State Machine Design in C++

David Lafreniere, 18 Apr 2016 CPOL

A compact C++ finite state machine (FSM) implementation that's easy to use on embedded and PC-based systems.

- **Download StateMachine.zip - 20.9 KB**
- **Download StateMachineCompact.zip - 4.2 KB**

Introduction

In 2000, I wrote an article entitled "*State Machine Design in C++*" for C/C++ Users Journal (R.I.P.). Interestingly, that old article is still available and (at the time of writing this article) the #1 hit on Google when searching for C++ state machine. The article was written over 15 years ago, but I continue to use the basic idea on numerous projects. It's compact, easy to understand and, in most cases, has just enough features to accomplish what I need.

This article provides a new implementation with updated features at the slight expense of a bit more code. I'll also correct the errors in the original implementation because if you're really tight on storage it's still be a viable solution. Both designs are table driven suitable for any platform, embedded or PC, with any C++ compiler.

Why another state machine design? Certainly by now there's an existing implementation out there that can be used, right? Maybe. On occasion, I'll try a new state machine and find it doesn't fit my needs for one reason or another. For me, the problem usually boils down to one or more of the following:

1. **Too large** – the resulting implementation takes too much code space to justify on an embedded platform.
2. **Too complex** – excessive templates or requires adoption of a complete framework to use the state machine.
3. **No compiler support** – relies upon new C++ language features not supported by the compiler.
4. **High learning curve** – some require quite a lot of effort in this regard.
5. **Difficult syntax** – non-intuitive state machine expression with hard to diagnose compiler errors.
6. **External libraries** – relies upon external libraries that add size or aren't supported on the platform.
7. **Too many features** – full UML compliance isn't required and therefore exceeds the basic needs of the problem at hand.
8. **No event data** – can't send unique event data to a state function.
9. **Central event handler** – a single event handling function and a switch statement handles events for each class.
10. **No type safety** – requires manual typecasting the event data based on an enumeration.
11. **Limited transition rules** – no support for event ignored and can't happen event transitions.
12. **No thread-safety** – the code is not thread-safe.

Don't get me wrong, some implementations are quite impressive and suitable for many different projects. Every design has certain tradeoffs and this one is no different. Only you can decide if this one meets your needs or not. I'll try to get you bootstrapped as quickly as possible through this article and sample code. This state machine has the following features:

1. **Compact** – the StateMachine class is not a template – only 448 bytes of code on Windows. Templates are used sparingly.
2. **Transition tables** – transition tables precisely control state transition behavior.
3. **Events** – every event is a simple public instance member function with any argument types.
4. **State action** – every state action is a separate instance member function with a single, unique event data argument if desired.
5. **Guards/entry/exit actions** – optionally a state machine can use guard conditions and separate entry/exit action functions for each state.
6. **State machine inheritance** – supports inheriting states from a base state machine class.
7. **State function inheritance** – supports overriding a state function within a derived class.
8. **Macros** – optional multiline macro support simplifies usage by automating the code "machinery".
9. **Type safe** – compile time checks catch mistakes early. Runtime checks for the other cases.
10. **Thread-safe** – adding software locks to make the code thread-safe is easy.

This state machine design is not trying to achieve a full UML feature set. It is also not a Hierarchical State Machine (HSM). Instead, its goal is to be relatively compact, portable, and easy to use traditional Finite State Machine (FSM) with just enough unique features to solve many different problems.

The article is not a tutorial on the best design decomposition practices for software state machines. I'll be focusing on state machine code and simple examples with just enough complexity to facilitate understanding the features and usage.

Background

A common design technique in the repertoire of most programmers is the venerable finite state machine (FSM). Designers use this programming construct to break complex problems into manageable states and state transitions. There are innumerable ways to implement a state machine.

A switch statement provides one of the easiest to implement and most common version of a state machine. Here, each case within the switch statement becomes a state, implemented something like:

Hide Copy Code

```
switch (currentState) {
   case ST_IDLE:
      // do something in the idle state
      break;
    case ST_STOP:
      // do something in the stop state
      break;
   // etc...
}
```

This method is certainly appropriate for solving many different design problems. When employed on an event driven, multithreaded project, however, state machines of this form can be quite limiting.

The first problem revolves around controlling what state transitions are valid and which ones are invalid. There is no way to enforce the state transition rules. Any transition is allowed at any time, which is not particularly desirable. For most designs, only a few transition patterns are valid. Ideally, the software design should enforce these predefined state sequences and prevent the unwanted transitions. Another problem arises when trying to send data to a specific state. Since the entire state machine is located within a single function, sending additional data to any given state proves difficult. And lastly these designs are rarely suitable for use in a multithreaded system. The designer must ensure the state machine is called from a single thread of control.

Why use a state machine?

Implementing code using a state machine is an extremely handy design technique for solving complex engineering problems. State machines break down the design into a series of steps, or what are called states in state-machine lingo. Each state performs some narrowly defined task. Events, on the other hand, are the stimuli, which cause the state machine to move, or transition, between states.

To take a simple example, which I will use throughout this article, let's say we are designing motor-control software. We want to start and stop the motor, as well as change the motor's speed. Simple enough. The motor control events to be exposed to the client software will be as follows:

1. **Set Speed** – sets the motor going at a specific speed.
2. **Halt** – stops the motor.

These events provide the ability to start the motor at whatever speed desired, which also implies changing the speed of an already moving motor. Or we can stop the motor altogether. To the motor-control class, these two events, or functions, are considered external events. To a client using our code, however, these are just plain functions within a class.

These events are not state machine states. The steps required to handle these two events are different. In this case the states are:

1. **Idle** — the motor is not spinning but is at rest.
- Do nothing.
2. **Start** — starts the motor from a dead stop.
- Turn on motor power.
- Set motor speed.
3. **Change Speed** — adjust the speed of an already moving motor.
- Change motor speed.
4. **Stop** — stop a moving motor.
- Turn off motor power.
- Go to the Idle state.

As can be seen, breaking the motor control into discreet states, as opposed to having one monolithic function, we can more easily manage the rules of how to operate the motor.

Every state machine has the concept of a "current state." This is the state the state machine currently occupies. At any given moment in time, the state machine can be in only a single state. Every instance of a particular state machine class can set the initial state during construction. That initial state, however, does not execute during object creation. Only an event sent to the state machine causes a state function to execute.

To graphically illustrate the states and events, we use a state diagram. Figure 1 below shows the state transitions for the motor control class. A box denotes a state and a connecting arrow indicates the event transitions. Arrows with the event name listed are external events, whereas unadorned lines are considered internal events. (I cover the differences between internal and external events later in the article.)

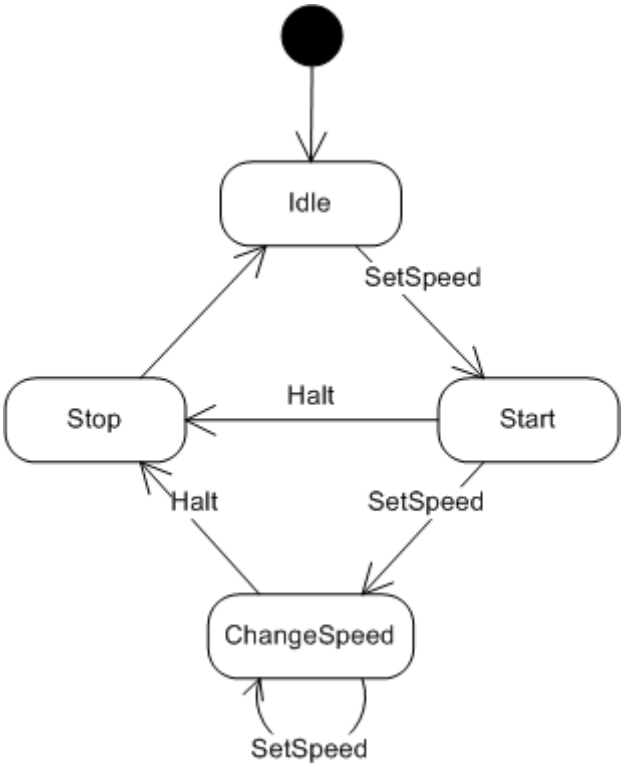

Figure 1: Motor state diagram

As you can see, when an event comes in the state transition that occurs depends on state machine's current state. When a SetSpeed event comes in, for instance, and the motor is in the Idle state, it transitions to the Start state. However, that same SetSpeed event generated while the current state is Start transitions the motor to the ChangeSpeed state. You can also see that not all state transitions are valid. For instance, the motor can't transition from ChangeSpeed to Idle without first going through the Stop state.

In short, using a state machine captures and enforces complex interactions, which might otherwise be difficult to convey and implement.

Internal and external events

As I mentioned earlier, an event is the stimulus that causes a state machine to transition between states. For instance, a button press could be an event. Events can be broken out into two categories: external and internal. The external event, at its most basic level, is a function call into a state-machine object. These functions are public and are called from the outside or from code external to the state-machine object. Any thread or task within a system can generate an external event. If the external event function call causes a state transition to occur, the state will execute synchronously within the caller's thread of control. An internal event, on the other hand, is self-generated by the state machine itself during state execution.

A typical scenario consists of an external event being generated, which, again, boils down to a function call into the class's public interface. Based upon the event being generated and the state machine's current state, a lookup is performed to determine if a transition is required. If so, the state machine transitions to the new state and the code for that state executes. At the end of the state function, a check is performed to determine whether an internal event was generated. If so, another transition is performed and the new state gets a chance to execute. This process continues until the state machine is no longer generating internal events, at which time the original external event function call returns. The external event and all internal events, if any, execute within the caller's thread of control.

Once the external event starts the state machine executing, it cannot be interrupted by another external event until the external event and all internal events have completed execution if locks are used. This run to completion model provides a multithread-safe environment for the state transitions. Semaphores or mutexes can be used in the state machine engine to block other threads that might be trying to be simultaneously access the same object. See source code function `ExternalEvent()` comments for where the locks go.

Event data

When an event is generated, it can optionally attach event data to be used by the state function during execution. Once the state has completed execution, the event data is considered used up and must be deleted. Therefore, any event data sent to a state machine must be created on the heap, via operator new, so that the state machine can delete it once used. In addition, for our particular implementation the event data must inherit from the `EventData` base class. This gives the state machine engine a common base class for which to delete all event data.

Hide Copy Code

```
class EventData
{
public:
    virtual ~EventData() {}
};
```

State transitions

When an external event is generated, a lookup is performed to determine the state transition course of action. There are three possible outcomes to an event: new state, event ignored, or cannot happen. A new state causes a transition to a new state where it is allowed to execute. Transitions to the existing state are also possible, which means the current state is re-executed. For an ignored event, no state executes. However, the event data, if any, is deleted. The last possibility, cannot happen, is reserved for situations where the event is not valid given the current state of the state machine. If this occurs, the software faults.

In this implementation, internal events are not required to perform a validating transition lookup. The state transition is assumed to be valid. You could check for both valid internal and external event transitions, but in practice, this just takes more storage space and generates busywork for very little benefit. The real need for validating transitions lies in the asynchronous, external events where a client can cause an event to occur at an inappropriate time. Once the state machine is executing, it cannot be interrupted. It is under the control of the class's private implementation, thereby making transition checks unnecessary. This gives the designer the freedom to change states, via internal events, without the burden of updating transition tables.

StateMachine class

Two base classes are necessary when creating your own state machine: `StateMachine` and `EventData`. A class inherits from `StateMachine` to obtain the necessary mechanisms to support state transitions and event handling. The `StateMachine` header also contains various preprocessor multiline macros to ease implementation of the state machine (explained later in the article). To send unique data to the state functions, the structure must inherit from the `EventData` base class.

The state machine source code is contained within the StateMachine.cpp and StateMachine.h files (see attached*StateMachine.zip*). The code below shows the class declaration.


```
class StateMachine
{
public:
    enum { EVENT_IGNORED = 0xFE, CANNOT_HAPPEN };

    StateMachine(BYTE maxStates, BYTE initialState = 0);
    virtual ~StateMachine() {}

    BYTE GetCurrentState() { return m_currentState; }

protected:
    void ExternalEvent(BYTE newState, const EventData* pData = NULL);
    void InternalEvent(BYTE newState, const EventData* pData = NULL);

private:
    const BYTE MAX_STATES;
    BYTE m_currentState;
    BYTE m_newState;
```

```
    BOOL m_eventGenerated;
    const EventData* m_pEventData;

    virtual const StateMapRow* GetStateMap() = 0;
    virtual const StateMapRowEx* GetStateMapEx() = 0;

    void SetCurrentState(BYTE newState) { m_currentState = newState; }

    void StateEngine(void);
    void StateEngine(const StateMapRow* const pStateMap);
    void StateEngine(const StateMapRowEx* const pStateMapEx);
};
```

`StateMachine` is the base class used for handling events and state transitions. The interface is contained within four functions:

Hide Copy Code

```
void ExternalEvent(BYTE newState, const EventData* pData = NULL);
void InternalEvent(BYTE newState, const EventData* pData = NULL);
virtual const StateMapRow* GetStateMap() = 0;
virtual const StateMapRowEx* GetStateMapEx() = 0;
```

`ExternalEvent()` generates an external event to the state machine using as arguments the new state and a pointer to an EventData object, if any. The `InternalEvent()` function generates internal events using the same set of arguments.

The `GetStateMap()` and `GetStateMapEx()` functions return an array of `StateMapRow` or `StateMapRowEx`instances which will be retrieved by the state engine when appropriate. The inheriting class must return an array with one of these functions. If the state machine only has state functions, `GetStateMap()` is used. If guard/entry/exit features are required, the `GetStateMapEx()` is used. The other unused version must return`NULL`. However, multiline macros are provided to implement these functions for us, as I will demonstrate shortly.

Motor example

`Motor` and `MotorNM` classes are examples of how to use `StateMachine`. `MotorNM` (No Macros) exactly matches the `Motor` design without relying upon macros. This allows viewing all the macro-expanded code for ease of understanding. However, once up to speed I find that the macros greatly simplify usage by hiding the required source machinery.

The `MotorNM` class declaration shown below contains no macros:

Hide Shrink Copy Code

```
class MotorNMData : public EventData
{
public:
```

```
    INT speed;
};

// Motor class with no macros
class MotorNM : public StateMachine
{
public:
    MotorNM();

    // External events taken by this state machine
    void SetSpeed(MotorNMData* data);
    void Halt();

private:
    INT m_currentSpeed;

    // State enumeration order must match the order of state method entries
    // in the state map.
    enum States
    {
        ST_IDLE,
        ST_STOP,
        ST_START,
        ST_CHANGE_SPEED,
        ST_MAX_STATES
    };

    // Define the state machine state functions with event data type
    void ST_Idle(const NoEventData*);
    StateAction<MotorNM, NoEventData, &MotorNM::ST_Idle> Idle;

    void ST_Stop(const NoEventData*);
    StateAction<MotorNM, NoEventData, &MotorNM::ST_Stop> Stop;

    void ST_Start(const MotorNMData*);
    StateAction<MotorNM, MotorNMData, &MotorNM::ST_Start> Start;

    void ST_ChangeSpeed(const MotorNMData*);
    StateAction<MotorNM, MotorNMData, &MotorNM::ST_ChangeSpeed> ChangeSpeed;

    // State map to define state object order. Each state map entry defines a
    // state object.
private:
    virtual const StateMapRowEx* GetStateMapEx() { return NULL; }
    virtual const StateMapRow* GetStateMap() {
        static const StateMapRow STATE_MAP[] = {
            &Idle,
            &Stop,
            &Start,
```

```
            &ChangeSpeed,
        };
        C_ASSERT((sizeof(STATE_MAP)/sizeof(StateMapRow)) == ST_MAX_STATES);
        return &STATE_MAP[0]; }
};
```

The `Motor` class uses macros for comparison:

Hide Shrink Copy Code

```
class MotorData : public EventData
{
public:
    INT speed;
};

// Motor class using macro support
class Motor : public StateMachine
{
public:
    Motor();

    // External events taken by this state machine
    void SetSpeed(MotorData* data);
    void Halt();

private:
    INT m_currentSpeed;

    // State enumeration order must match the order of state method entries
    // in the state map.
    enum States
    {
        ST_IDLE,
        ST_STOP,
        ST_START,
        ST_CHANGE_SPEED,
        ST_MAX_STATES
    };

    // Define the state machine state functions with event data type
    STATE_DECLARE(Motor,     Idle,           NoEventData)
    STATE_DECLARE(Motor,     Stop,           NoEventData)
    STATE_DECLARE(Motor,     Start,          MotorData)
    STATE_DECLARE(Motor,     ChangeSpeed,     MotorData)

    // State map to define state object order. Each state map entry defines a
    // state object.
    BEGIN_STATE_MAP
```

```
        STATE_MAP_ENTRY(&Idle)
        STATE_MAP_ENTRY(&Stop)
        STATE_MAP_ENTRY(&Start)
        STATE_MAP_ENTRY(&ChangeSpeed)
    END_STATE_MAP
};
```

`Motor` implements our hypothetical motor-control state machine, where clients can start the motor, at a specific speed, and stop the motor. The`SetSpeed()` and `Halt()` public functions are considered external events into the `Motor` state machine. `SetSpeed()` takes event data, which contains the motor speed. This data structure will be deleted upon completion of the state processing, so it is imperative that the structure inherit from`EventData` and be created using `operator new` before the function call is made.

When the `Motor` class is created, its initial state is Idle. The first call to `SetSpeed()` transitions the state machine to the Start state, where the motor is initially set into motion. Subsequent `SetSpeed()` events transition to the ChangeSpeed state, where the speed of an already moving motor is adjusted.
The `Halt()` event transitions to Stop, where, during state execution, an internal event is generated to transition back to the Idle state.

Creating a new state machine requires a few basic high-level steps:

1. Inherit from the `StateMachine` base class.
2. Create a `States` enumeration with one entry per state function.
3. Create state functions using the `STATE` macros.
4. Optionally create guard/entry/exit functions for each state using the `GUARD`, `ENTRY` and `EXIT` macros.
5. Create one state map lookup table using the `STATE_MAP` macros.
6. Create one transition map lookup table for each external event using the `TRANSITION_MAP` macros.

State functions

State functions implement each state — one state function per state-machine state. In this implementation, all state functions must adhere this state-function signature, which is as follows:


```
void <class>::<func>(const EventData*)
```

<class> and <func> are not template parameters but just placeholders for the particular class and function name respectively. For example, you might choose a signature such as`void Motor::ST_Start(const MotorData*)` . The important thing here is that the function returns no data (has a `void` return type) and that it has one input argument of type `EventData*` (or a derived class thereof).

Declare state functions with the `STATE_DECLARE` macro. The macro arguments are the state machine class name, state function name and event data type.

Hide Copy Code

```
STATE_DECLARE(Motor,    Idle,          NoEventData)
STATE_DECLARE(Motor,    Stop,          NoEventData)
STATE_DECLARE(Motor,    Start,         MotorData)
STATE_DECLARE(Motor,    ChangeSpeed,    MotorData)
```

Expanding the macros above yields:

Hide Copy Code

```
void ST_Idle(const NoEventData*);
StateAction<Motor, NoEventData, &Motor::ST_Idle> Idle;

void ST_Stop(const NoEventData*);
StateAction<Motor, NoEventData, &Motor::ST_Stop> Stop;

void ST_Start(const MotorData*);
StateAction<MotorNM, MotorData, &Motor::ST_Start> Start;

void ST_ChangeSpeed(const MotorData*);
StateAction<Motor, MotorData, &Motor::ST_ChangeSpeed> ChangeSpeed;
```

Notice the multiline macros prepend "ST_" to each state function name. Three characters are added to each state/guard/entry/exit function automatically within the macro. For instance, if declaring a function using`STATE_DEFINE(Motor, Idle, NoEventData)` the actual state function is called `ST_Idle()`.

1. ST_ - state function prepend characters
2. GD_ - guard function prepend characters
3. EN_ - entry function prepend characters
4. EX_ - exit function prepend characters

After a state function is declared, define a state function implementation with the `STATE_DEFINE` macro. The arguments are the state machine class name, state function name, and event data type. The code to implement your state behavior goes inside the state function. Note, any state function code may call`InternalEvent()` to switch to another state. Guard/entry/exit functions cannot call `InternalEvent()` otherwise a runtime error will result.

Hide Copy Code

```
STATE_DEFINE(Motor, Stop, NoEventData)
{
    cout << "Motor::ST_Stop" << endl;
    m_currentSpeed = 0;

    // perform the stop motor processing here
    // transition to Idle via an internal event
    InternalEvent(ST_IDLE);
}
```

Expanding the macro yields this state function definition.

Hide Copy Code

```
void Motor::ST_Stop(const NoEventData* data)
{
    cout << "Motor::ST_Stop" << endl;
    m_currentSpeed = 0;

    // perform the stop motor processing here
    // transition to Idle via an internal event
    InternalEvent(ST_IDLE);
}
```

Each state function must have an enumeration associated with it. These enumerations are used to store the current state of the state machine. In `Motor`, `States` provides these enumerations, which are used later for indexing into the transition map and state map lookup tables.

Hide Copy Code

```
enum States
{
    ST_IDLE,
    ST_STOP,
    ST_START,
    ST_CHANGE_SPEED,
    ST_MAX_STATES
};
```

It is important that the enumeration order match the order provided within the state map. This way, a state enumeration is tied to a particular state function call. `EVENT_IGNORED` and `CANNOT_HAPPEN` are two other constants used in conjunction with these state enumerations. `EVENT_IGNORED` tells the state engine not to execute any state, just return and do nothing. `CANNOT_HAPPEN` tells the state engine to fault. This abnormal catastrophic failure condition is never supposed to occur.

State map

The state-machine engine knows which state function to call by using the state map. The state map maps the`m_currentState` variable to a specific state function. For instance, if `m_currentState` is 2, then the third state-map function pointer entry will be called (counting from zero). The state map table is created using these three macros:

Hide Copy Code

```
BEGIN_STATE_MAP
STATE_MAP_ENTRY
END_STATE_MAP
```

`BEGIN_STATE_MAP` starts the state map sequence. Each `STATE_MAP_ENTRY` has a state function name argument. `END_STATE_MAP` terminates the map. The state map for `Motor` is shown below.

Hide Copy Code

```
BEGIN_STATE_MAP
    STATE_MAP_ENTRY(&Idle)
    STATE_MAP_ENTRY(&Stop)
    STATE_MAP_ENTRY(&Start)
    STATE_MAP_ENTRY(&ChangeSpeed)
END_STATE_MAP
```

The completed state map just implements the pure virtual function `GetStateMap()` defined within the`StateMachine` base class. Now the `StateMachine` base class can get all the `StateMapRow` objects via this call. The macro-expanded code is shown below:

Hide Copy Code

```
private:
    virtual const StateMapRowEx* GetStateMapEx() { return NULL; }
    virtual const StateMapRow* GetStateMap() {
        static const StateMapRow STATE_MAP[] = {
            &Idle,
            &Stop,
            &Start,
            &ChangeSpeed,
        };
        C_ASSERT((sizeof(STATE_MAP)/sizeof(StateMapRow)) == ST_MAX_STATES);
        return &STATE_MAP[0]; }
```

Notice the `C_ASSERT` macro. It provides compile time protection against a state map having the wrong number of entries. Visual Studio gives an error of "error C2118: negative subscript". Your compiler may give a different error message.

Alternatively, guard/entry/exit features require utilizing the _EX (extended) version of the macros.

Hide Copy Code

```
BEGIN_STATE_MAP_EX
STATE_MAP_ENTRY_EX or STATE_MAP_ENTRY_ALL_EX
END_STATE_MAP_EX
```

The `STATE_MAP_ENTRY_ALL_EX` macro has four arguments for the state action, guard condition, entry action and exit action in that order. The state action is mandatory but the other actions are optional. If a state doesn't have an action, then use 0 for the argument. If a state doesn't have any guard/entry/exit options, the`STATE_MAP_ENTRY_EX` macro defaults all unused options to 0. The macro snippet below is for an advanced example presented later in the article.

Hide Copy Code

```
BEGIN_STATE_MAP_EX
```

```
    STATE_MAP_ENTRY_ALL_EX(&Idle, 0, &EntryIdle, 0)
    STATE_MAP_ENTRY_EX(&Completed)
    STATE_MAP_ENTRY_EX(&Failed)
    STATE_MAP_ENTRY_ALL_EX(&StartTest, &GuardStartTest, 0, 0)
    STATE_MAP_ENTRY_EX(&Acceleration)
    STATE_MAP_ENTRY_ALL_EX(&WaitForAcceleration, 0, 0, &ExitWaitForAcceleration)
    STATE_MAP_ENTRY_EX(&Deceleration)
    STATE_MAP_ENTRY_ALL_EX(&WaitForDeceleration, 0, 0, &ExitWaitForDeceleration)
END_STATE_MAP_EX
```

Each entry within the transition map is a `StateMapRow`:

Hide Copy Code

```
struct StateMapRow
{
    const StateBase* const State;
};
```

The `StateBase` pointer has a pure virtual interface called by `StateEngine()`.

Hide Copy Code

```
class StateBase
{
public:
    virtual void InvokeStateAction(StateMachine* sm, const EventData* data) const = 0;
};
```

The `StateAction` derives from `StateBase` and its sole responsibility is to implement `InvokeStateAction()`and cast the `StateMachine` and `EventData` pointers to the correct derived class types, then call the state member function. Therefore, the state engine overhead to call each state function is one virtual function call, one`static_cast<>` and one `dynamic_cast<>`.

Hide Copy Code

```
template <class SM, class Data, void (SM::*Func)(const Data*)>
class StateAction : public StateBase
{
public:
    virtual void InvokeStateAction(StateMachine* sm, const EventData* data) const
    {
        // Downcast the state machine and event data to the correct derived type
        SM* derivedSM = static_cast<SM*>(sm);

        // Dynamic cast the data to the correct derived type
        const Data* derivedData = dynamic_cast<const Data*>(data);
        ASSERT_TRUE(derivedData != NULL);

        // Call the state function
```

```
        (derivedSM->*Func)(derivedData);
    }
};
```

The template arguments to `StateAction<>` are a state machine class (`SM`), an event data type (`Data`) and a member function pointer to the state function (`Func`).

`GuardCondition<>`, `EntryAction<>` and`ExitAction<>` classes also exist and their role is the same – typecast state machine and event data then call the action member function. Minor variations exist with the template arguments. The `GuardCondition<>` class `Func` template parameter changes slightly and returns a`BOOL`. `ExitAction<>` doesn't have a `Data` template argument.

Transition map

The last detail to attend to are the state transition rules. How does the state machine know what transitions should occur? The answer is the transition map. A transition map is lookup table that maps the `m_currentState`variable to a state enum constant. Every external event has a transition map table created with three macros:

Hide Copy Code

```
BEGIN_TRANSITION_MAP
TRANSITION_MAP_ENTRY
END_TRANSITION_MAP
```

The `Halt()` event function in `Motor` defines the transition map as:

Hide Copy Code

```
void Motor::Halt()
{
    BEGIN_TRANSITION_MAP                                    // - Current State -
        TRANSITION_MAP_ENTRY (EVENT_IGNORED)                 // ST_IDLE
        TRANSITION_MAP_ENTRY (CANNOT_HAPPEN)                 // ST_STOP
        TRANSITION_MAP_ENTRY (ST_STOP)                       // ST_START
        TRANSITION_MAP_ENTRY (ST_STOP)                       // ST_CHANGE_SPEED
    END_TRANSITION_MAP(NULL)
}
```

The macro-expanded code for `Halt()` is below. Again, notice the `C_ASSERT` macro providing compile time protection against an incorrect number of transition map entries.

Hide Copy Code

```
void Motor::Halt()
{
    static const BYTE TRANSITIONS[] = {
        EVENT_IGNORED,                  // ST_IDLE
```

```
        CANNOT_HAPPEN,                  // ST_STOP
        ST_STOP,                        // ST_START
        ST_STOP,                        // ST_CHANGE_SPEED
    };
    ExternalEvent(TRANSITIONS[GetCurrentState()], NULL);
    C_ASSERT((sizeof(TRANSITIONS)/sizeof(BYTE)) == ST_MAX_STATES);
}
```

`BEGIN_TRANSITION_MAP` starts the map. Each `TRANSITION_MAP_ENTRY` that follows indicates what the state machine should do based upon the current state. The number of entries in each transition map table must match the number of state functions exactly. In our example, we have four state functions, so we need four entries. The location of each entry matches the order of state functions defined within the state map. Thus, the first entry within the `Halt()` function indicates an `EVENT_IGNORED` as shown below.

Hide Copy Code

```
TRANSITION_MAP_ENTRY (EVENT_IGNORED)    // ST_IDLE
```

This is interpreted as "If a Halt event occurs while the current state is state Idle, just ignore the event."

Similarly, the third entry in the map is:

Hide Copy Code

```
TRANSITION_MAP_ENTRY (ST_STOP)        // ST_START
```

This indicates "If a Halt event occurs while current is state Start, then transition to state Stop."

`END_TRANSITION_MAP` terminates the map. The argument to this end macro is the event data, if any. `Halt()`has no event data so the argument is `NULL`, but`ChangeSpeed()` has data so it is passed in here.

State engine

The state engine executes the state functions based upon events generated. The transition map is an array of`StateMapRow` instances indexed by the `m_currentState` variable. When the `StateEngine()` function executes, it looks up a `StateMapRow` or `StateMapRowEx` array by calling `GetStateMap()` or`GetStateMapEx()`:

Hide Copy Code

```
void StateMachine::StateEngine(void)
{
    const StateMapRow* pStateMap = GetStateMap();
    if (pStateMap != NULL)
        StateEngine(pStateMap);
    else
    {
```

```
        const StateMapRowEx* pStateMapEx = GetStateMapEx();
        if (pStateMapEx != NULL)
            StateEngine(pStateMapEx);
        else
            ASSERT();
    }
}
```

Indexing into the `StateMapRow` table with a new state value a state functions is executed by calling`InvokeStateAction()`:

Hide Copy Code

```
const StateBase* state = pStateMap[m_newState].State;
state->InvokeStateAction(this, pDataTemp);
```

After the state function has a chance to execute, it deletes the event data, if any, before checking to see if any internal events were generated. One entire state engine function is shown below. The other overloaded state engine function (see attached source code) handles state machines with a `StateMapRowEx` table containing the additional guard/entry/exit features.

Hide Shrink Copy Code

```
void StateMachine::StateEngine(const StateMapRow* const pStateMap)
{
    const EventData* pDataTemp = NULL;

    // While events are being generated keep executing states
    while (m_eventGenerated)
    {
        // Error check that the new state is valid before proceeding
        ASSERT_TRUE(m_newState < MAX_STATES);

        // Get the pointer from the state map
        const StateBase* state = pStateMap[m_newState].State;

           // Copy of event data pointer
        pDataTemp = m_pEventData;

        // Event data used up, reset the pointer
        m_pEventData = NULL;

        // Event used up, reset the flag
        m_eventGenerated = FALSE;

        // Switch to the new current state
        SetCurrentState(m_newState);

        // Execute the state action passing in event data
        ASSERT_TRUE(state != NULL);
```

```
        state->InvokeStateAction(this, pDataTemp);

        // If event data was used, then delete it
        if (pDataTemp)
        {
            delete pDataTemp;
            pDataTemp = NULL;
        }
    }
}
```

The state engine logic for guard, entry, state, and exit actions is expressed by the following sequence. The`StateMapRow` engine implements only #1 and #5 below. The extended `StateMapRowEx` engine uses the entire logic sequence.

1. Evaluate the state transition table. If `EVENT_IGNORED`, the event is ignored and the transition is not performed. If `CANNOT_HAPPEN`, the software faults. Otherwise, continue with next step.
2. If a guard condition is defined execute the guard condition function. If the guard condition returns `FALSE`, the state transition is ignored and the state function is not called. If the guard returns `TRUE`, or if no guard condition exists, the state function will be executed.
3. If transitioning to a new state and an exit action is defined for the current state, call the current state exit action function.
4. If transitioning to a new state and an entry action is defined for the new state, call the new state entry action function.
5. Call the state action function for the new state. The new state is now the current state.

Generating events

At this point, we have a working state machine. Let's see how to generate events to it. An external event is generated by creating the event data structure on the heap using new, assigning the structure member variables, and calling the external event function. The following code fragment shows how a synchronous call is made.

Hide Copy Code

```
MotorData* data = new MotorData();
data->speed = 50;
motor.SetSpeed(data);
```

To generate an internal event from within a state function, call `InternalEvent()`. If the destination doesn't accept event data, then the function is called with only the state you want to transition to:

Hide Copy Code

```
InternalEvent(ST_IDLE);
```

In the example above, once the state function completes execution the state machine will transition to the Idle state. If, on the other hand, event data needs to be sent to the destination state, then the data structure needs to be created on the heap and passed in as an argument:

Hide Copy Code

```
MotorData* data = new MotorData();
data->speed = 100;
InternalEvent(ST_CHANGE_SPEED, data);
```

Hiding and eliminating heap usage

The `SetSpeed()` function takes a `MotorData` argument that the client must create on the heap. Alternatively, the class can hide the heap usage from the caller. The change is as simple as creating the `MotorData` instance within the `SetSpeed()` function. This way, the caller isn't required to create a dynamic instance:

Hide Copy Code

```
void Motor::SetSpeed(INT speed)
{
    MotorData* data = new MotorData;
    pData->speed = speed;

    BEGIN_TRANSITION_MAP                          // - Current State -
        TRANSITION_MAP_ENTRY (ST_START)            // ST_IDLE
        TRANSITION_MAP_ENTRY (CANNOT_HAPPEN)       // ST_STOP
        TRANSITION_MAP_ENTRY (ST_CHANGE_SPEED)     // ST_START
        TRANSITION_MAP_ENTRY (ST_CHANGE_SPEED)     // ST_CHANGE_SPEED
    END_TRANSITION_MAP(data)
}
```

On some systems, using the heap is undesirable. For those systems, I've included the `xallocator` fixed block allocator within the attached source code. It's optional, but when used it creates memory blocks from static memory or previously recycled heap memory. A single `XALLOCATOR` macro in the `EventData` base class provides fixed block allocations for all `EventData` and derived classes.

Hide Copy Code

```
#include "xallocator.h"
class EventData
{
public:
    virtual ~EventData() {}
    XALLOCATOR
};
```

For more information on xallocator, see the article "**Replace malloc/free with a Fast Fixed Block Memory Allocator**".

State machine inheritance

Inheriting states allows common states to reside in a base class for sharing with inherited classes. `StateMachine`supports state machine inheritance with minimal effort. I'll use an example to illustrate.

Some systems have a built in self-test mode where the software executes a series of test steps to determine if the system is operational. In this example, each self-test has common states to handle Idle, Completed and Failed states. Using inheritance, a base class contains the shared states. The classes `SelfTest` and `CentrifugeTest`demonstrate the concept. The state diagram is shown below.

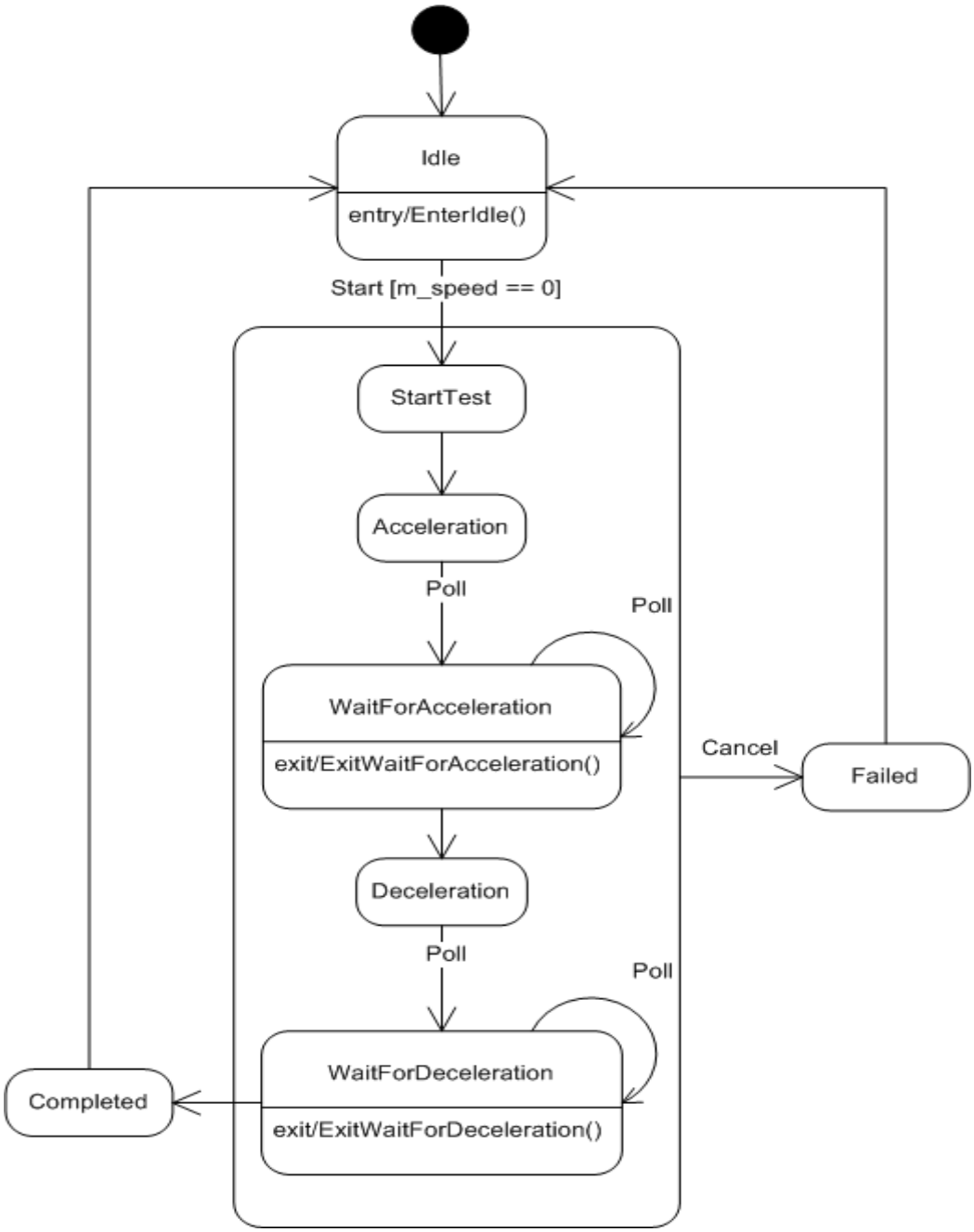

Figure 2: CentrifugeTest state diagram

`SelfTest` defines the following states:

1. Idle
2. Completed
3. Failed

`CentrifugeTest` inherits those states and creates new ones:

3. Start Test
4. Acceleration
5. Wait For Acceleration
6. Deceleration
7. Wait For Deceleration

The `SeftTest` base class defines the `States` enumeration and states but not the state map. Only the most derived state machine class within the hierarchy defines a state map.

Hide Copy Code

```
enum States
{
    ST_IDLE,
    ST_COMPLETED,
    ST_FAILED,
    ST_MAX_STATES
};

// Define the state machine states
STATE_DECLARE(SelfTest,     Idle,           NoEventData)
ENTRY_DECLARE(SelfTest,     EntryIdle,      NoEventData)
STATE_DECLARE(SelfTest,     Completed,      NoEventData)
STATE_DECLARE(SelfTest,     Failed,         NoEventData)
```

The `CentrifugeTest` class defines its state machine below. Take note of the "`ST_START_TEST = SelfTest::ST_MAX_STATES`" enumeration entry. It is critical that the derived class continue numbering states where the base class left off.

Hide Shrink Copy Code

```
enum States
{
    // Continue state numbering using the last SelfTest::States enum value
    ST_START_TEST = SelfTest::ST_MAX_STATES,
    ST_ACCELERATION,
    ST_WAIT_FOR_ACCELERATION,
    ST_DECELERATION,
    ST_WAIT_FOR_DECELERATION,
    ST_MAX_STATES
};
```

```
// Define the state machine state functions with event data type
STATE_DECLARE(CentrifugeTest,     Idle,                     NoEventData)
STATE_DECLARE(CentrifugeTest,     StartTest,                NoEventData)
GUARD_DECLARE(CentrifugeTest,     GuardStartTest,           NoEventData)
STATE_DECLARE(CentrifugeTest,     Acceleration,             NoEventData)
STATE_DECLARE(CentrifugeTest,     WaitForAcceleration,      NoEventData)
EXIT_DECLARE(CentrifugeTest,      ExitWaitForAcceleration)
STATE_DECLARE(CentrifugeTest,     Deceleration,             NoEventData)
STATE_DECLARE(CentrifugeTest,     WaitForDeceleration,      NoEventData)
EXIT_DECLARE(CentrifugeTest,      ExitWaitForDeceleration)

// State map to define state object order. Each state map entry defines a
// state object.
BEGIN_STATE_MAP_EX
    STATE_MAP_ENTRY_ALL_EX(&Idle, 0, &EntryIdle, 0)
    STATE_MAP_ENTRY_EX(&Completed)
    STATE_MAP_ENTRY_EX(&Failed)
    STATE_MAP_ENTRY_ALL_EX(&StartTest, &GuardStartTest, 0, 0)
    STATE_MAP_ENTRY_EX(&Acceleration)
    STATE_MAP_ENTRY_ALL_EX(&WaitForAcceleration, 0, 0, &ExitWaitForAcceleration)
    STATE_MAP_ENTRY_EX(&Deceleration)
    STATE_MAP_ENTRY_ALL_EX(&WaitForDeceleration, 0, 0, &ExitWaitForDeceleration)
END_STATE_MAP_EX
```

The state map contains entries for all states within the hierarchy, in this case `SelfTest` and `CentrifugeTest`. Notice the use of the _EX extended state map macros so that guard/entry/exit features are supported. For instance, a guard condition for the StartState is declared as:

Hide Copy Code

```
GUARD_DECLARE(CentrifugeTest, GuardStartTest, NoEventData)
```

The guard condition function returns `TRUE` if the state function is to be executed or `FALSE` otherwise.

Hide Copy Code

```
GUARD_DEFINE(CentrifugeTest, GuardStartTest, NoEventData)
{
    cout << "CentrifugeTest::GuardStartTest" << endl;
    if (m_speed == 0)
        return TRUE;    // Centrifuge stopped. OK to start test.
    else
        return FALSE;   // Centrifuge spinning. Can't start test.
}
```

Base class state machines support external events but can’t use a transition map table. Only the most-derived class within the hierarchy may define a transition map. To have external event functions

within lower hierarchy levels, use `GetCurrentState()` and `ExternalEvent()` manually and without macro support. The `SelfTest`class has one external event function:`Cancel()` . As shown in the code below, the state machine transitions to the Failed state if the current state is not Idle.

Hide Copy Code

```
void SelfTest::Cancel()
{
    // State machine base classes can't use a transition map, only the
    // most-derived state machine class within the hierarchy can. So external
    // events like this use the current state and call ExternalEvent()
    // to invoke the state machine transition.
    if (GetCurrentState() != ST_IDLE)
        ExternalEvent(ST_FAILED);
}
```

The state machine inheritance technique expressed here does not implement a Hierarchical State Machine (HSM). An HSM has different semantics and behaviors including, among other things, a hierarchical event processing model. This feature explained here offers code reuse by factoring common states into a base class, but the state machine is still considered a traditional FSM.

State function inheritance

State functions can be overridden in the derived class. The derived class may call the base implementation if so desired. In this case, `SelfTest` declares and defines an Idle state:

Hide Copy Code

```
STATE_DECLARE(SelfTest, Idle, NoEventData)

STATE_DEFINE(SelfTest,  Idle, NoEventData)
{
    cout << "SelfTest::ST_Idle" << endl;
}
```

`CentrifugeTest` also declares and defines the same Idle state with the same event data type.

Hide Copy Code

```
STATE_DECLARE(CentrifugeTest,  Idle, NoEventData)

STATE_DEFINE(CentrifugeTest,   Idle, NoEventData)
{
    cout << "CentrifugeTest::ST_Idle" << endl;

    // Call base class Idle state
    SelfTest::ST_Idle(data);
    StopPoll();
```

```
}
```

The `CentrifugeTest::ST_Idle()` function calls the base implementation `SelfTest::ST_Idle()`. State function inheritance is a powerful mechanism to allow each level within the hierarchy to process the same event.

In the same way a state function is overridden, a derived class may also override a guard/entry/exit function. Within the override, you decide whether to call the base implementation or not on a case-by-case basis.

StateMachine compact class

If the `StateMachine` implementation is too large or too slow due to the virtual function and typecasting, then the compact version is available at the expense of no type checking of the member function pointers. The compact version is only 68 bytes (on Windows release build) vs. 448 bytes on the non-compact version. See*StateMachineCompact.zip* for the source files.

Notice that we have to use `reinterpret_cast<>` operator within the `STATE_MAP_ENTRY` macro to cast the derived class member function pointer to a `StateMachine` member function pointer.

Hide Copy Code

```
reinterpret_cast<StateFunc>(stateFunc)
```

It is necessary to perform this upcast since the `StateMachine` base class has no idea what the derived class is. So, it is imperative that the entries provided to `STATE_MAP_ENTRY` are really member functions of an inheriting class and that they conform to the state function signature discussed earlier (see State Functions section). Otherwise bad things will happen.

On most projects, I'm not counting CPU instructions for the state execution and a few extra bytes of storage isn't critical. The state machine portion of my projects have never been the bottleneck. So I prefer the enhanced error checking of the non-compact version.

Multithread safety

To prevent preemption by another thread when the state machine is in the process of execution, the`StateMachine` class can use locks within the `ExternalEvent()` function. Before the external event is allowed to execute, a semaphore can be locked. When the external event and all internal events have been processed, the software lock is released, allowing another external event to enter the state machine instance.

Comments indicate where the lock and unlock should be placed if the application is multithreaded. Note that each `StateMachine` object should have its own instance of a software lock. This prevents a single instance from locking and preventing all other `StateMachine` objects from executing. Note,

software locks are only required if a `StateMachine` instance is called by multiple threads of control. If not, then locks are not required.

Alternatives

Occasionally you need something more powerful to capture the behavior of a system using events and states. The version presented here is a variation of a conventional FSM. A true Hierarchical State Machine (HSM), on the other hand, can significantly simplify the solution to certain types of problems. Many good HSM projects exist ready for you to explore. But sometimes a simple FSM is all that you need.

Benefits

Implementing a state machine using this method as opposed to the old switch statement style may seem like extra effort. However, the payoff is in a more robust design that is capable of being employed uniformly over an entire multithreaded system. Having each state in its own function provides easier reading than a single huge switch statement, and allows unique event data to be sent to each state. In addition, validating state transitions prevents client misuse by eliminating the side effects caused by unwanted state transitions.

I've used variations of this code for self-test engines, a gesture recognition library, user interface wizards, and machine automation, among other projects. This implementation offers easy use for the inheriting classes. With the macros it lets you just "turn the crank" without much thought given to the underlying mechanics of how the state engine operates. This allows you more time to concentrate on more important things, like the design of the state transitions and state function implementation.

History

- 23rd March, 2016
  - First release
- 28th March, 2016
  - Updated attached StateMachine.zip to help with porting.
- 3rd April, 2016
  - Corrected an error in figure 1.
- 17th April, 2016
  - Updated attached StateMachine.zip source code to include xallocator bug fixes.

License

About the Author

David Lafreniere

United States
I've been a profession software engineer for over 20 years. When not writing code, I enjoy spending time with the family, camping and riding motorcycles around Southern California.