UML Hierarchical State Machine OOP Framework in C

Contents

Note: The code examples below are from a sample implementation of a hierarchical state machine. The entire sample state machine implementation is shown in the file **example\fsm_example.c**.

See **example-state-machine.pdf** in this folder for the state diagram.

See **UML state machine.pdf** in the **references** folder for a general discussion of UML Hierarchical State Machines.

1 Introduction

The framework defines FSM, State, and Event classes in C to implement UML Hierarchical State Machines. The framework is implemented in the files **fsm.c** and **fsm.h**. The header file exposes the framework API. It would be helpful to have both files open to refer to while reading this document.

The class attributes are standard C structs. Class methods are just C functions. Virtual functions are just function pointers in structs, typically initialized to default functions (or NULL for pure virtual functions).

```
Classes: Fsm, FsmState, FsmEvent
Fsm         - State Machine class
FsmState    - State class
FsmEvent    - Event class
```

Derive subclass structures by including the Base Class structure at the top of derived structures.

Override virtual base class functions by setting non-default function pointers in structs.

In the examples below, and the sample implementation, the objects are global and statically initialized. You can dynamically allocate them and/or initialize them at run time if you want.

1.1 Note on structure static initialization with designated initializers

The examples use the g++ form for structure designated initializers. So why are we using g++ and writing these objects in C? Long story for another day. Bottom line: Designated initializers are not part of the C++ standard, but g++ has an extension that allows them. The standard C99 way is:

```
{.field = value, ...}
```

g++ does it like this

```
{ field : value, ...}
```

Also, with g++, you have to list the fields in the order they are defined in the struct. (Sigh - at least it complains if the field order changes...) This extension also works with gcc, but the preferred way in gcc is to use the C99 technique.

Use the gcc/g++ extension technique if you have C files that will potentially be compiled with both gcc and g++.

If you are using a standard C compiler and don't care about C++, use the C99 technique for static initialization of structures.

You can always obfuscate things a little more and define an initializer macro that is ifdef'ed to detect C++ and/or g++. It's not a bad way to go if you're worried about portability. The framework defines the `DESIG_INIT` macro to do just that.

You don't have to use designated initializers in static initialization. That works in C and C++, but is kinda scary...

Note: You don't have to use static initialization at all. You can write constructors to do everything at run time if you want. In C the constructors kind of clutter things up, you have to explicitly call them anyway, and you can't use them for static initialization. So that's why we didn't include them. But if you want to use them, knock yourself out - they can be handy.

2 Overview

In this framework, State Machine (`Fsm`) objects only know what the current state is. There is no table of states to maintain. All the logic for triggering state transitions is coded in the event handlers.

A default "state handler" function (`FsmStateDefaultHandler`) takes care of finding event objects and calling event handlers for each event that states process. This default state handler also takes care of passing events to nested state machines.

In classic state machine implementations, you write a handler function for each state. The function usually handles each event id as a case in a switch statement, and either processes the event in place or calls an event handler.

In this framework, you do not need to write a state handler for each state. (You may, however, override the default state handler if you want.) You just initialize event objects which associate event ids with event handlers, and write the event handlers. The default state handler takes care of the rest.

Other than the event handler functions, there are no functions to write, unless you want to override the default state handler function.

You do not need to call the default state handler function directly unless you override it. In that case, you can (should!) call it from your override function.

2.1 How to use the State Machine classes

(See sections below for more details. Refer to **fsm.h** for base class definitions and functions.)

- Define a list of event IDs (an enum) for each FSM [Section 3]
- Define an `Fsm` object for each Finite State Machine (nested or not) [Section 0]
- Define prototypes for each event handler. There may be as many as one per event id per state. [Section 4 and 6]
- Define event objects (an event id and a handler function) for each event handled by each state. [Section 6]
- Create a list (an array) of event objects for each state. [Section 6]
- Define a state object for each state [Section 0]
- Write the event handler functions. [Section 4]

2.2 How to run the State Machine

After initializing the structures, call `FsmInit` for the top level state machine. This sets the initial state and sends an `EVT_FSM_ENTRY` event to the initial state. (Top level state machine event handlers must take care of initializing nested FSMs.)

Then for each event received, call `FsmRun` with the id of the event, and `pFsm` pointing to the top level state machine.

3 Event IDs

The framework defines a set of reserved event identifiers (see section 3.3 below). Normally you'll have additional event IDs for your state machines. Recommended practice is just to have a single list of event IDs for your whole system. This has two advantages:

1. each ID in the system is unique, and
2. you can get the whole list in a single enum for stronger type checking

This really helps when debugging (like when you send an event to the wrong thread or state machine).

You have several options for defining your event IDs. You can define a macro in the framework's user include file and the framework will create a single enum typedef with all the event identifiers. Or you can create your own event IDs outside of the user include file. The following sections describe these techniques.

3.1 Event Identifiers in the User Include file

3.1.1 FSM_USER_EVENTS Macro

Create a list of identifiers as a macro symbol named `FSM_USER_EVENTS` in the FSM user include file **fsm_events.h** . Each identifier in the list must be the argument to the framework's `FSM_EVENT_ID` macro. Use whitespace to separate the identifiers. For example:

```
#define FSM_USER_EVENTS         \
     FSM_EVENT_ID(EVT_1)        \
     FSM_EVENT_ID(EVT_2)        \
     FSM_EVENT_ID(EVT_3)        \
     FSM_EVENT_ID(EVT_4)
```

Compound macros are ok too. For example:

```
#define MY_EVENTS_A            \
     FSM_EVENT_ID(EVT_1)       \
     FSM_EVENT_ID(EVT_2)

#define MY_EVENTS_B            \
     FSM_EVENT_ID(EVT_3)       \
     FSM_EVENT_ID(EVT_4)

#define FSM_USER_EVENTS        \
     MY_EVENTS_A               \
     MY_EVENTS_B
```

Define as many events as you want. Substitute your event ID symbols for EVT_1, etc.

3.1.2 eFsmEvent enum typedef

The framework creates the `eFsmEvent` enum typedef which includes the reserved identifiers and the identifiers defined in the `FSM_USER_EVENTS` macro. The enum also defines some negative values used by the framework, but all the "real" event IDs are positive values, starting at 0.

Identifiers included in `FSM_USER_EVENTS` become enums which lie in the range

`0 < your_identifiers < EVT_FSM_EOL` .

Your identifier symbols will be enums of type `eFsmEvent`, auto incrementing in the order you define them. The first few positive enum values, starting at zero, are used by the framework, so the minimum value for your identifiers is not specified. Your identifiers, though, will be greater than 0. (Negative event identifier values are reserved by the framework.)

3.1.3 Identifier name array

The framework also creates an identifier name array with strings that have the same name as the event id symbols. Your code can get a pointer to the string identifier for each event id by using

`FSM_EVT_NAME(x)`

where x is the identifier symbol (e.g., EVT_1, etc.) The list includes the framework's reserved identifiers.

This also gives the framework access to your identifiers for debug messages.

3.2 Event Identifiers Outside The User Include File

You may also define event IDs outside the User Include File. Include **fsm.h** in your source file, then define your identifiers so that they are all >= `EVT_FSM_EOL`. You can have different identifier lists for each thread or state machine, you just can't use any values below `EVT_FSM_EOL`.

For example:

```
typedef enum {
     MY_EVT_1 = EVT_FSM_EOL,
     MY_EVT_2,                                  // etc...
} MyEventsA;

typedef enum {
     MY_EVT_3 = EVT_FSM_EOL,
     MY_EVT_4,                                  // etc...
} MyEventsB;
```

Use whatever symbols you want. In an enum, you can make the first one equal to `EVT_FSM_EOL`, then let the rest of them be defined sequentially. `EVT_FSM_EOL` is defined in the framework, and is one greater than the last event ID defined by the framework (including any events defined in `FSM_USER_EVENTS` - see section 3.1 above). You must not define any event IDs with values less than `EVT_FSM_EOL`.

You will probably need to cast event IDs defined this way to `eFsmEvent` types when you call framework APIs.

Your identifiers will all be positive numbers. You may not have system-wide unique event ID values depending on how you define them (e.g. in the example above, ID values in MyEventsA and MyEventsB are not unique).

The framework will not have access to event ID names for events not defined in `FSM_USER_EVENTS`. Identifiers not included in `FSM_USER_EVENTS` will **not** be in the framework's identifier name string array. Do **NOT** use `FSM_EVT_NAME(x)` with these identifiers - at best you will get garbage, at worst your system will crash inexplicably at random times.

3.3 Framework Reserved Event Identifiers

3.3.1 State Entry and Exit

The state entry/exit event IDs defined by the framework are

- `EVT_FSM_ENTRY` transition to state at current hierarchy level
- `EVT_FSM_EXIT` transition from state at current hierarchy level
- `EVT_FSM_SUPERSTATE_ENTRY` transition to superstate
- `EVT_FSM_SUPERSTATE_EXIT` transition from superstate

You must write handlers for these events if you want to perform the associated entry and/or exit actions.

`EVT_FSM_ENTRY` and `EVT_FSM_EXIT` indicate that a transition has occurred at the current hierarchy level.

`EVT_FSM_SUPERSTATE_ENTRY` and `EVT_FSM_SUPERSTATE_EXIT` indicate that a transition occurred in the superstate.

`EVT_FSM_ENTRY` and `EVT_FSM_SUPERSTATE_ENTRY` are sent when a *transition to* the state occurs.

`EVT_FSM_EXIT` and `EVT_FSM_SUPERSTATE_EXIT` are sent when a *transition away from* the state occurs.

You can implement UML Transition to History (H) and Deep History (H*) pseudo states by handling these events. (See the pdf files ***Pattern History.pdf*** by Miro Samek and ***UML-History.pdf*** by Bruce Powell Douglass in the **references** folder for a discussion of UML Transition to History).

`FsmInit` sends `EVT_FSM_ENTRY`.

3.3.2 Default Event

Sometimes you'd like to have a default event handler (as in a default switch case) for all events not otherwise explicitly handled. The framework defines `EVT_FSM_DEFAULT` as a default event ID. If this event is in a state's event list, the associated handler is called for any event not in the list. See section 6.1 below for more information on handling default events.

3.3.3 Null Event

`EVT_FSM_NULL` is the event ID in the fsmNullEvent event object (see section 6 below). This event ID is a negative number.

4 Event Handlers

You must write event handlers (functions) for all events handled in each state in your state machine. If a state doesn't handle an event, no handler for that state/event is required.

4.1 Event Handler Arguments

Event handlers have a specific prototype:

```
typedef FsmStatePtr (*FsmEvtHandler)(FsmState* pState, FsmEvent * pEvent);
```

You can use the `FSM_EVENT_HANDLER` macro to define your prototypes:

```
FSM_EVENT_HANDLER( MyNestedFsm1State1Evt_Entry );
```

Event handlers return a pointer to the next state (`FsmState*` or `FsmStatePtr`) when an event causes a transition. Event handlers return `NULL` when an event does not cause a transition.

If an event handler "consumes" an event (i.e., no further processing of the event is to be performed by a higher level state machine), the event handler sets

```
pEvent->consumed = true;
```

The 'consumed' field is false on entry to the event handler.

Note: If an event handler returns a pointer to the current state, it will cause exit actions and entry actions to be performed even though there is no transition. Most likely you want the event handler to return NULL when there is no transition.

Also Note: The handler function prototype for entry and exit events (`EVT_FSM_ENTRY`, `EVT_FSM_EXIT`, `EVT_FSM_SUPERSTATE_ENTRY`, and `EVT_FSM_SUPERSTATE_EXIT`) is the same as for other event handlers. However, exit event handlers should return `NULL`; i.e., *do not transition in an exit event handler*. The framework default state handler ignores `non-NULL` values returned by exit event handlers (otherwise, in this framework, that causes an infinite recursive loop).

4.2 Hierarchical Event Processing

The default state handler processes events in this order

- `EVT_FSM_ENTRY` passed to top-level state first, then `EVT_FSM_SUPERSTATE_ENTRY` passed to nested FSM. Lowest-level FSM event handler should set `pEvent->consumed = true` . This means entry events are processed in top-down order, which is similar to the order of class constructor execution in C++.
- When the handler sees `EVT_FSM_EXIT`, it sends `EVT_FSM_SUPERSTATE_EXIT` to the substate first, then sends `EVT_FSM_EXIT` to the current hierarchy state.
- All other events, including `EVT_FSM_EXIT`, are processed in bottom-up order; i.e., sub-states get the opportunity to override higher level states. If a sub-state event handler sets `pEvent->consumed = true` , the event is not passed to higher-level states' event handlers.

- EVT_FSM_EXIT/EVT_FSM_SUPERSTATE_EXIT event handlers should ***never*** set pEvent->consumed = true. This allows higher-level states to do exit processing after lower-level states, which is similar to destructor processing order in C++.

You can add default events to state event lists,

5 Define FSM Objects

Use the FSM macro to define State Machine objects:

```
FSM(fsm_Top, "Top", NULL, NULL, NULL );// instantiate the top level (superstate) FSM
FSM(fsm_Nested1, "Nested1", NULL, NULL, NULL );    // instantiate a nested FSM
FSM(fsm_Nested2, "Nested2", NULL, NULL, NULL );    // instantiate a nested FSM
```

Note there is no difference instantiating a nested state machine or a non-nested one. To nest an FSM, initialize the pNestedFsm field of an FsmState object to point to the nested FSM. (See details in Section 7.2 below.) The last two arguments passed to the FSM macro are for defer and recall queues. See section 8 below for more information.

6 Define Event objects and lists

You must define prototypes for your handlers, instantiate event objects, then put pointers to the event objects into event lists. Each event list defines a set of events that a state handles.

Use the FSM_EVENT_HANDLER macro to define event handlers for your events.

Use the FSM_EVENT macro to create event objects which associate event IDs with event handlers. The same id may have different handlers in different states - you need a separate event object for each (id, handler) pair.

There is no macro to create an event list. You just declare an array of type FsmEvent* and place pointers to event objects in the array.

The framework defines one event object: fsmNullEvent. You must terminate each event list with a pointer to the fsmNullEvent object so the default state handler can find the end of the list. It doesn't matter what order events are placed in the list.

The example below defines event handler prototypes, event objects, and an event list for a single state using the FSM_EVENT_HANDLER and FSM_EVENT macros:

```
//++++ Nested FSM 1 State 1 events ++++

// Method prototypes
FSM_EVENT_HANDLER( Nested1_State1_Entry );
FSM_EVENT_HANDLER( Nested1_State1_EVT3 );

// Objects
FSM_EVENT( evt_Nested1_State1_Entry, EVT_FSM_ENTRY, Nested1_State1_Entry );
FSM_EVENT( evt_Nested1_State1_EVT3,  EVT_3,         Nested1_State1_EVT3  );

// Event list array
FsmEvent* eventList_Nested1_State1[] = {
           &evt_Nested1_State1_Entry,
           &evt_Nested1_State1_EVT3,
           // keep this last
           &fsmNullEvent
};
```

Do similarly for the events handled by each state. The example uses a naming scheme to help identify the state which event handlers belong to. It's not required - this is just C, so you can name the functions anything you want. The important thing is to initialize event objects to associate event ids with handlers, then put event objects in a list for each state. You'll use the event list in the state object (`FsmState`) instantiation [see Section 7.1 below].

6.1 Default Events

Define a default event ID using the `EVT_FSM_DEFAULT` reserved identifier. For example:

```
FSM_EVENT( evt_Nested1_State1_Default, EVT_FSM_DEFAULT, Neste1_State1_Default);
```

Put this event in the event list for any state where you want a default event handler. The default handler will be called for any event not in the event list. For example:

```
// Event list array
FsmEvent* eventList_Nested1_State1[] = {
           &evt_Nested1_State1_Entry,
           &evt_Nested1_State1_EVT3,
           &evt_Nested1_State1_Default,
           // keep this last
           &fsmNullEvent
};
```

On entry to the default event handler, the `FsmEvent` structure `id` field is set to `EVT_FSM_DEFAULT`, and the `altId` field is set to the event ID originally passed to the state handler.

7 State Objects

7.1 Instantiating State Objects

Instantiate and initialize state objects (`FsmState`) for each state in the state machine using the `FSM_STATE` macro. The example below sets the parent state machine to the Nested1 FSM, uses the event list created in Section 6 above, has no nested state machine to pass events to, and uses an override state handler (`MyFsmStateHandler`).

```
FSM_STATE( state_Nested1_State1, &fsm_Nested1, NULL, eventList_Nested1_State1,
"State1", MyFsmStateHandler );
```

For the default state handler, use `FsmStateDefaultHandler` instead of `MyFsmStateHandler`.

You only need to initialize the data structure for each state. There is no state handler to write unless you override the default state handler.

7.2 Nested FSMs

To create a nested FSM hierarchy, create lists of FSMs. Then initialize each state object's nested FSM list pointer (`nestedFsmList`) to point to an FSM list.

The example below creates a top-level state with a nested state machine:

```
// Top state 1 Nested FSM list array
Fsm* nestedFsmList_Top_State1[] = {
        &fsm_Nested1,
        NULL
};

FSM_STATE( state_Top_State1, &fsm_Top, nestedFsmList_Top_State1, eventList_Top_State1,
"State1", MyFsmStateHandler );
```

This allows a single state to have multiple nested FSMs. This is particularly useful for implementing UML "orthogonal regions" (sometimes called "cooperative state machines") as nested component FSMs (see the pdf file ***Pattern Orthogonal.pdf*** by Miro Samek in the example folder for a discussion of this technique).

The default state handler dispatches events to all nested FSMs in the order of the nested FSM list. All event handlers for nested FSMs run in the same thread of execution (e.g., a single thread or task in a Real Time Operating System) as the state handler. For nested states to communicate or synchronize with each other, they must post events to a queue for the dispatcher to read. Event handlers must not call "dispatch" directly; otherwise, the framework's Run To Completion model will be broken.

7.3 Overriding the Default State Handler

To override the default state handler, initialize `FsmState` objects' state handler function pointer to point to a different function than `FsmStateDefaultHandler`.

A simple override state handler creates log entries for all events as they pass through the state machine. In this case, set the state handler argument in the `FSM_STATE` macro for all states to point to your override function, such as:

```
FSM_STATE( state_Nested1_State1, &fsm_Nested1, NULL, eventList_Nested1_State1,
"State1", MyFsmStateHandler );
```

Then implement `MyFsmStateHandler` like this:

```
bool MyFsmStateHandler(FsmState *pState, int eventId)
{
    bool consumed;

    FSM_ENTER_LOG("%s,%s,%s", pState->pFsm->name, pState->name,
                  FSM_EVT_NAME(eventId));

    consumed = FsmStateDefaultHandler(pState, eventId);

    FSM_EXIT_LOG("%s,%s,%s,%sconsumed", pState->pFsm->name, pState->name,
                  FSM_EVT_NAME(eventId), (consumed==0? "not_" : ""));

    return consumed;

} // MyFsmStateHandleEvent
```

7.4 Extending the State structure

If you need extended state variables to evaluate guard conditions in your event handlers (i.e., you have conditional transitions), you can derive your own State structure from the base class by including the base class structure as the first field in your derived structure. In the example below, the derived structure is named `MyFsmState`.

```
typedef struct {
     <some field definitions>
} MyExtraStuff;

typedef struct {
     FsmState     base;
     MyExtraStuff myStuff;
} MyFsmState;
```

Then create your state objects as `MyFsmState` objects. When your event handlers want to change states, they need to return pointers to `FsmState` objects, so they'll need to cast `MyFsmState` pointers to `FsmState` pointers for the return value (you wouldn't have to do the cast in C++, but, alas, this is C).

Likewise, to access the extended variables in the `MyFsmState` struct, event handlers need to cast the event handler `FsmStatePtr` argument to a `MyFsmState` pointer.

As long as the `FsmState` structure is the first element of the derived structure, the framework will handle everything correctly.

8 Queues

The framework implements simple circular queues for storing and retrieving event IDs. These queues are not thread-safe, and are intended to be used on a single thread of execution. You may instantiate different queues for different execution threads, you just can't use the same queue on multiple threads.

To instantiate a queue, use the FSM_Q macro, passing the name and size of the queue as arguments. For example:

```
FSM_Q(myQ,128)
```

creates an array of 128 `eFsmEvent` (i.e., event ID) elements . Call `FsmPutEvent` to put elements in the queue. Call `FsmGetEvent` to remove elements from the queue.

The framework's queues are also used to implement event deferral and recall. (See the pdf file ***Pattern_DeferredEvent.pdf*** by Miro Samek in the **references** folder for a discussion of deferred events.) The FSM macro takes two `FsmQ*` arguments: one is for a deferral queue, the other is a recall queue.

To use the framework's defer/recall mechanism, instantiate two queues using the FSM_Q macro. One is your deferral queue, the other is your recall queue. Pass the addresses of the queues to the FSM macro when instantiating the top level state machine in the hierarchy.

To defer an event, call `FsmDeferEvent`, with the `Fsm*` argument pointing to the top level state machine.

When you are ready to handle deferred events, call `FsmRecallEvent`. with the `Fsm*` argument pointing to the top level state machine. The framework moves the next event from the deferral queue to the recall queue. After processing the current event, in the `FsmRun` function the framework dispatches all events on the recall queue to the state machine.

9 Trace Log Output

Macros `FSM_ENTER_LOG`, `FSM_EXIT_LOG`, and `FSM_RUN_LOG` are defined for tracing. This is a very handy thing to do. We implemented tracing outside the framework to allow tracing in some state machines and not others, rather than having traces from all state machines jumbled together.

The tracing macros, when used as shown in **fsm_example.c**, create log output that can be pasted into a .csv text file, and opened with a spreadsheet program. If using Microsoft Excel, open the .csv file and paste the contents into **fsm-trace.xlsx**, beginning at cell A2. This spreadsheet has conditional formatting to make reading the trace output easier.

To use the macros:

- Override the default state handler
- In your state handler, include the `FSM_ENTER_LOG` / `FSM_EXIT_LOG` macros, as shown in Section 0
- Use `FSM_RUN_LOG` as you like, typically in event handlers.
- Define `#define FSM_TRACE 1` in your .c file before including **fsm.h**

10 Example State Machine

Refer to the diagram in **example-state-machine.pdf**.

The example state machine has a top-level state machine and two nested state machines.

When the machine transitions to Top State 1, Nested FSM1 always resumes from the state it was in when the machine transitioned away from Top State 1. In this state machine, we want to resume without going through entry and exit events, so entry/exit events from the superstate are ignored.

When the machine transitions to Top State 2, Nested FSM 2 always resumes from Nested State 1. When the machine transitions away from Top State 2, Nested FSM 2 receives exit events for whatever state it is in. The entry events for Nested State 1 occur when the machine transitions back to Top State 2.

The event handlers in the example FSM do the housekeeping required by the framework; i.e. :

- return a pointer to the next state for a transition, or NULL for no transition
- set the `pState->consumed` field if necessary
- setup reentry for substates in entry event handlers

Normally your state machine will have more to do in the event handlers.

11 Testing

The framework includes features used for testing which are enabled by:

```
#define FSM_TEST 1
```

at the top of **fsm.h**.

Test features include

- Insert events before and/or after other events
- Notify when a state is passed a specified event (not restricted to the events handled by the state)

These are very handy for testing complex state machines, especially when state machines execute in different threads and pass events between themselves. Using these features, you can simulate different event ordering in timing windows that might be hard to test in a lab.

11.1 Insert Events

The framework instantiates two arrays of queues (see Section 8 above), indexed by the FSM reserved event IDs and the event IDs declared in `FSM_USER_EVENTS` . For each event, there is an "insert before" queue and an "insert after" queue. `FsmRun` uses the event ID currently being processed as an index into the "before" and "after" arrays. Any events found in the "before" array will be processed before the current event. Any events found in the "after" array will be processed after the current event.

Use `FsmInsertBefore` and `FsmInsertAfter` to insert events into the before and after queues.

This is especially useful for testing state machine responses to corner cases and race conditions between events. For example, if EVT_X and EVT_Z are user inputs to a state machine, and EVT_Y is a response from the state machine:

```
if (0 < FsmInsertBefore(EVT_Y, EVT_Z))
     printf("insert failed\n");
FsmRun(EVT_X);
```

This inserts EVT_Z immediately before EVT_Y. If EVT_Y is the normal response to EVT_X from some state deep in the state machine hierarchy, this allows you to insert EVT_Z in front of EVT_Y when EVT_Y would normally be processed by the top level state machine. This simulates a timing window where EVT_Z enters the event queue immediately before EVT_Y.

11.2 Notify Events

Sometimes you need to know when a state "sees" a particular event so you can send another event at the right time, or have your test program wait before changing some global test variable. For example, a state machine may enter a state which processes interrupt events, but which doesn't generate any events in that state. You could be "notified" when the state machine enters the interrupt event processing state, and then have your test program send events to simulate interrupts.

The framework calls a function that you specify whenever the default state handler FsmStateDefaultHandler sees a "notify event" specified for the current state. Use the FSM_SET_NOTIFY_FCN and FSM_CLR_NOTIFY_FCN macros to set and clear the notify function. Use the FSM_SET_NOTIFY_EVENT and FSM_CLR_NOTIFY_EVENT macros to set and clear the notify event. For example:

```
FSM_SET_NOTIFY_FCN(TestNotifyEvent);
FSM_SET_NOTIFY_EVENT((&myState),EVT_FSM_ENTRY);

// ... send some events to the state machine..., then

// Wait for state machine to enter myState
TestWaitEvent();

// do some stuff, send more events, etc., then

FSM_CLR_NOTIFY_EVENT((&myState));
FSM_CLR_NOTIFY_FCN();
```

In this example, myState is the state you are interested in, and EVT_FSM_ENTRY (the entry event) is the event for which you want notification. In this case, the framework calls your notification function when it processes the entry event. You can specify any event you want - not just the entry event.

You write the notify function called by the framework. In the example above, the notify function name is TestNotifyEvent. Typically this is an OS-dependent function that uses a synchronization mechanism to wake up a sleeping thread. In the example above, TestWaitEvent puts the test thread to sleep waiting for notification from the framework that myState has received an entry event (EVT_FSM_ENTRY). Implementation of these functions on WIN32 is shown below:

```
HANDLE hTestNotify;

void SomeOtherFunction(void)
{
     ...
     hTestNotify = CreateEvent(NULL,FALSE,FALSE, NULL);
}


void TestNotifyEvent(void)
{
     SetEvent(hTestNotify);
}

void TestWaitEvent(void)
{
     WaitForSingleObject(hTestNotify, INFINITE);
}
```

12 Notation Guide