PART

IV Hands-on Device Construction

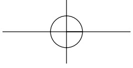
Once you have gained familiarity with our architecture and approach, Part IV will show you how the "rubber meets the road." The four chapters in this section present the design and implementation of a full simulator, to demonstrate simple to moderately complex designs in practice. Certain sections, particularly in terms of the design, will be accessible to a wider audience than the technical programming details.

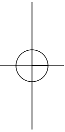
The most basic simulator is presented in Chapter 15. This is a crane-like arm you can move about using the joystick component developed in Chapter 13. The chapter is intended to describe the basic design perspective and interesting challenges encountered in the process of building the crane arm, rather than as an exhaustive, detailed description of its construction. Chapter 16, on the other hand, describes in great detail how to build a multi-modal analog/digital watch, and shows how to extend the original design when your client wants to add or change functionality.

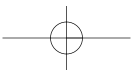
Chapter 17 develops a commercial fish finder, which demonstrates concepts involved with multiple and overlapping software screen elements. Chapter 18 teaches you how to build a simple, fictitious cellular phone. We revisit the implementation in Chapter 27 to demonstrate necessary extensions to permit integration into and manipulation by a sample simulation-based presentation.

After reading Part IV, you will possess the knowledge to connect the simulator architecture theory with the implementation considerations involved with designing real-world devices. You still will need practice and experience to make these processes more natural, but you will have a solid practical base for your own projects.

2667-ch15 10/16/02 8:46 AM Page 254







CHAPTER

15 Modeling a Crane Arm

OBJECTIVES

In this chapter you will:

- Construct a crane that can move left, move right, extend, retract, open a claw, and close a claw
- Utilize a joystick component
- Understand the separation of the model and control object layers
- Examine a strategy for nesting movie clips within a virtual device simulation

This chapter will walk you through the design and construction and of a simple crane simulation. It will include a description of the user interface, the control and model objects, and the event handlers. It will further provide a discussion of the challenges involved in building this type of simulation.

USER INTERFACE

The crane user interface contains a joystick, a set of indicator lamps, and two sector dials. The space bar is also part of the user interface and is used to control the crane's claw.

Joystick

The joystick controls the horizontal and vertical movement of the crane. It can move in any direction in two-dimensional (2D) space. Changing the position of the joystick generates a *JStickChg* message to the control object event handler. The message also sends a parameter object containing the *x* and *y* coordinate position of the joystick head movie clip. The control object uses these values to control the crane's movement. When configuring the joystick component, you must supply the name of the listener, in this case *cranectrl* as illustrated in figure 15–1.

Space Bar

The space bar is used to control the opening and closing of the crane's claw. An invisible movie clip in the root timeline captures the space bar press event and generates a "grasp" message to the control object's event handler. If the claw is

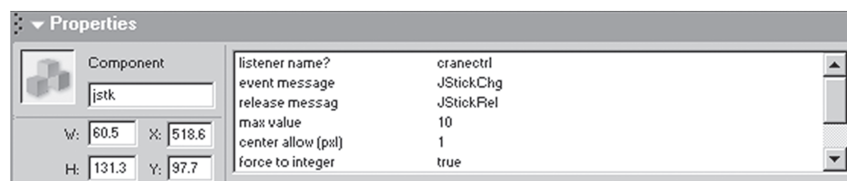


FIGURE 15–1
The property window for the joystick component.

2667-ch15 10/16/02 8:46 AM Page 256

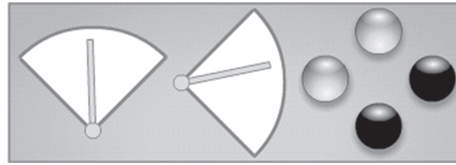


FIGURE 15–2
The indicator panel for the crane arm.

open, pressing the space bar triggers the claw to close, and vice versa. The code on the keyPress handler is as follows.

```
on (keyPress "<Space>") {
    cranectrl.ieh("grasp", true);
}
```

Indicator Lamps

The indicator panel consists of four indicator lights. The lights indicate the "articulation" states of the control object. In figure 15–2, for example, the lights indicate that the control object is requesting the crane to move up and to the left.

Sector Dials

Two sector dials are provided to inform the operator of the position of the crane relative to its range of movement. The horizontal sector dial indicator needle moves as the crane moves horizontally, whereas the vertical sector dial moves as it moves vertically.

CONTROL OBJECT

The control object processes all user interface messages and controls the behavior of the panel indicators. It is also responsible for sending messages to the model object to articulate the crane.

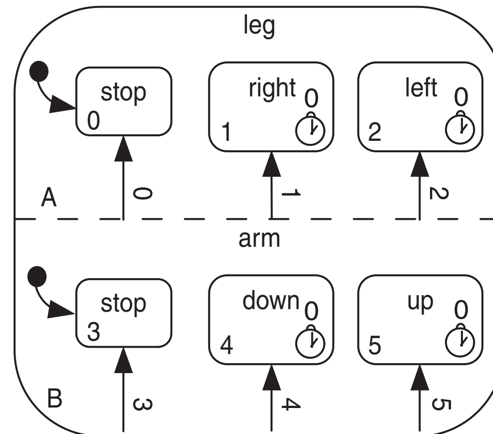
Top-Level State Network

The top-level state network illustrated in figure 15–3 consists of two concurrent states denoted on the statechart as A and B. State A represents the horizontal movement of the leg, whereas State B represents the vertical extension and retraction of the arm. Each concurrent state starts in a default resting state and can transition into either of its movement states. We model the interface with two concurrent states so that they can work independently of the others. For instance, the leg can be commanded to slide while the arm is retracting.

2667-ch15 10/16/02 8:46 AM Page 257

FIGURE 15–3

The crane's top-level network statechart.



Leg Control

The sections that follow explore the leg control of the device.

States and Transitions

State A contains three exclusive states that represent the control states of the leg. State *s0* is the default state and represents the interface when the leg is at rest. State *s1* represents the interface when commanding the leg to move left, whereas *s2* represents the interface when commanding the leg to move right. Because the parent state *crane_net* provides transitions directly to each substate, substates can be activated at any time.

Triggers

Anytime the joystick is moved, it sends a *JStickChg* message to the control object event handler that includes the joysticks *x* and *y* coordinates. The control object event handler uses the *x* coordinate to determine which leg state transition to trigger. As the following section of the control object event handler illustrates, a message containing an *x* coordinate equal to –4 (left of the joystick centerline) triggers transition 2, which activates state *s1.*

Recall that *s1* is the control state for commanding the crane to move left. Alternatively, as the joystick moves to the right of the center point, it will send a message with a positive *x* coordinate value to the event handler, triggering transition 1 and activating state *s2.* When the joystick is released, it will return to center and trigger transition 0, returning to *s0* (the "rest" state).

xchgBy Property

In order to control how fast the crane moves along the horizontal axis, the *x* coordinate is stored in the *xchgBy* property of the control object and sent to the model object to control the articulation rate, as follows.

```
cranectrl.ieh = function(msgid, val) {
    ...
    if (msgid == "JStickChg"){
        cranectrl.change.xchgBy = val.x;
        cranectrl.change.ychgBy = val.y;
        if (val.x == 0 and val.y == 0){
            crane_net.chg_st(0);
            crane_net.chg_st(3);
        }
        else if (val.x <> 0 and val.y <> 0){
            if (val.x < 0){
                crane_net.chg_st(2);
            } else if (val.x > 0){
                crane_net.chg_st(1);
            }
            ...
        }
    }
}
```

Actions

When the joystick is moved away from the center rest position, a transition is triggered to enter either *s1* or *s2*. As the following code indicates, the *s1* enter actions set the *slideRight* lamp movie clip to its ON state, informing the operator that the control object received the request to move right. The *s1* enter action also sends a *slideRight* message to the model object, with the *cranectrl.xchgBy* property as a parameter.

```
s0.entacts = function() {
cranectrl.notifylisteners("slideStop", cranectrl.change);
}

s1.entacts = function() {
    slideRightLamp.gotoAndStop(2);
    cranectrl.notifylisteners("slideRight", cranectrl.change);
}

s1.lvacts = function() {
    slideRightLamp.gotoAndStop(1);
}

getXpos = function () {
    Xpos = parseInt(cranectrl.AskOracle("Xpos")*100);
    Xpos.text = Xpos;
    XposSec.setNeedle(Xpos);
}

s1.add_pulse_activity(0, 100, "getXpos", this);
...
```

While state s1 is active, the s1 pulse activity polls the model object requesting the x position of the crane, and displays the result in the horizontal sector dial. Upon returning to the s0 “rest” state, the s1 leave action returns the “slideRight” lamp movie clip to its “off” state. The enter actions, pulse activities and leave actions for state s2 “slideLeft” work similarly.

Arm Control

States and Transitions

State B contains three exclusive states that represent the control states of the arm. State *s3* the default state, representing the interface when the arm is at rest. State *s4* represents the interface when commanding the arm to extend, whereas *s5* represents it when commanding the arm to retract. Because the parent state *crane_net* provides transitions directly to each substate, substates can be activated at any time.

Triggers

The control object event handler uses the joystick *y* coordinate to determine which transition to trigger. As the following section of the control object event handler illustrates, a joystick *y* coordinate equal to 4 (below the joystick horizontal centerline), triggers transition 5, activating state *s4.* Recall that *s4* is the control state for commanding the crane to extend. Alternatively, as the joystick moves above the center point, it will send a message with a negative Y coordinate to the event handler, triggering transition 6 and activating state *s5.* When the joystick is released, it will return to center and trigger transition 3, returning to state *s3* (the “rest” state).

yChgBy Property

To control how fast the crane moves along the vertical axis, the *y* coordinate is stored in the *ychgBy* property of the control object and sent to the model object to control the articulation rate, as follows.

```
cranectrl.ieh = function(msgid, val) {
    ...
    if (msgid == "JStickChg"){
        cranectrl.change.xchgBy = val.x;
        cranectrl.change.ychgBy = val.y;
        if (val.x == 0 and val.y == 0){
            crane_net.chg_st(0);
            crane_net.chg_st(3);

        else if (val.x <> 0 and val.y <> 0){
            ...
            if (val.y < 0){
                crane_net.chg_st(5);
            } else if (val.y > 0){
```

2667-ch15 10/16/02 8:46 AM Page 260


```
                crane_net.chg_st(4);
            }
        }
    }
}
```

Actions

When the joystick is moved from center, a transition is triggered to enter either *s4* (extend)or *s5* (retract). As the following code indicates, the *s4* enter actions set the "extend" lamp movie clip to its ON state, informing the operator that the control object received the request to extend. The *s1* enter action also sends a *liftdown* message to the model object, with the *cranectrl.ychgBy* property as a parameter.

```
s3.entacts = function() {
    cranectrl.notifylisteners("liftStop", cranectrl.change);
}

s4.entacts = function() {
    extendLamp.gotoAndStop(2);
    cranectrl.notifylisteners("liftdown", cranectrl.change);
}

s4.lvacts = function() {
    extendLamp.gotoAndStop(1);
}

getYpos = function () {
    Ypos = parseInt(cranectrl.AskOracle("Ypos")*100);
    Ypos.text = Ypos;
    YposSec.setNeedle(Ypos);
}

s4.add_pulse_activity(0, 100, "getYpos", this);
```

While *s4* is active, the *s4* pulse activity polls the model object requesting the *y* position of the crane arm, and displays this value in the vertical sector dial. Upon returning to the *s3* "rest" state, the *s4* leave action returns the "extend" lamp movie clip to its OFF state. The enter actions, pulse activities, and leave actions for the *s5* "retract" work similarly.

Claw Control

Because the interface does not provide any indication of the claw's state, the control object event handler simply sets the *claw_open* property and forwards the relevant messages to the model object. The space bar generates a message of "grasp" and sends it to the control object event handler. When this message is received, the event handler evaluates the *claw_open* property to determine its current state.

2667-ch15 10/16/02 8:46 AM Page 261

If *cranectrl.claw_open* is false, it will be set to true and the "grasp" message generated by the space bar will be forwarded to the model object. If the space bar is pressed again, and *cranectrl.claw_open* is true, it will be set to false and again the message will be forwarded to the model object. The following shows the claw control coding.

```
cranectrl.ieh = function(msgid, val) {
    if (msgid == "grasp" and cranectrl.claw_open == false) {
        cranectrl.claw_open = true;
        cranectrl.notifylisteners("handopen");
    } else if (msgid == "grasp" and cranectrl.claw_open == true) {
        cranectrl.claw_open = false;
        cranectrl.notifylisteners("handclose");
    }
    ...
}
```

MODEL OBJECT

The sections that follow explore the model object shown in figure 15–4. The model object is responsible for receiving and processing events from the control object. Upon receiving these control object events, the model object

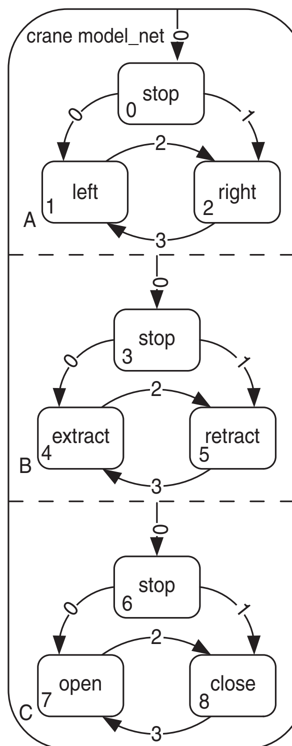


FIGURE 15–4
The top-level statechart for the crane's model layer.

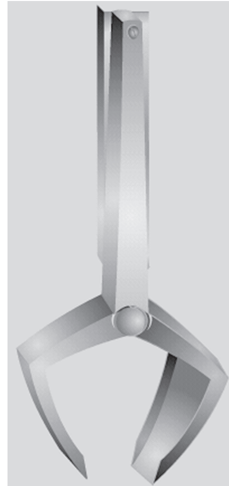


FIGURE 15–5
The crane assembly movie clip consists of a leg, an arm, and a claw.

articulates the crane by changing the position properties of the leg, arm, and claw movie clips.

Top-Level State Network

The top-level *cranemodel_net* network illustrated in figure 15–4 contains three concurrent states A for the leg, B for the arm, and C for the claw. Each state starts in a default resting state and then can transition to either of its movement states. Separate concurrent states are used so that each articulation is performed independently of the others. For instance, the leg can be rotating while the arm is retracting.

Crane Movie Clip Hierarchy

The crane assembly illustrated in figure 15–5 consists of five nested movie clips. The outermost movie clip instance is named *leg.* The leg is positioned on the stage so that the top portion is slightly out of view. The *arm* movie clip is located within the *leg* movie clip. By placing the *arm* movie clip within the *leg* movie clip, we can slide the assembly by simply changing the *x* and *y* position properties of the *leg* movie clip.

The *claw* movie clip is located within the *arm* movie clip, inheriting the position properties of its parent movie clips. The claw's registration point is positioned at the base of the arm. The two *finger* movie clips are located within the *claw* movie clip and are used to animate the grasping of the crane.

Nesting the sections of the crane in this manner provides a convenient way of animating the various sections of the crane. For example, if we rotate the *arm* movie clip, the claw will move with it. If we slide the assembly left, all of its children movie clips retain their relative positions.

Leg Articulation

The sections that follow explore the leg articulation of the device.

States and Transitions

State A contains three exclusive states that represent the horizontal movement of the leg assembly. State *s0,* the default, represents the leg assembly at rest. State *s1* represents movement to the left, whereas state *s2* represents movement to the right. Although the leg substate network looks different from the leg substate network in the control object, they are functionally identical.

Triggers

The model object event handler receives messages exclusively from the control object. Each message will trigger a transition to one of the articulation states. As illustrated in the following section of the model object event handler code, a *slideLeft* message triggers a transition to *s1,* the "slide left" state, whereas a *slideRight* message triggers a transition into *s2,* the "slide right" state. A *slideStop* message triggers the transition back into *s0,* the leg assembly rest state.

```
cranemodel.ieh = function(msgid,val){
    cranemodel.xchgBy = val.xchgBy;
    cranemodel.ychgBy = val.ychgBy;
    if (msgid == "slideStop"){
        if (s1.isActive() or s2.isActive()){
            cranemodel_net.chg_st(0);
        }
    } else if (msgid == "slideLeft"){
        xchgBy.text = cranemodel.xchgBy;
        if (s0.isActive()){
            s0.chg_st(0);
        } else if (s2.isActive()){
            s2.chg_st(0);
        }
    } else if (msgid == "slideRight"){
        xchgBy.text = cranemodel.xchgBy;
        if (s0.isActive()){
            s0.chg_st(1);
        }else if (s1.isActive()){
            s1.chg_st(0);
        }
    }
    ...
}
```

2667-ch15 10/16/02 8:46 AM Page 264

Actions

While *s1* is active, the *s1* pulse activity will change the leg assembly's *x* property until it reaches its predefined limit. The leg assembly will move at the rate contained in the *xChgBy* property of the model object. Similarly, the *s2* pulse activity will animate the leg assembly to the right until it reaches the limit. This functionality is shown in the following code.

```
function slideleftfn () {
    if (leg._x > -750){
        leg._x -= cranemodel.xchgBy;
    }
}

function sliderightfn () {
    if (leg._x < - 250){
        leg._x += cranemodel.xchgBy;
    }
}

s1.add_pulse_activity(0, 100, "slideleftfn", this);
s2.add_pulse_activity(0, 100, "sliderightfn", this);
```

Arm Articulation

The sections that follow explore the arm articulation of the device.

States and Transitions

State A contains three exclusive states that represent the horizontal movement of the leg assembly. State *s3,* the default, represents the arm assembly at rest. State *s4* represents arm extension, whereas state *s5* represents arm retraction. Because each state has a transition to the other states, any state can be activated at any time.

Triggers

As illustrated in the following section of the model object event handler code, a *liftdown* message triggers a transition to *s4,* the extend state, whereas a *liftup* message triggers a transition into *s5,* the retract state. A *liftStop* message triggers the transition back into *s3,* the arm assembly rest state.

```
cranemodel.ieh = function(msgid,val){
    ...
    } else if (msgid == "liftStop"){
        if (s4.isActive() or s5.isActive()){
            cranemodel_net.chg_st(1);
        }
```

2667-ch15 10/16/02 8:46 AM Page 265


```
    } else if (msgid == "liftdown"){
        if (s3.isActive()){
            s3.chg_st(0);
        } else if (s5.isActive()){
            s5.chg_st(0);
        }
    } else if (msgid == "liftup"){
        if (s3.isActive()){
            s3.chg_st(1);
        } else if (s4.isActive()){
            s4.chg_st(0);
        }
    }
    ...
}
```

Actions

While *s4* is active, the *s4* pulse activity will lower the arm assembly's *y* property until it reaches its predefined limit. The arm assembly will move at the rate contained in the *yChgBy* property of the model object. Similarly, the *s5* pulse activity will raise the arm assembly until it reaches the limit. The code for this functionality follows.

```
function liftdownfn () {
    if (leg.arm._y > -20) {
        leg.arm._y -= cranemodel.ychgBy;
    }
}

function liftupfn () {
    if (leg.arm._y < 90) {
        leg.arm._y -=cranemodel.ychgBy;
    }
}

s4.add_pulse_activity(0, 100, "liftdownfn", this);
s5.add_pulse_activity(0, 100, "liftupfn", this);
```

Claw Articulation

The sections that follow explore the claw articulation of the device.

States and Transitions

State C contains three exclusive states that represent the claw's behavior. State *s6,* the default, represents the claw at rest. State *s7* represents the claw when it is opening, whereas state *s8* represents the claw while it is closing.

2667-ch15 10/16/02 8:46 AM Page 266

Triggers

As illustrated in the following section of the model object event handler code, a *handopen* message triggers a transition to *s7*, the claw open state, whereas a *handclose* message triggers a transition into *s8,* the claw close state. A *handstop* message triggers the transition back into *s6,* the claw rest state.

```
cranemodel.ieh = function(msgid,val){
    ...
    } else if (msgid == "handstop"){
        cranemodel_net.chg_st(2);

    } else if (msgid == "handopen"){
        if (s6.isActive()){
            s6.chg_st(0);
        } else if (s8.isActive()){
            s8.chg_st(0);
        }
    } else if (msgid == "handclose"){
        if (s6.isActive()){
            s6.chg_st(1);
        } else if (s7.isActive()){
            s7.chg_st(0);
        }
    }
}
```

Actions

Similar to the actions used to move the assembly to the left and the right, the model object uses pulse activities to move the fingers of the claw assembly. When *s7* is active, the claw will rotate each finger until it reaches its predefined limit. State *s8* rotates the fingers in the direction opposite to *s7.* The code for this functionality follows.

```
function handopenfn () {
    if (leg.arm.hand.finger1._rotation < 25){
        leg.arm.hand.finger1._rotation += 2;
        leg.arm.hand.finger2._rotation -= 2;
    }
}
function handclosefn () {
    if (leg.arm.hand.finger1._rotation > -11){
        leg.arm.hand.finger1._rotation -= 2;
        leg.arm.hand.finger2._rotation += 2;
    }
}
s7.add_pulse_activity(0, 100, "handopenfn", this);
s8.add_pulse_activity(0, 100, "handclosefn", this);
```

2667-ch15 10/16/02 8:46 AM Page 267

SUMMARY

The intent of this chapter was to introduce you to an actual device simulation using the state engine architecture described in this book. Although this example is purposefully simple, it uses the same framework as the other more complex examples. Separation of the control object from the model object may in this example seem unnecessary to some readers, but it provides a foundation from which to construct simulations in which the interface behavior is not tied directly to the model object.

This separation is especially useful when simulating abnormal conditions in which the interface says one thing while the actual device is doing another. The control object state network, although different in appearance from the model object network, provides similar functionality. In a device with only three states per substate network, each transitioning to the other, there is little advantage from one to the other. As the number of states increases, however, coding transitions from the parent state to each child state can significantly reduce the number of transitions required. This point is more thoroughly examined in Chapter 8.

2667-ch15 10/16/02 8:46 AM Page 268