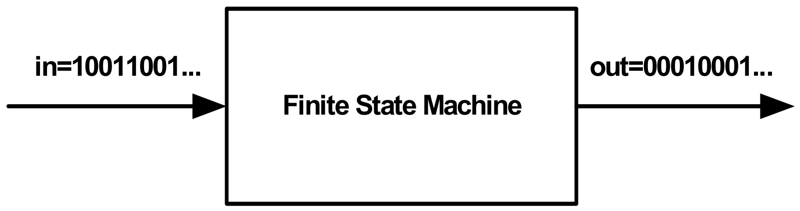

Fig. 7 − Block diagram of sequence detector.

The sequence detector is designed to detect the {0,0,1} sequence of binary values present at its input. If this sequence is present at the input, the system drives its output to the value 1, otherwise the output has the value 0. The states graph for the sequence detector finite state machine is presented in Fig. 8.

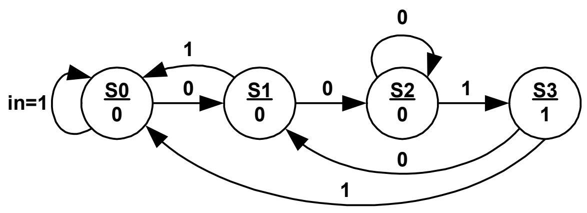

Fig. 8 − State graph of sequence detector.

For each state in the states graph, a table of transitions is defined. Each element of this table has two fields: a pointer to a function that evaluates a condition or check for an event and the state that the finite state machine goes into if the condition is true or the event has occurred, as presented in Fig. 9.

```
// state transitions for each state
static __flash FSME_TRANS FSM1_S0_TRANS[] =
{ { FSM1_EventsUpdate0, FSM1_S1 } };
static __flash FSME_TRANS FSM1_S1_TRANS[] =
{ { FSM1_EventsUpdate0, FSM1_S2 },
  { FSM1_EventsUpdate1, FSM1_S0 } };
static __flash FSME_TRANS FSM1_S2_TRANS[] =
{ { FSM1_EventsUpdate1, FSM1_S3 } };
static __flash FSME_TRANS FSM1_S3_TRANS[] =
{ { FSM1_EventsUpdate0, FSM1_S1 },
  { FSM1_EventsUpdate1, FSM1_S0 } };
```

Fig. 9 − Transitions tables for each state.

For the case of the sequence detector finite state machine, the functions that verify the value of the input variable are presented in Fig. 10.

```
static uint8 FSM1_EventsUpdate0( void )
{
  return ( in == 0 );
}
static uint8 FSM1_EventsUpdate1( void )
{
  return ( in == 1 );
}
```

Fig. 10 − Condition checking functions.

The states table for the sequence detector finite state machine is presented in Fig. 11. Each state is described by its action function, the number of transition from the state and a table that contains the transitions as pairs of a condition checking function and the code for the next state.

```
static void FSM1_ActionClr( void )
{
  if ( FSM1.StateChanged ==
  FSME_STATE_CHANGED )
{
//actions to be executed only once in this state at the
state change
    out = 0;
    // reset state changed flag
    FSM1.StateChanged =
    FSME_STATE_NOT_CHANGED;
  }
  else
  {
    // actions to be executed continuously in this state
  }
}

static void FSM1_ActionSet( void )
{
  if ( FSM1.StateChanged  ==
  FSME_STATE_CHANGED )
   {
// actions to be executed only once in this state at the
state change
    out = 1;
    // reset state changed flag
    FSM1.StateChanged =
    FSME_STATE_NOT_CHANGED;
  }
  else
  {
    // actions to be executed continouslly in this state
  }
}
```

Fig. 11 − States table.

```
// state outputs and transitions; entire table
static __flash FSME_STATE FSM1_STATES[] =
{ { FSM1_ActionClr, 1, FSM1_S0_TRANS},
  { FSM1_ActionClr, 2, FSM1_S1_TRANS},
  { FSM1_ActionClr, 1, FSM1_S2_TRANS},
  { FSM1_ActionSet, 2, FSM1_S3_TRANS}
};
```

Fig. 12 − Action functions.

The action functions for the sequence detector finite state machine are presented in Fig. 12. Each output function has two control paths. One is for actions that are executed only once at the state change. The other is for actions that are executed continuously as long as the finite state machine stays in the same state.

The instance of the finite state machine that implements the sequence detector is presented in Fig. 13.

The finite state machine fields are initialized as follows: the state machine is enabled (FSME_ENABLE), initial state is FSM1_S0, the number of states is set to FSM1_STATES_NO, the flag for state change is set to FSME_STATE_CHANGED in order to force the execution of the action function for initial state and finally the states table is set to FSM1_STATES. The last two fields are set to the number of transitions from initial state (1) and the transitions table for initial state.

In order to evaluate the performance of the implementation, multiple configurations of the application were designed and built. In this way, the impact of various factors over the performance of the application may be evaluated.

```
FSME_FSM FSM1 =  { FSME_ENABLE,
                   FSM1_S0,
                   FSM1_STATES_NO,
   FSME_STATE_CHANGED,
                       FSM1_STATES,
                  1,
                  FSM1_S0_TRANS
                  };
```

Fig. 13 − Instance of a finite state machine.

The finite state machine application is built and compiled in four different configurations:

a) Configuration 1: finite state machine information stored in SRAM as data and compiler optimizations set to low level;

b) Configuration 2: finite state machine information stored in SRAM as data and compiler optimizations set to high level;

c) Configuration 3: finite state machine information stored in FALSH as data and compiler optimizations set to low level;

d) Configuration 4: finite state machine information stored in FLASH as data and compiler optimizations set to high level;

The memory usage obtained after the compilation process of each configuration is presented in Fig. 14. The SRAM memory of the microcontrollers used in embedded application is a valuable resource. Its size is in general smaller than the size of FLASH memory. Depending on the specific application needs, various designs may choose to store finite state machine information into SRAM or FLASH memory. For the sequence detector, the impact of this decision on memory usage is represented in Fig. 14.

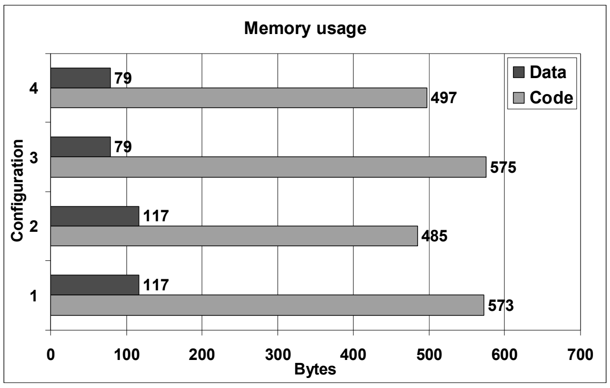

Fig. 14 − Memory usage.

In order to evaluate the execution time of the finite state machine model, a basic script for the IAR simulator [3] was developed. The script measures the number of cycles between successive calls of the function FSM_Run (&FSM1). The input sequence used in simulation process has values that activate all possible transitions of this finite state machine.

The average cycle count obtained after the simulation process of each configuration is presented in Fig. 15. The cycle count is slightly larger for configurations that stored the finite state machine information into FLASH memory. This fact occurs because reading data from FLASH memory takes more cycles than reading data from SRAM memory.

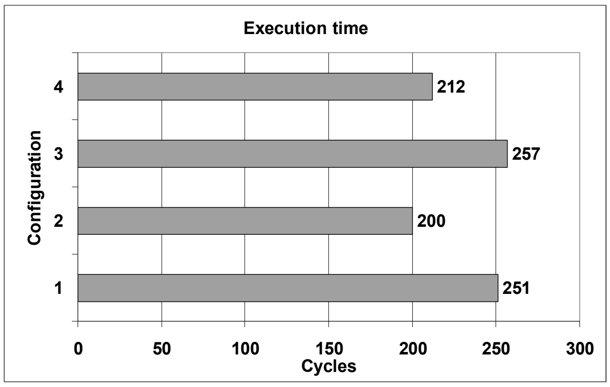

Fig. 15 − Execution time.

The simulation script also computes the number of cycles necessary for each transition that occurred while running the simulation process. The number of cycles for each transition for the configuration 2 is presented in Table 1 and for the configuration 4 is presented in Table 2.

Table 1
Number of Cycles for Configuration 2

Crt state / Next state	S0	S1	S2	S3
S0	156	202	0	0
S1	239	0	202	0
S2	0	0	156	203
S3	239	202	0	0

Table 2
Number of Cycles for Configuration 4

Crt state / Next state	S0	S1	S2	S3
S0	160	216	0	0
S1	255	0	216	0
S2	0	0	160	217
S3	255	216	0	0

This feature may be used to validate the transition table of a finite state machine. In the tables above, a zero value means that the respective transition never occurred and a nonzero value means that the respective transition occurred during simulation process and lasted for the number of cycles presented in the table.

4. Conclusions

The proposed software approach for the implementation of finite state machines has some specific properties that must be underlined.

All finite state machines from a design have uniform treatment. They are instances of the same structure type that abstracts and encapsulates the characteristics of a finite state machine.

All finite state machines from a design share the common code for state update function, for running function and for the action function. Only specific information must be specified and described for a finite state machine: states, transitions tables, condition check functions and action functions.

The proposed solution is portable because does not use specific features of a specific microcontroller.

The presented approach proves itself not very well fitted to 8 bit microcontrollers. This conclusion arises from the high number of cycles necessary to run the code. This is a consequence of extensive use of pointers, which in the presented case are represented on 16 bits. The resulted compiled code is large and uses many instructions in order to operate with 16 bit values.

Despite the C source code simplicity, the compiled result for a 8 bit microcontroller yells a large instruction number.

The proposed further work is aimed at porting the C code to 16 bit and 32 bit microcontroller architectures: MSP430 or HCS12 for 16 bit and STM3 for 32 bit. An evaluation environment will be created in order to measure the performance.

For complex finite state machines (with more states and more transitions per state with possible more complex conditions) a statistical analysis on inputs may be performed in order to find an optimal order for transitions in the transition table for each state: more probable transitions to be placed at the beginning of the transition table. This analysis highly depends on the specific application.

The structure of the finite state machine may be modified to fit various application needs. For example, may be added a timeout value for an entire finite state machine or for each state of a finite state machine. This timeout value may be used to trigger a safe transition in the case of a dead lock.

Received: March 25, 2008

"Gheorghe Asachi" Technical University of Iaşi,
Department of Computer Science
and Engineering
e-mail: andreis@cs.tuiasi.ro

R E F E R E N C E S

1. Sukittanon S., Dame S.G., *Algorithm – Embedded State Machine Design for PSoC Using C Programming*. AN2329 Cypress, 2006.
2. *** *ATMega 16 8-bit Microcontroller with 16K Bytes in-System Programmable Flash Datasheet*. ATMEL Corporation, 2008.
3. *** *AVR IAR C/C++ Compiler, Reference Guide*. IAR Systems, 2005.
4. *** *AVR IAR Embedded Workbench IDE*. User Guide, IAR Systems, 2005.

MODELUL UNUI AUTOMAT UTILIZAT ÎN DEZVOLTAREA SOFTWARE PENTRU SISTEME EMBEDDED

(Rezumat)

Automatele reprezintă un model foarte utilizat în procesele de proiectare şi dezvoltare pentru programele care rulează pe sisteme embedded. Automatele permit o reprezentare abstractă a funcționării sistemelor, sunt uşor de înțeles de către oameni cu diverse pregătiri şi nu necesită cunoştințe anterioare despre limbaje de programare. Translatarea unei diagrame ce descrie funcționarea automat în cod sursă C este un proces simplu dacă se utilizează un model specific. În acest articol propunem un astfel de model (nucleu) care este implementat ca o bibliotecă de funcţii în limbajul de programare C. Limbajul C este un limbaj de programare matur şi foarte utilizat în dezvoltarea de aplicaţii pentru sistemele embedded. Acest fapt oferă portabilitate între diverse arhitecturi de calcul pentru biblioteca propusă. Biblioteca pune la dispoziţie abstracţii pentru toate conceptele necesare descrierii automatelor: stări, tranziţii, condiţii sau evenimente, acţiuni. Toate aceste concepte sunt implementate folosind structuri de date şi funcţii potrivit alese. Pentru a implementa un automat utilizând librăria propusă, utilizatorul trebuie să definească structurile de date şi funcţiile conform cerinţelor aplicaţiei sale. Funcţia principală a librăriei este FSM_Run şi operează astfel: verifică în mod secvenţial condiţiile pentru toate tranziţiile definite din starea curentă iar prima condiţie care este adevărată determină realizarea tranziţiei. Implementarea realizată a fost evaluată folosind următorul scenariu: s-a modelat un detector de secvenţă, s-a construit o soluţie pentru microcontrolerul STM32 de la STelectronics şi s-a măsurat numărul de cicli necesari execuţiei.