DECK: As part of a project to develop a library of software for programming a robot, the authors developed C++ classes and utilities for state machine implementation that are highly portable and applicable to a wide variety of event-driven systems.

The RHex robot was designed with a preference for simplicity, minimal actuation, and sensing. As a result, it is distinguished among walking robots in its ability to easily negotiate broken terrain and obstacles exceeding its ground clearance at relatively high speeds (see rhex.net for more). Slightly different versions of the robot are being developed at several universities by many researchers, each of whom is interested in experimenting with one of various mechanical, sensory, electronic and behavioral alternatives. RHexLib is an attempt to meet the demands of these researchers by providing a design environment that facilitates this joint development and the easy exchange of software. It provides a runtime environment that manages the various tasks the robot must perform to sense and actuate in the world. It also serves as an API for programmers to easily code and integrate new behaviors and other components.

One of RHexLib’s most useful tools for programming new robot behaviors is its state machine facility. State machines are a useful abstraction, well established in the control community, for systems with discrete states that are switched among based on events sensed in the world. A common first example of a state machine is a vending machine, in which transitions between system states are triggered by coin insertions and button presses. Larger systems with discrete state transitions can be quite complex and difficult to manage; one state machine in the RHex robot has fifteen states and sixteen events. Furthermore, RHex may run several state machines simultaneously, dramatically increasing the number of states to be managed. We introduced RHexLib’s state machine facility to manage these state-event systems, thereby making the code more modular and easier to read and maintain.

State machines are very useful and fundamental objects. They are used, for example, in automated factories and automotive control and there are numerous software packages available for modeling and controlling systems using state machines or similar constructs. MATLAB’s Stateflow is one such package. It allows the user to interface discrete logical control with continuous controllers represented in Simulink. Without a general tool for managing the discrete logic of a controller, embedded programmers have to resort to ad-hoc coding with multiple if-then-else or switch-case statements, often resulting in programs that are difficult to read and navigate. The good news is that incorporating state machines into your own projects is not difficult.

In this article, after providing a brief overview of RHexLib, we describe in detail the library’s state machine facility. We explain how to represent states and events and how to
put them together into a state machine, and we give a detailed example. The ideas are simple and powerful and may be easily adapted to applications other than robot programming, as well as systems not using RHxLib.

**RHxLib**

The source code for RHxLib, the examples in this article, and complete documentation are all freely available at [rhex.sourceforge.net](http://rhex.sourceforge.net). As supplied, the code will compile and run on an Intel x86 running either Linux or QNX. RHxLib also includes a RHx simulator so you do not need one of the robots to execute the compiled code.

Many of the tasks a robot needs to carry out are periodic and repetitive. For example: reading analog inputs, computing control commands, and updating analog outputs. Consequently, robot control software commonly takes the form of a list of update functions, one for each task, executed at a required frequency (e.g., 1 kHz).

Examples of tasks in RHx include basic analog and digital I/O, PWM conversion, sensor interpretation, and timing. Higher-level (behavioral) tasks include data logging, sensor filtering, reference trajectory generation, and motor calibration. Most of these tasks are common to all but the simplest robots. An application that uses RHxLib is likely to require most of these tasks to be managed to support experimental controllers and supervisory machines that activate and deactivate behaviors based on user input from the remote control.

![The life cycle of a module.](image)

**Figure 2:** The life cycle of a module.

In RHxLib, a task is encoded as a *module*, which means that it is derived from a C++ abstract base class called `Module`. The class `Module` contains five pure virtual methods that must be instantiated by derived classes. These are

```cpp
virtual void init(void);
virtual void activate(void);
```
virtual void update(void);
virtual void deactivate(void);
virtual void uninit(void);

These methods have special meanings to an object called the *module manager*, which is responsible for maintaining a list of modules and executing their code. The life cycle of a module within the module manager is shown in Figure 1. A module’s *init* and *uninit* methods are called when it is added to and removed from the module manager using the *MMAddModule* and *MMRemoveModule* functions respectively. The methods are used to initialize and cleanup the module’s data structures. The *activate* and *deactivate* methods are called when the module is activated or deactivated using the *MMActivate* and *MMDeactivate* functions, respectively. These functions perform activities such as activating and deactivating *other* modules that may be required. The *update* method is called at a certain (configurable) frequency by the module manager, once the module has been added and activated, and until the module is deactivated. Thus, the module manager acts like a kernel of a simple operating system and modules essentially encode separate processes or threads.

Some modules, such as those that send commands to motors, can only be activated and used by one other module at a time. A module that requires such a “single-user” module, uses the functions *MMGrabModule* and *MMReleaseModule*, which behave similarly to the *MMActivate* and *MMDeactivate* functions except that they do not allow more than one module to use the “grabbed” module at a time.

All of the tasks described above are encoded as modules, from low-level hardware tasks to high-level behavior controllers. The basic module interface, however, has some limitations, especially when it comes to event driven controllers with discrete states. Without some organization, the *update* methods of this type of controller can be quite complicated. RHexLib’s state machine facility helps solve this problem by allowing a module’s *update* method to behave differently according to the *state* of the module.

**State Machines**

Probably the most common example of a state machine is the vending machine controller. Such a controller has a number of states, corresponding to the amount of money so far inserted or the sequence of buttons so far pressed. It also monitors the occurrence of a number of events, such as whether a quarter has just been inserted, or whether a certain product is out of stock. In each state, certain events will cause a transition to another state and others will not. So when the machine is in state “no money deposited” inserting a quarter changes the state to “25 cents deposited”; from the “no money” state, pressing the button for a soda does nothing. In robotics and manufacturing, state machines behave essentially the same way, possibly with the added complication that while a state is active, a corresponding continuous controller may be run as with an automatic garage door that is either closed, open, or *in the process of opening (or closing).*
In RHexLib, a state machine is a certain type of module, which implements the virtual methods described above. State machine controllers are modules derived from the StateMachine class, which is itself derived from the Module class. An example state machine, discussed in detail later, is shown in Figure 2. Before we describe state machines properly, we describe how events and states are represented.

An Event is an abstract base class with one virtual Boolean method, called check, which takes no arguments. Derived classes must implement the check method to return true or false based on checking some condition in the world. For example, a RHex program may have an event that returns true or false based on whether the robot is upside-down or rightside-up. Coding events as objects in their own right allows them to be reused and makes the code easy to manage.

A State is an abstract base class with three methods that need to be defined by any derived state. These are

```c
virtual void entry(void);
virtual void during(void);
virtual void exit(void);
```

At the time a state becomes activated, the state machine that “owns” it calls its entry method once. This allows the state to perform any one-time set up procedures it may require. While it is active, its during function essentially replaces the update method of the state machine, which you will recall is itself a module (see Listing 1). The during method is where any activity requiring continuous updates, such as the control of a motor, takes place. When a transition occurs out of a state, its exit method is called once, allowing the state to, for example, release whatever resources it may have been using.
Each state also contains a list of outgoing arcs (of class Arc in RHexLib), each of which consists of an event and a target state. A state machine keeps a pointer to a currently active state and continuously checks its outgoing arcs. As soon as one of them becomes active (i.e., its event’s check method returns true), the current state is set to the target state of the arc.

Now, as we have said, state machines are modules derived from the class StateMachine. This base class implements the init, deactivate, and uninit methods of the base class Module as empty functions. Listing 1 shows how the activate and update methods are implemented. The only other difference between a state machine module and a basic module is that the former also contains a pointer to an initial State object. This pointer is usually set up in the state machine’s constructor using the initialize method.

A programmer does the following to set up a state machine:

1) Define the event objects
2) Define the state objects
3) Add arcs to the states
void StateMachine::activate ( void ) {
    current = initial;
    current->entry();
}

void StateMachine::update( void ) {
    Arc * a = current->getActiveArc();
    if ( a != NULL ) {
        current->exit();
        current = a->target;
        current->entry();
    }
    current->during();
}

Listing 1: The StateMachine class’s activate and update methods.

4) Set the initial state of the state machine

Because state machines all have the same basic form, the RHexLib distribution provides a Perl script called sm-setup.pl (in the RHexLib/util directory), which converts a simple textual description (e.g., Listing 2) of a state machine into corresponding C++ code that defines the machine. The resulting code is intended to be a skeleton of the final code. That is, the methods of Event, State, and StateMachine in the skeleton code need to be augmented with the intended functionality of the state machine.

Example

RHexLib and its state machine facility may be used to produce a wide variety of state-event based controllers not necessarily related to robotics. However, because we program robots, we give a more substantial example of a state machine for the RHex robot to demonstrate how the rest of RHexLib works in conjunction with state machines. The example we consider is a supervisor controller, shown in Figure 2 and described textually in Listing 2. The supervisor starts and stops other controllers based on user input via a remote control and also on events occurring in the world. This somewhat simple supervisor activates three other controller types: one for calibrating the leg motors, one
Transition unCalibrated startCommand calibrating
Transition calibrating calFail unCalibrated
Transition calibrating calSuccess standing
Transition standing doneStanding ready
Transition ready accWalkCommand accelerating
Transition ready walkCommand walking
Transition accelerating upToSpeed walking
Transition accelerating noCommand ready
Transition walking noCommand ready
Transition walking stopCommand decelerating
Transition decelerating doneDecel ready
Initial unCalibrated

Listing 2: A text file describing the states and arcs in Figure 2.

for standing the robot up, and one for walking. These other controllers are all encoded as state machines and are part of the RHexLib distribution.

void Supervisor::Calibrating::entry ( void ) {
    for ( int i = 0; i < 6; i++ ) {
        OWNER->calibMachine[i]->setMode( CalibMachine::GROUND );
        MMGrabModule ( OWNER->calibMachine[i], owner );
    }
}

void Supervisor::Calibrating::during ( void ) {}

void Supervisor::Calibrating::exit ( void ) {
    for ( int i = 0; i < 6; i++ )
        MMReleaseModule ( OWNER->calibMachine[i], owner );
}

Listing 3: The entry, during and exit methods of the calibrating state.

To code the Supervisor state machine, we first create the skeleton code for it from Listing 2 using the Perl script discussed in the last section. (Because of the length of the resulting skeleton code, we do not list it here. The entirety of the code for this example is included with the RHexLib distribution in the directory RHexLib/examples/esp.) We add functionality to the skeleton code by changing each event’s check definitions and the state’s virtual methods. In the subsequent listings, we show the skeleton code in black text and the code we added in blue. (FY: Please note the previous sentence when laying out the listings. –MB) We also add data and methods to the Supervisor prototype and functionality to its activate, deactivate, and init methods.
Our state machine functions as follows: It is initially in the **uncalibrated state**. When the user issues a start command (via a remote control unit described shortly), the state machine switches to state **calibrating** and activates a calibration state machine for each leg. These machines set the zero position of the motor encoders and determine the polarity of the motor drives. The calibration machines are themselves state machines with about ten states each. Six of these machines are run in parallel in the **calibrating state**. The simplicity of the code in light of this complex task speaks to the power of state machines.

The **entry**, **during**, and **exit** methods of the calibrating state are shown in Listing 3. In the **entry** method, each calibration machine mode is set to **GROUND**, meaning that the calibration machine swings its leg until the toe hits the ground and determines where it is. Then, using the **MMGrabModule** function, the calibration module is activated. The **OWNER** is a pointer to the **Supervisor** object that contains the **Calibrating** event and is assigned when the event object is constructed. The **during** method of the calibrating state does nothing, although the calibration state machines are running separately. The **exit** method of the state releases (and thereby deactivates) the calibration machines. This technique of grabbing and releasing required modules in the **entry** and **exit** methods of states is common with RHexLib.

```cpp
bool Supervisor::CalSuccess::check ( void ) {
    for ( int i = 0; i < 6; i++ )
        if ( OWNER->calibMachine[i]->getStatus() != CalibMachine::SUCCESS )
            return false;
    return true;
}
```

**Listing 4:** The **check** method of the **CalSuccess** event.
bool Supervisor::AccWalkCommand::check ( void ) {
  return bool ( OWNER->rc->leftStick() == RemoteControl::WEST
  && ( OWNER->rc->rightStick() == RemoteControl::NORTH
  || OWNER->rc->rightStick() == RemoteControl::SOUTH ) );
}

Listing 5: The check method of the AccWalkCommand event.

The calSuccess and calFail events check for the success or failure of each the calibration machines. Occasionally, calibration fails due to low battery voltage or the presence of obstacles in the environment. If calibration fails for any leg, the unCalibrated state is reentered. If the calibration of each leg succeeds, then the standing state is entered. Listing 4 shows an example of an event::check method where the success of the calibration machines is checked.

Once the standing controller is done, the Supervisor enters the Ready state. In this state, the Supervisor waits for input via the remote control. The user may either start the walking controller at full speed via the WalkCommand event or may start the walk controller at a slower speed and then accelerate to full speed. The latter is accomplished via the AccWalkCommand event. These events check the state of the remote control input to RHex. We presently control the robot remotely using a standard two-joystick, radio frequency (RF) remote controller. On the robot, RF commands are converted into pulse width modulation (PWM) signals and are fed into RHex’s processor via an analog input board. In software, the PulseWidth module uses the AnalogInput module to read and convert the PWM signals into floating point numbers. These numbers are then converted into values for each joystick that may be read using the leftStick and rightStick methods of the RemoteControl module. These return a value of NORTH, SOUTH, etc. For example, as shown in Listing 5, the AccWalkCommand checks that the left joystick is pushed to the right (WEST) and that the right joystick is pushed either up (NORTH) or down (SOUTH).

The accelerating, walking, and decelerating states grab the walking controller and control its speed as required. For example, in the state accelerating, whose methods are shown in Listing 6, the entry method sets up a profile for the speed of the walking controller so that at the beginning of the state it has speed 1.0 (meaning the legs take 1.0 second per cycle) and at the end of the state it has speed 0.5. The entry method also grabs the walking controller. The during method of this state simply gets the value of the profile for the current time and then sets the speed of the walking controller.
The rest of the state machine events and states are similar. There are also some basic changes to be made to the rest of the skeleton code generated. Interested readers can investigate the full source code of this example, which is available online.

**Conclusion**

The state machine abstraction is a fundamental tool for discrete event system control. Example applications of the state machine range from robotics to manufacturing to automatic transmissions. Here we have described how we implement state machines in RHexLib, providing one representation that happens to fit quite nicely with the module-based approach. We hope that our ideas and classes may find other applications, in other robots and, more generally, in other embedded applications.

The state machine idea is not limited to RHexLib, of course. The ideas are used in many control software packages. Hopefully, the short description we have given here can serve as a guide to incorporating general state machine tools into any project. The gains in code reusability, ease of maintenance, and elegance are worth the small overhead of thinking in terms of states and events.