## Contents

1. **Welcome to BetterState** ................................................................. 1
   1.1 Meeting Today’s Design Challenges .................................................. 1

1.2 Graphical Programming ................................................................. 2
   1.2.1 State Transition Diagrams .......................................................... 3
   1.2.2 Statecharts ............................................................................... 4
   1.2.3 Flowcharts ............................................................................... 5

1.3 BetterState ....................................................................................... 6
   1.3.1 Designing with BetterState ....................................................... 6
         Design Features ........................................................................... 6
         Design Environment ................................................................... 7
   1.3.2 Generating Code with BetterState ........................................... 10
   1.3.3 Integrating Your Code ............................................................... 12
   1.3.4 Debugging and Testing Your Code ......................................... 12
         Testing ...................................................................................... 13
         Debugging ............................................................................... 13
         Round-Trip Engineering ........................................................... 13

1.4 Summary .......................................................................................... 14
# BetterState Tutorial

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Design Requirements</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Starting BetterState</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Setting Your Project Properties</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Creating Your Chart</td>
<td>21</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Creating a New Chart for Your Model</td>
<td>22</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Setting Your Chart Properties</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Stopping and Restarting BetterState</td>
<td>25</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Saving Your Project and Stopping BetterState</td>
<td>25</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Restarting BetterState and Reopening Your Project</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>Creating the High-Level States in Your Chart</td>
<td>26</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Drawing the States in Your Model</td>
<td>26</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Assigning State Properties to Your States</td>
<td>27</td>
</tr>
<tr>
<td>2.7</td>
<td>Creating the Transitions Between States in Your Chart</td>
<td>30</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Drawing Transitions in Your Model</td>
<td>30</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Creating a New Event for Your Statechart</td>
<td>31</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Assigning Transition Properties to Your Transitions</td>
<td>32</td>
</tr>
<tr>
<td>2.8</td>
<td>Creating the Self-Looping Transitions in Your Chart</td>
<td>34</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Drawing Self-Looping Transitions in Your Model</td>
<td>35</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Assigning Transition Properties to the Self-Looping Transitions</td>
<td>35</td>
</tr>
<tr>
<td>2.9</td>
<td>Creating a Subpage in Your Chart</td>
<td>37</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Constructing a Subpage for Your Model</td>
<td>38</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Navigating from the Child Page to the Parent Page</td>
<td>39</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Navigating from the Parent Page to the Child Page</td>
<td>40</td>
</tr>
</tbody>
</table>
2.10 Creating Threads of Control on the Subpage ............................................... 40
  2.10.1 Constructing Threads of Control in Your Model ........................... 41
  2.10.2 Constructing the States and Transitions Within the Threads ...... 42

2.11 Completing Your Model .................................................................................. 43
  2.11.1 Copying Files to Your Tutorial Directory and Resaving Your Project 44
  2.11.2 Adding User Code to Your Chart .................................................... 44
       Options for Supplying User Code ...................................................... 44
       Using Functions in a Separate File ................................................... 45
  2.11.3 Setting Additional Chart Properties ............................................... 45
  2.11.4 Generating Code .............................................................................. 46
  2.11.5 Saving and Closing Your Model ...................................................... 46

2.12 Using Your Model ............................................................................................ 47
  2.12.1 Key Components of an Application ................................................ 47
       Interface to the External World ........................................................ 49
       Application Scheduler ....................................................................... 49
  2.12.2 Creating an Application Scheduler and Interface for Your Model 49

2.13 Running Chart Animation for the Model ..................................................... 50
  2.13.1 Getting the Model Running .............................................................. 50
       Getting Your Model Running ........................................................... 51
       Getting the Demo Model Running ................................................... 52
  2.13.2 Running Chart Animation in the Model ........................................ 52

2.14 Gaining Expertise in Using BetterState ....................................................... 56

Index ......................................................................................................................... 57
BetterState™ is a graphical programming tool developed to enhance your productivity when designing software. BetterState provides the two most commonly used methods of graphically designing and programming reactive systems: statecharts and flowcharts. BetterState augments these popular specification approaches with powerful utilities such as automatic code generation and chart animation.

This chapter provides a general overview of today’s design challenges in the embedded systems and object-oriented software design market followed by a discussion of graphical programming techniques. Then we introduce you to BetterState, which provides the ability to design for reactive systems with much more capability than previous techniques offer. BetterState supports your complete development cycle: design, implementation (code generation), code integration, and testing and debugging. This chapter introduces you to BetterState in terms of these phases.

1.1 Meeting Today’s Design Challenges

Modern systems are seldom isolated from their environment. On the contrary, the reason the system exists is typically to collaborate or interact with some entity or entities in its environment. Such collaboration is done by sending, receiving, recognizing, and rejecting messages and data in some form. Modern systems need
to handle reactive behavior, that is, a system that never has all its inputs ready—the inputs arrive in endless and perhaps unexpected sequences.

The embedded system industry is a major area of interest for reactive systems, also known as controllers. Application areas include the automotive industry, aerospace and military applications, consumer electronics, data communications, telecommunications, office automation, process control, and medical instrumentation. An embedded controller is often relying on software and hardware implementation to ensure its proper functioning. With increasing demand for functionality, designing embedded controller software is a critical phase in the product development life cycle.

Traditionally, software has been developed through text-based (manual) coding. More recently, graphical programming techniques have emerged to simplify this process.

### 1.2 Graphical Programming

Graphical programming is a technique that allows you to specify your application behavior graphically and have production-ready code generated automatically. It is easier, faster, and better than manual programming.

Graphical programming is easier because humans can create and understand diagrams with much greater comprehension than text-based code. Much of the documentation is the diagram itself. Thus, graphical programming is also more maintainable than text-based code.

Graphical programming is faster than manual programming because humans can draw diagrams faster than they can create text-based code that accurately represents a system. You focus on what your design is supposed to do without spending countless hours trying to decide how to textually implement your design. Furthermore, graphical programming allows for rapid prototyping of all or parts of a system, with automatic code generation performed very fast. With these advantages, iterations also happen more quickly. As a result, graphical programming translates into faster time to market.

Graphical programming is better than manual programming because automatic code generation produces uniform code, earlier detection of errors is possible through rapid prototyping, and more iterations help to ensure higher quality.
Graphical programming can include several different types of diagrammatic constructs, including:

- State transition diagrams
- Statecharts
- Flowcharts

1.2.1 State Transition Diagrams

State transition diagrams, the visual counterpart to finite state machines (FSMs), have been used to specify and design reactive systems. They are well known, well accepted, highly visual, and intuitive. Their ability to describe finite and infinite sequences, combined with their visual appeal, make FSMs one of the most commonly accepted formalisms.

State diagrams are easier to design, comprehend, modify, and document than the corresponding text-based (manual) approach. They are well suited for sequential applications.

Figure 1-1 shows a simple state transition diagram representing a three-state controller.
1.2.2 Statecharts

Statecharts are state transition diagrams extended with hierarchy, concurrence, and history states.

Let’s look at the state transition diagram in Figure 1-1 again. As designers, we recognize that the On state is somewhat more complex, so we begin to add that complexity. We know that we have two independent activities (light and radio) within the On state. We expand the On state and add that complexity: hierarchy (states within a state) and concurrence (two or more independent activities happening at the same time). We have now created a statechart, which appears in Figure 1-2.
1.2.3 Flowcharts

Flowcharts provide a graphic map of the flow of control or data through a program. Two important flowchart constructs are diamonds, which represent visual switches, and squares, which represent discrete operations. Flowchart elements are connected via conditional transitions.
1.3 BetterState

BetterState is a powerful graphical programming tool for embedded controller software modules based on state transition diagrams, statecharts, and flowcharts. The key steps in creating any project include design, implementation, integration, and testing and debugging. We introduce you to those steps with BetterState in the following sections:

- Designing with BetterState
- Generating Code with BetterState
- Integrating Your Code
- Debugging and Testing Your Code

1.3.1 Designing with BetterState

BetterState allows you to design statecharts, which have been extended to include flowchart constructs and a number of advanced features unique to BetterState.

Design Features

BetterState supports the following traditional statechart concepts:

- States
- History states
- Transitions
- Concurrence (threads of control)
- Hierarchy

Supported flowchart constructs include non-resting states and visual switches.

Advanced features of BetterState include:

- Visual synchronization
- Visual priorities
- Critical regions
- Sub-statecharts
Welcome to BetterState

- Layers
- Assertions

The latest two features, layers and assertions, are particularly useful for testing.

**Design Environment**

The primary design entry point in BetterState is the Editor window, which we refer to as the Statechart window. This is where you create your graphical programming diagrams. Figure 1-3 shows this window with a page of a statechart in it.

*Figure 1-3  Statechart Window*
The design environment provides a number of other features as well. Samples of these features appear in Figure 1-4. The Find and Replace dialog allows you to search for text that you provided in the dialogs through a statechart. The properties dialogs allow you to specify actions, conditions, comments, layers, and other properties for each type of object.

This environment is available internationally in English, Japanese, and German.
Figure 1-4  Design Environment Sample Windows and Dialogs
1.3.2 Generating Code with BetterState

BetterState provides numerous options, including optimization, for customizing your code to suit your specific design needs. These options are available through the Chart Properties dialog shown in Figure 1-5.

Figure 1-5 Chart Properties Dialog
You can generate code automatically to implement the design in one of the following languages:

- Ada
- C
- C++
- Java

You can generate code not directed to a specific operating system, or you can target the code as indicated in Table 1-1 for one of the following real-time operating systems:

- pSOSystem™
- OSEK/VDX 2.0-compliant operating system
- VxWorks®

<table>
<thead>
<tr>
<th>Language</th>
<th>pSOSystem</th>
<th>OSEK</th>
<th>VxWorks</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>C++</td>
<td>✓</td>
<td>N/A</td>
<td>✓</td>
</tr>
</tbody>
</table>

You can also choose between event-driven and procedural code:

- Event-driven code. Event-driven systems are used when each input to your system can be associated with an event. Events can come from anywhere. When events occur, they are handled by some type of event handler. The event handler then notifies the application scheduler that an event has occurred. The application scheduler calls a function that is designed to handle that event for all transitions in the statechart or the project that are labeled with that event. Thus, the flow of control is grouped by events.

- Procedural code. In a procedure-driven statechart, typically an application scheduler calls the controller’s code in some type of loop. The controller examines conditions, changes states, and executes actions. No events are described for such a system.

Once you have set up the code generation options, generating code is easy: you just click the Generate Code toolbar button.
Highlights of the code generated are as follows:

- **Ada code.** BetterState generates one source file for every statechart in your project for both procedural and event-driven code. In addition, with event-driven code, BetterState generates one procedure per event that is labeled in the statechart.

- **C code.** If you use procedural style, you get one function per statechart. If you use event-driven code, you get one function per event labeled in the statechart; you can use the same event multiple times if appropriate. Transitions without events are collected into a single procedure.

- **C++ code.** If you use procedural style, you get one class that contains one method per statechart; if you use event-driven code, you get one class that contains one method per event labeled in the statechart.

- **Java code.** If you use procedural style, you get one class that contains one method per statechart; if you use event-driven code, you get one class that contains one method per event labeled in the statechart.

- **pSOSystem code.** BetterState can generate C or C++ code with calls designed specifically for pSOSystem.

- **OSEK code.** BetterState can generate C code with calls designed specifically for OSEK/VDX v2.0 compliant operating systems.

- **VxWorks code.** BetterState can generate C or C++ code with calls designed specifically for VxWorks.

### 1.3.3 Integrating Your Code

When your code is generated, you integrate it with the remainder of your application. Typically, you provide an application scheduler that calls the generated code. The application scheduler also provides the data input to the application. The application scheduler, for example, might be a unit test harness, or it might be your final, overall application.

### 1.3.4 Debugging and Testing Your Code

We designed BetterState with the entire development cycle in mind. Therefore, BetterState contains a number of features that are specific to both testing and debugging.
Testing

When you are planning for and doing your testing, you might want to make use of the following features:

- **Layers** You can partition the states in your statechart into any number of layers of your choice. At code generation, you determine what layers to include in your generated code. For example, you can create specific layers that you only use for testing.

- **Assertions.** BetterState allows you to specify both statechart and state assertions. You use assertions primarily to insert testing code within your statechart. Assertions can be turned off at code generation.

Debugging

You can debug your design graphically using BetterState itself, or you can debug textually using the debugger for the language you are using.

BetterState provides a means of debugging in your statecharts called *chart animation*. Chart animation automatically highlights BetterState states as your program executes. Features of chart animation include:

- Current and previous state animation
- Path animation
- Break points

Round-Trip Engineering

When you find a problem in any code that you supplied—conditions, actions, and so forth—you can correct that code in a text or code editor. Later, you can synchronize your statechart with the changes you manually performed while textually debugging.
1.4 Summary

As you work with BetterState, you will see that BetterState makes module generation for embedded system applications easier, faster, and better than using more traditional methods (manual code generation). BetterState supports all phases of project development: design, implementation, integration, and testing. And, you can use BetterState in your current tool framework and development process.
This tutorial contains the Design Requirements and then each of the major tasks to start BetterState and construct a model that implements the design requirements:

- Starting BetterState
- Setting Your Project Properties
- Creating Your Chart
- Stopping and Restarting BetterState
- Creating the High-Level States in Your Chart
- Creating the Transitions Between States in Your Chart
- Creating the Self-Looping Transitions in Your Chart
- Creating a Subpage in Your Chart
- Creating Threads of Control on the Subpage

Completing Your Model

Using Your Model is a conceptual section designed to help you understand how you can use your BetterState models in a complete application. Running Chart Animation for the Model shows you how to run chart animation on your model and what it means.

Finally, Gaining Expertise in Using BetterState points you to additional resources for learning more about BetterState.
2.1 Design Requirements

For your first statechart, we have chosen a digital clock. Everyone is familiar with some version of this system; therefore it is easy to understand.

Our digital clock has an LCD that displays either the time in hours and minutes or the date in month (1–12) and day (1–31). The clock operates in two modes: display or set. The clock has two buttons: Date and Set.

The normal display is time; in display mode, the colon between the hour and day flashes at one-second intervals. The hours go from 1–12 with a tilde (~) indicating afternoon hours. While the Date button is being held down, the LCD displays the date; in this case, the month and day appear with no colon, and nothing flashes.

Using the Set button enables you to set the date and the time. Each time you press the Set button, the clock moves from one function to another in the following order: display mode, set month, set day, set hour, set minute, and then back to display mode. When in the set mode, the quantity being set flashes at one-second intervals. If you want to reset the quantity that is flashing, press the Date button to increment the digit.

Figure 2-1 shows an implementation of the user interface to this clock in display mode.

Figure 2-1 Clock User Interface

You are going to create the statechart that controls a digital clock. The buttons and display of the user interface have already been created for you.
2.2 Starting BetterState

We assume that you have either installed BetterState or BetterState Lite on your computer. (See the BetterState Installation Guide for installation information for BetterState; see the Wind River website at http://www.windriver.com to download BetterState Lite.) In this section we tell you how to start BetterState and BetterState Lite.

**NOTE:** The difference between the functionality of BetterState and BetterState Lite is the number of states that you can create and for which you can generate code; you can create only eight states with BetterState Lite. However, you can load and run a bigger existing model. BetterState Lite is designed to introduce you to BetterState and to help you evaluate BetterState. This tutorial works on both products.

To start BetterState:

Select Start→Programs→BetterState xx.x→BetterState or BetterState Lite.

The BetterState window appears with a default project (see Figure 2-2).

Figure 2-2 BetterState Window with Default Project
2.3 Setting Your Project Properties

A BetterState project consists of one or more statecharts, also referred to as charts or controllers. Each chart, which represents a controller in the project, can have multiple pages. A page, also referred to as a frame or window, is where you create graphic objects and text. You view each page of a chart in the Statechart window. BetterState allows you to create objects and text, as well as to modify them, as you see fit.

Before starting to work with your model directly, you first need to set up project properties, which you do from the BetterState window. Although you can change any properties at any time, it is most convenient to set project properties at the beginning of the project.

Project properties consist of a description, visual properties, and the locale. These properties affect the appearance of your project, not its performance. Of the various object types in BetterState, our model uses states, threads, and transitions; therefore, you only set visual properties related to those object types. You can skip this section entirely, but your diagrams will look different from the ones in this tutorial.

To set up your project properties:

1. Make the BetterState window (see Figure 2-2, p.17) the active window.
2. From the BetterState window, select Edit→Project Properties.
   The Description tab of the Project Properties dialog comes on view.
3. Enter ClockEventDrivenDemo in the Project Name field.
4. Click the Visual Settings tab (see below).

This tab allows you to change the visual settings for all types of objects in BetterState. All objects have initial default visual settings, and you can use these values, or you can change the default settings at any time. Additionally, you can change visual settings for objects after they are drawn using a different mechanism (see the BetterState User’s Guide).

Notice that the Use UML symbols checkbox (Unified Modeling Language symbols used) is enabled, which tells BetterState to use Unified Modeling Language symbols.
5. From the dropdown menu, select States, and then click Modify defaults. The Text tab of the Visual Settings dialog comes on view.

a. With State Name selected, click SansSerif; ensure that the font size is 12, the font is Plain, and that the Visible checkbox is enabled; and accept the remaining defaults.
b. With State On-entry Action selected, click SansSerif; ensure that the Visible checkbox is enabled; and accept the remaining defaults.

c. Accept the remaining defaults, and click OK.
   This action takes you back to the Visual Settings tab of the Project Properties dialog.

6. From the dropdown menu, select Threads, and then click Modify defaults.
   a. With Thread Name selected, click SansSerif; ensure that the Visible checkbox is enabled; and accept the remaining defaults.
   b. Accept the remaining defaults, and click OK.
      This action takes you back to the Visual Settings tab of the Project Properties dialog.

7. From the dropdown menu, select Transitions, and then click Modify defaults.
   a. With Transition Event Name selected, click SansSerif, and ensure that the Visible checkbox is enabled. In the Color box, click Modify, and change the color to black. Accept the remaining defaults.
   b. With Transition Action selected, click SansSerif, and ensure that the Visible checkbox is enabled. In the Color box, click Modify, and change the color to black. Accept the remaining defaults.
   c. With Transition Priority selected, disable the Visible checkbox.
   d. Click the Path tab, and notice that the default setting for path type is Vertical elbow; accept the default.
   e. Click OK to accept the remaining defaults.

8. Click OK in the Project Properties dialog to accept all changes you made in this dialog.

2.4 Creating Your Chart

Statecharts, also known as charts, are the basic units for which code is generated. When you use the default project, as you are doing for this tutorial, you have to create the charts for the project.
2.4.1 Creating a New Chart for Your Model

To create a new chart:

Select File → New → New Chart in the BetterState window.

The Statechart window appears with a new, untitled statechart named Untitled_1 (see Figure 2-3).

Figure 2-3  Statechart Window with New, Untitled Statechart

2.4.2 Setting Your Chart Properties

Chart properties determine what options are considered in code generation. BetterState allows you to set up or change your chart properties at any point. All properties have defaults; therefore, you can generate code without changing any properties. However, some properties influence the interface that you can use in BetterState, and your coding language, in particular, determines how you should write the user-code portions of your chart. You set all chart properties from the Chart Properties dialog.

At a minimum, you should set the language for code generation, the control implementation, the type of priorities, and the conditionals at the start of a statechart. You are also going to give the chart a name while you are setting the properties.
Examining the specifications of the model that you are creating (see 2.1 Design Requirements), you can see that two buttons and a timer control the behavior of the clock. Furthermore, these buttons act independently at any given time even though the Date button performs different functions at different times. Although this model can be implemented with procedural code, it is ideally suited for an event-driven system, and this tutorial uses that control implementation.

To set the initial properties:

1. Make the untitled Statechart window (see Figure 2-3) the active window.
2. Click the Chart Properties toolbar button (see picture in left margin) to bring up the Chart Properties dialog (see Figure 2-4).
   The toolbar buttons have tooltips to guide your usage.
3. On the General view, enter **clock** in the Chart Name field.
   This name becomes part of your code; the name must conform to the naming requirements of the coding language you select. In this example, you must use the names that we suggest or you will have problems later in the tutorial.
4. Click Code Generation in the Chart Properties tree (left pane). Notice that the default language is C, which you are going to use because it is more universal.

5. Click Settings in the tree. Note that the default Control implementation is Procedural.

6. Enable Event-Driven to change the control implementation to better suit our model.

For a short description of procedural and event-driven statecharts, see *Generating Code with BetterState*, p.10.
7. Under Priorities, Enable None.
   Note the default setting for the conditionals (If - Then).
8. Click OK to accept these settings, including the remaining defaults, and dismiss the dialog.

2.5 Stopping and Restarting BetterState

You can stop and restart BetterState at any time and pick up the tutorial at any point, provided that you save your project immediately prior to exiting. We have you do that now so that you know how.

2.5.1 Saving Your Project and Stopping BetterState

To save your project and exit BetterState:
1. From the BetterState window, save your project by selecting File→Save Project As.
   Hereafter, use File→Save Project. Save your model periodically.
2. Create a directory called bestTut1 in which to save your tutorial, and enter ClockEventDrivenDemo in the File name field.
   These menu items work in the standard Save/Save As fashion for Windows® operating systems.

2.5.2 Restarting BetterState and Reopening Your Project

To restart BetterState and reopen your project file:
1. Select Start→Programs→BetterState xx.x→BetterState or BetterState Lite.
   The BetterState window appears.
2. Click File→Reopen→ClockEventDrivenDemo.
   BetterState reloads your project exactly as you saved it.
3. Under Contents of Project in the BetterState window, double-click the statechart name clock to bring up the Statechart window.

2.6 Creating the High-Level States in Your Chart

BetterState allows you to group sets of states together using hierarchy. In BetterState, we represent hierarchy by placing states within states. We call the outside states higher-level states and the inside states, lower-level states or substates.

The specifications indicate that you have five functions to perform:

- Clock display
- Setting the month
- Setting the day
- Setting the hour
- Setting the minute

Furthermore, the functions are performed in a defined order with the clock returning to the display function at the end of the fifth function. These functions can be represented by five states that progress from one to the other in an endless loop; these five states are the high-level states in this model. In this section you draw these states and then assign state properties to them.

2.6.1 Drawing the States in Your Model

States, represented by boxes with rounded corners, are the most basic building blocks in BetterState.

NOTE: When we refer to a state without qualification, we are referring to a resting state. BetterState provides specialized types of states, such as non-resting and history states, but we always refer to them with the qualifier.
To draw the high-level states on the parent page of your model:

1. With the Statechart window on view, click the State toolbar button.

2. Place the cursor in the work area where you want one corner of the state.
   Place the first state near the top middle of the window.

3. Click and hold down MB1 (left mouse button for right-handed definition) while dragging the mouse to make the state as large as you want.
   When sizing your states, keep in mind that this main diagram has five high-level states.

4. Release MB1 when the state is the size you want it to be.
   Small boxes called selection handles, appear on the state you created. These handles indicate that this state is the current object. BetterState allows you to resize, move, copy, or delete the current object using standard graphics techniques.

5. Using the technique outlined above, draw a total of five states in a circular arrangement.

### 2.6.2 Assigning State Properties to Your States

You assign all properties to a state on the State Properties dialog. From the State Properties dialog, you can assign the state a name; provide qualifications to the type of state (history, non-resting, and so forth); and assign various types of actions, assertions, comments, and the layer where you want the state to reside.

For purposes of this tutorial, you use a small subset of the total properties:

- Though not required, you give each state a name. The names must conform to the naming requirements for your code generation language.

- Every statechart and set of hierarchical states must have a default state, the initial state visited in a statechart or anywhere within a state hierarchy where there is a choice of the next state. You make a state a default state by enabling its Default checkbox.
An on-entry action is code that is executed when a controller enters the state. You supply on-entry actions on the On-Entry Action tab of the dialog; remember that this is code and you need to supply appropriate syntax for your programming language.

To assign state properties to the five states of the clock statechart:

1. Ensure that your cursor is in select mode by clicking the Select Mode toolbar button.
   
   The cursor appears as a white, outlined arrowhead (↑).

2. To bring up the State Properties dialog, double-click the state to which you want to assign properties.

   The State Properties dialog for that state comes on view.
3. Using the State Properties dialog for each of these five states, give each one a name listed in the table below, remembering that the order is important. While you have the dialog open, click the Actions tab, click the Edit On-Entry Action button, and add the corresponding on-entry action:

<table>
<thead>
<tr>
<th>State Name</th>
<th>On-Entry Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>clockmode();</td>
</tr>
<tr>
<td>MonthSet</td>
<td>displayMonth();</td>
</tr>
<tr>
<td>DaySet</td>
<td>displayDay();</td>
</tr>
<tr>
<td>HourSet</td>
<td>displayHour();</td>
</tr>
<tr>
<td>MinuteSet</td>
<td>displayMinute();</td>
</tr>
</tbody>
</table>

These on-entry actions are user code in the form of function calls with C syntax.

4. Using the State Properties for the clock state, enable the Default checkbox to make it the default state.

Your diagram should look something like Figure 2-5.

Figure 2-5  clock Statechart with Five States
2.7 Creating the Transitions Between States in Your Chart

Examining the specifications of the model that you are creating (see 2.1 Design Requirements, p.16), five states progress or transition from one to the other in an endless loop. The event that causes this progression is pressing the Set button.

In this section, you draw the transitions between the high-level states in your model and then assign their properties.

2.7.1 Drawing Transitions in Your Model

A transition is a graphic link that enables a controller to go from state to state. A transition is represented by a line from the source object with an arrowhead pointing to the target object.

Within BetterState, you have a choice of three transition styles, which you select from the Path tab of the Visual Settings dialog. The vertical elbow style is the default, and you use it in this tutorial.

Once drawn, a transition stays attached even as you resize and move the objects.

To draw the transitions between the five high-level states in your model:

1. Click the Transition toolbar button.

2. Place the cursor inside the state (can be a different type object) that is your source state.

   The first state is clock.

   The cursor can be anywhere inside the object, but it must be inside the state and not just close to the border of the state.

3. Click and hold down MB1, drag the mouse to your target object (usually a state), and release it on the target object.

   The target of state clock is state MonthSet.

   Transitions can connect states, connector states, sub-statecharts, visual switch polygons, and transition connectors. BetterState controls the initial attachment points of transitions to objects.

4. Create a transition between each of the states, keeping the following order:

   clock→MonthSet→DaySet→HourSet→MinuteSet→clock
2.7.2 Creating a New Event for Your Statechart

Before you can assign transition properties to the transitions that you just created, you must create the event(s) that you need in the Data Dictionary, the central location where all variables, arguments, and events are defined.

To create a new event in the Data Dictionary:

1. From the Statechart window, click File→Data Dictionary.
   The Data Dictionary for the chart clock comes on view. At this point, it contains no data.

2. Click the Arguments tab and then the Events tab.
3. Ensure that the event field is blank. Use the dropdown menu, and position it on the blank event if one exists in this field.
   This is the case after the first event is defined for each statechart.
4. Type **SetPressed** in the event field, and press **Return**.

Notice that the argument radio buttons become active following this definition. You use the default chart arguments.

![Data Dictionary window](image)

5. From the Data Dictionary window, click **File→Close** to close the Data Dictionary.

You add more items to the Data Dictionary as you need them.

### 2.7.3 Assigning Transition Properties to Your Transitions

You assign all properties to a transition on the Transition Properties dialog.

Since this is an event-driven controller, you can assign an event to the transition; you can also assign a condition, action, and comments for any type of control implementation. For purposes of this tutorial, key points are:

- You should define an **event** or a **condition** that causes a transition to occur; you can use both, but that is not necessary.
- If you want to provide a **condition** for the transition, provide the Boolean expression for that condition on the Condition tab. This tutorial does not use conditions.
- If you want an **action** to take place as the transition occurs, provide the code for that action on the Action tab.
To assign properties to the major transitions in your model:

1. Put your cursor in select mode by clicking the Select Mode toolbar button. The cursor appears as a white, outlined arrowhead (🔗).

2. To bring up the Transition Properties dialog, double-click the transition for which you want to assign properties.

3. Using the Transition Properties, assign the SetPressed event to each of these transitions.

   When a user presses the Set button, the controller progresses from one state to the other. Therefore, the same event (SetPressed) controls each of these transitions.

   a. For the first transition, examine the dropdown menu for Event name, and verify that SetPressed is listed.

      You must define events in the Data Dictionary before you can use them in the Transition Properties dialog (see 2.7.2 Creating a New Event for Your Statechart).

   b. Select SetPressed from the dropdown menu, and click OK.

   c. For each of the other transitions, select SetPressed from the dropdown menu.

   Your diagram should now contain the same constructs as Figure 2-6.
2.8 Creating the Self-Looping Transitions in Your Chart

Four states set some quantity—month, day, hour, or minute. Each increments the current value when a user presses the Date button. However, each state stays in the same state. You can represent this behavior by a transition from the state and then back to the same state; we call these self-looping transitions. Their purpose is to perform some action while remaining in a given state.

In this section, you draw self-looping transitions for the parent page of your model and then assign the transition properties to them.
2.8.1 Drawing Self-Looping Transitions in Your Model

To draw the self-looping transitions in the model:

1. Click the Transition toolbar button.
2. Place the cursor inside the state that is your source object.
   For the first source state, choose MonthSet.
3. Click and hold down MB1, drag the mouse outside the state and then back into it again, and then release the mouse inside the same state.
   Hint: Placing the transition to the left of the state enables you to see the event better on the diagram.
4. Put your cursor in select mode by clicking the Select Mode toolbar button.
   The cursor appears as a white, outlined arrowhead (стрелка).
5. Click the transition to select it.
6. Press and hold down MB1 with the cursor on either of the handles attached to the state, and drag the handle to the desired location; repeat for the other handle as necessary.
7. Press and hold down MB1 with the cursor pointing to the middle (horizontal) handle on the transition, drag the transition to the size and location desired, and then release MB1.
   You can change the size and shape of the transition as needed when you add events and actions to your transitions.
8. Draw a self-looping transition from each of the four “set” states (MonthSet, DaySet, HourSet, MinuteSet).

2.8.2 Assigning Transition Properties to the Self-Looping Transitions

Looking back at the design requirements again (see p.16), each of the four set states allows the user to increment its quantity: month, day, hour, minute. Therefore, each of the four set states needs a self-looping transition drawn around it with the appropriate function to increment its basic quantity.

To assign transition properties to the self-looping transitions in your model:

1. Put your cursor in select mode by clicking the Select Mode toolbar button.
   The cursor appears as a white, outlined arrowhead (стрелка).
2. To bring up the Transition Properties dialog, double-click the transition for which you want to assign properties.

3. Using the procedure provided in 2.7.2 Creating a New Event for Your Statechart, add the event `DatePressed` to the Data Dictionary.

4. Using the Event tab of the transition properties, assign the event `DatePressed` to each of these transitions.

   When a user presses the Date button, the quantity in that state is incremented. Therefore, you also need an action in each of these transitions.

5. Using the Action tab of the Transition Properties, enter the action for each of these transitions as follows:

<table>
<thead>
<tr>
<th>State Name</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>MonthSet</td>
<td><code>incrementMonth();</code></td>
</tr>
<tr>
<td>DaySet</td>
<td><code>incrementDay();</code></td>
</tr>
<tr>
<td>HourSet</td>
<td><code>incrementHour();</code></td>
</tr>
<tr>
<td>MinuteSet</td>
<td><code>incrementMinute();</code></td>
</tr>
</tbody>
</table>

These transition actions are user code in the form of function calls with C syntax.

Your diagram should now look something like Figure 2-7.
2.9 Creating a Subpage in Your Chart

When you take a look at the design requirements again, you find that the clock displays either the time or the date in both the display mode and the set mode. Without going into detail at this point, it is reasonable to assume that the clock state needs to be something other than a simple state. You have used the space available in the Statechart window reasonably effectively, and it contains about as much information as you want to put in one graphical representation; therefore, you can use the subpage mechanism to contain the complexity of the clock state.

In this section, you construct a subpage for your model and learn how to navigate between parent and child pages.
2.9.1 Constructing a Subpage for Your Model

To construct the child page of the clock statechart:

Select state clock, and then click the Make subpage toolbar button.

BetterState creates a subpage (or child page) for the contents of the higher-level state. State clock becomes a coarse state on the parent page of the display (delineated with a parallel-line fill pattern). The subpage contains the lower-level states, if any, that originally appeared on the initial page of the statechart; in this case, it is initially empty. The entire Statechart window is available for the contents of the coarse state.

You can create a coarse state before you create your state hierarchy as well as afterwards. If a state is empty and you make a coarse state out of it, then you can add state hierarchy on the subpage. You employ the latter technique in this tutorial; you provide the contents of this state in the next section.

When you use the Make subpage command, the subpage comes on view (see Figure 2-8). BetterState gives the subpage the name Contents of: parent state.

Figure 2-8 Contents of clock Subpage Immediately After Creation
2.9.2 Navigating from the Child Page to the Parent Page

We talked about creating a coarse state on the parent page, so let’s go back and look at it.

To make the parent page the active window from a child page:

- Click the Go to Parent toolbar button.

BetterState brings the parent page to the front and makes it the active window.

The completed main diagram looks something like Figure 2-9.

Figure 2-9  Completed Main Page of clock Statechart with clock as Coarse State
2.9.3 Navigating from the Parent Page to the Child Page

Now that you’ve looked at the parent page, you need to go back to the child page and complete the contents of the clock state.

To make the child page the active window:
1. On the parent page, select the desired coarse state clock.
2. Click the Go to Child toolbar button.

BetterState brings the child page to the front and makes it the active window.

2.10 Creating Threads of Control on the Subpage

When the clock is in display mode, it either displays the time or the date. Therefore, the state that represents the display mode contains two other states: one for displaying time and one for displaying the date. Looking at the model more carefully, it becomes obvious that you have to have some type of timer in the clock in order to have a clock at all.

Since the timer is a fundamental part of the clock’s normal operation and has nothing to do with setting the clock, you can think of it as an independent but parallel function of the display mode. The timer must provide information that you use in the display mode, but otherwise these functions are separate.

BetterState provides threads of control for concurrent and relatively independent paths within a state. If you think of our primary state as the clock rather than the display function, then you can have the display mode as one thread and the timer in a second thread. Implementing the model in this way maintains the progression of functions controlled by the Set button that you discussed above.

In this section, you create threads within your model and then assign properties to the threads.
2.10.1 Constructing Threads of Control in Your Model

To construct two threads in the clock state:

1. With the subpage on view in the active window, click anywhere in the window that represents the state clock.

2. Click the Concurrence toolbar button.

If no threads existed previously, two threads of control appear inside the reference state; if threads already existed, one additional thread appears. BetterState represents threads of control with dashed lines. You can select threads, resize them, and move them like other objects. You can also name them.

3. Resize the threads so that each occupies approximately half the diagram:
   a. Click in one of the threads to select it.
   b. Using the handles of the thread, resize it.
   c. Repeat for the other thread.

The subpage now has the appearance shown below.

4. Double-click a thread to bring up the Thread Properties dialog, and name one thread DisplayThread; using the same procedure, name the other TimerThread.
2.10.2 Constructing the States and Transitions Within the Threads

When the clock is in display mode, it either displays the time or the date. Therefore, the state that represents the display mode contains two other states: one for displaying time and one for displaying the date. The normal (default state) is to display the time. The event that causes the clock to display the date is pressing the Date button; the event that causes the clock to again display the time is releasing the Date button. This information provides what you need for DisplayThread.

The timer goes into TimerThread. The lowest unit that you increment in the clock is a minute. Therefore, you need to know when to increment the minute on the display. You can represent this logic with a single state and a self-looping transition that increments the time once per minute. You can increment every other displayed number with logic.

In this section, you construct the states and transitions described above. If you need help with any of the techniques, look back to previous sections where you used them.

To construct the states and transitions in the threads for your model:

1. Create two states within thread DisplayThread, and name them DisplayTime and DisplayDate (see 2.6 Creating the High-Level States in Your Chart, p.26). Make DisplayTime the default state.

The display transitions from displaying the time to displaying the date when the Date button is pressed, and it transitions from displaying the date when the Date button is released.

2. Using the procedure provided in 2.7.2 Creating a New Event for Your Statechart, add the events DatePressed, DateReleased, and Timer to the Data Dictionary.

3. Create a transition from the state DisplayTime to DisplayDate, and label the transition with the event DatePressed (see 2.7 Creating the Transitions Between States in Your Chart, p.30).

4. Create a transition from the state DisplayDate to DisplayTime, and label the transition with the event DateReleased.

Within the timer thread, you can create a timer with one state. Every minute the controller exits the state, increments the time by one minute, and reenters the state.

5. Within the TimerThread, create a default state, and name it OneMinuteWait.
6. Create a self-looping transition from/to state OneMinuteWait (see 2.8 Creating the Self-Looping Transitions in Your Chart, p.34).

7. Label the self-looping transition with the event Timer and the action incrementTime().

At this point, the subpage of your diagram should look similar to the diagram below.

8. Click the Go to Parent toolbar button to return to the parent page of the diagram.

9. From the BetterState window, click File→Save Project to save your model.

### 2.11 Completing Your Model

At this point, you have completed the drawing of your model, and it is loaded in BetterState. You still have to add user code to your chart and generate code for your chart. In this section, you will perform these tasks, as well as some preparatory and cleanup tasks.
2.11.1 Copying Files to Your Tutorial Directory and Resaving Your Project

For purposes of this tutorial, you are going to use parts of the application created at Wind River. As you work through the next two major sections, you will learn about these parts of the model. For now, we ask you to copy a set of files into your tutorial directory and then resave your tutorial model.

To copy clock demo files into your user directory and resave your project:

1. Copy the contents of \isihome\bs_6xx.x\examples\clockeventdrivendemo to your_project_directory\bestTut1, where \isihome is the root installation directory for BetterState.

2. Rename EventDrivenClockDemo.bzp, the demo version of the project, if you think you might want to look at it.

3. Save your project again immediately.

   When you copied the demo files to your current directory, you overwrote your project file, EventDrivenClockDemo.bzp. Resaving the model corrects this situation.

2.11.2 Adding User Code to Your Chart

In this section, you learn what options are available for supplying user code and the option that you are going to use for this tutorial.

Options for Supplying User Code

We want to introduce you to the options that BetterState provides for supplying user code:

- Supply all code required for actions on the action tabs of the State Properties and Transition Properties dialogs.

- Use function calls for the actions, supply the code for the functions in a separate file, and then link the code together.

For this tutorial, we are using the last method listed above, but we want to emphasize that this is a choice that you make for every action. The only important point is that you have to supply the code and make it available to the controller.
Using Functions in a Separate File

You specified a number of functions for both state and transition actions in the clock statechart. If you were going to have you complete this model in its entirety, you would have to supply the code for these functions. However, we know that you can write a set of these functions that work, and this tutorial is about learning how to use BetterState—it is not about writing C functions. Therefore, we do not discuss the details of these functions further.

For this tutorial, we suggest that you ignore the content of these functions, and take it on faith that we can all create functions that satisfy the requirements of a digital clock. If you want to examine the set of functions that we have written for this model, you can find these functions in your_project_directory\bestTut1\clockcontrol.c.

2.11.3 Setting Additional Chart Properties

You can set or change any chart property at any time. We have left these final chart properties to be set just prior to code generation.

To set other chart properties:

1. Click the Chart Properties toolbar button to bring up the Chart Properties dialog.
2. Click Debug in the chart tree, and enable Include code for Chart Animation.
3. Click Include Files (under User Code) in the tree.
4. In the lower section for Imports/includes for generated header file (.h), enter the following two include statements:
   ```c
   #include "altia.h"
   #include "clockcontrol.h"
   ```
5. Click Output in the tree.
   a. Enter the directory your_project_directory\bestTut1 for the location of the code to be generated.
   b. Enter a filename (clock).
      Do not include an extension for this file; BetterState does this.
   c. Leave the Create separate header file checkbox enabled.
You might want to glance through the remaining chart properties at this time by clicking each view of the tree just to see what is available.

### 2.11.4 Generating Code

To generate code for your statechart:

- Click the **Generate Code for Chart** toolbar button.

  BetterState generates code for your statechart and places `clock.c` and `clock.h` in `your_project_directory/bestTut1`, which you specified on the Output view of the Chart Properties dialog, provided there are no errors in your chart that prevent code generation.

  If you have errors, the Error and Warning Browser appears with the message(s); when you double-click a message, BetterState attempts to highlight the region of your diagram in which the error occurred. If you need additional help understanding the messages, they are documented in the BetterState User's Guide.

**NOTE:** If you do not get your model correct so that BetterState can generate code for it, you can still complete the tutorial; the procedures provided use the demonstration model that you copied into your directory.

### 2.11.5 Saving and Closing Your Model

When you complete this process, we recommend that you save and close your model. It is not necessary to close BetterState.

To save and close your model:

1. From the BetterState window, select `File→Save Project`.

   You saved your project previously as `your_project_directory/bestTut1/ClockEventDrivenDemo.bzp`.

2. From the BetterState window, select `File→Close→Close Project`.

   This leaves BetterState running with no project open, which is the desired state for completing this tutorial.
2.12 Using Your Model

At this point, you have drawn your controller, supplied the user code that is part of the controller, and generated code for the controller. Many people think that the job is done at this point, and in many cases, the majority of the work is done. However, one or more controllers does not make a complete system.

In this section, we show you the minimum required to use the code generated by BetterState. We hope that you use this information as a starting point in understanding what BetterState can do for you without letting it limit your imagination in terms of other ways that you might use this code.

2.12.1 Key Components of an Application

The key components of an application are represented in Figure 2-10.
The interface to the external world and application scheduler are your responsibility. In the following section, we provide some tips regarding these functions for your conceptual understanding of the process. Then we tell you how we deal with these components in this tutorial.
**Interface to the External World**

The interface to the external world might provide a user interface and almost always provides some hardware components. For example, our clock example contains a timer, and a traffic light controller depends upon the arrival of vehicles in each of the directions controlled by the light. This interface generates events to which you want the controller to respond. In the examples given, the event might be the pressing of a button, the completion of some given time frame, or the arrival of a particular number of vehicles at a traffic light.

You must set up some communication system between the interface and the application scheduler for the interface to notify the application scheduler when events occur.

**Application Scheduler**

Typically, the application scheduler is where most of the run time is spent. It ties the interface to the controller. The interface creates events. The application scheduler waits for events and sends them to the controller when they occur. The controller then sends changes to the interface.

Example 2-1 provides pseudo code for the main program of an application scheduler that you could use for the model that you created.

Example 2-1  
**Pseudo Code for an Application Scheduler (main Only)**

```c
/* Declarations */

void main();
{
    /* Establish communication with the interface */
    /* Register a function with the interface for each event */
    /* Release control to the interface */
}
```

**2.12.2 Creating an Application Scheduler and Interface for Your Model**

As you can see, our application contains an interface to the external world, an application scheduler, and a controller. This tutorial does not walk you through the steps of creating an application scheduler and an interface. Instead, you use these components from the demo model for running chart animation in the next section.
At this point, we refer you to the application scheduler for this model:

`your_project_directory\bestTut1\clockmain.c`

### 2.13 Running Chart Animation for the Model

In this section, we show you how to use the chart animation feature. Chart animation is one of the debugging tools that BetterState provides. The chart animation feature allows code generated by BetterState running in a separate process to communicate with BetterState. You can observe states in transition in the Statechart window as your code is running.

In the earlier parts of this tutorial, you built a model of the digital clock. We discussed using the model in terms of creating an interface and an application scheduler to put it all together. Because this tutorial is not about writing code or forcing you to use any particular type of interface, we supply these portions for you. The files you copied to your `bestTut1` directory contain these components.

#### 2.13.1 Getting the Model Running

In 2.11.5 Saving and Closing Your Model, p.46, you closed your project and left BetterState running. If you only have BetterState Lite, then you must have it running (see 2.2 Starting BetterState, p.17 if necessary). If you have the full BetterState product, it is not necessary that the product be running, but you also don’t need to close it.

We provide two procedures in this section: one for getting your model running and one for getting the demo model running. If you were unable to generate code for your model, then go directly to the demo model. If you have trouble getting your model running in this section, then go to the demo model.
Getting Your Model Running

To run your model:

1. Open the file ClockEventDrivenDemo.bat in a text editor.
   You copied this file to your working directory, your_project_directory\bestTut1.

2. Look for the following line:
   
   ```
   set DEMODIR=%BEST%\examples\%PROJNAME%
   ```
   and change it to reflect the full pathname of your working directory, your_project_directory\bestTut1. For example,
   
   ```
   set DEMODIR=C:\BetterState\BS6_Projects\bestTut1
   ```

3. Save the batch file.

4. Select Start→Programs→Command Prompt to bring up the Command Prompt window.

5. Set the directory to your current working directory, your_project_directory\bestTut1. For example,
   
   ```
   cd C:\BetterState\BS6_Projects\bestTut1
   ```

6. Run the makefile for your model by issuing the following command in the Command Prompt window:
   
   ```
   ClockEventDrivenDemo.bat -make
   ```
   This command creates a new subdirectory Release and puts a new clock.exe in it. The original clock.exe in the current directory is not overwritten.

7. Run your model by issuing the following command:
   
   ```
   ClockEventDrivenDemo.bat -run
   ```
   This command executes the new clock.exe in the Release subdirectory.
Getting the Demo Model Running

Remember that you want to have BetterState running with no project loaded.

To run the demo model:

Select Start→Programs→BetterState xx.x→Demos→Clock (event driven).

If you do not have BetterState running, the batch file (clockeventdrivendemo.bat) launches BetterState. It loads the model and executes the code for the interface and the chart; these interact with BetterState itself to produce chart animation. You see the clock interface in one window (see Figure 2-11) and the chart in another window on your terminal, as well as a Command Prompt window.

NOTE: The visual settings for this chart are different than we instructed you to create for your model. These settings, however, do not affect the functionality of the model.

Figure 2-11  Digital Clock Interface Provided for the Event-Driven Clock Demo

2.13.2 Running Chart Animation in the Model

Initially the main page of chart clock comes on view with the clock state shown in red. Additionally, the red, circled bar symbol ( ) appears in this state. The red state indicates that the state is being currently visited. The icon in the corner means that this is the state previously visited (see Figure 2-12).
Each time the statechart is called, the present state, previous state, and last transition are updated. This statechart is called periodically once per minute and for the following events:

- DatePressed
- DateReleased
- SetPressed

We suggest that you treat this model just like you would a new digital clock. However, you are not just the user; you are also the designer, so you want to have both the interface window and the animation window on view together and observe what is happening in the animation window. Click the Set button in the interface to move from one state to another. Click the Date button to set the number in question.

When you press the Set button the first time, the current state becomes MonthSet. MonthSet and the transition from clock to MonthSet are shown in red (see Figure 2-13); showing the transition in red indicates that it was traversed last. The icon remains in clock to indicate that it is the previous state.
If you remain in the MonthSet state for a few seconds without touching the interface, you will notice that the previous state changes to MonthSet, and the transition from clock to MonthSet turns black again. These changes are caused by the call to the statechart that occurs every minute.

In the interface window, notice that the first number is blinking, and the colon disappears. Click the Date button repeatedly until you reach the current month. If you watch the chart carefully while you are doing this, you can see the transition from MonthSet to itself flash red quickly. The icon then moves to state MonthSet.

If you press and hold down the Date button, the transition around MonthSet turns red; it stays red until you release the button, which initiates another call to the statechart. Hence, the flash when you click the button.

You can move around the states on the main page and see similar behavior for each of them. When you reach the states representing time, then the colon returns, and a tilde (\(~\)) represents afternoon (PM) hours.

When you reach the clock state again, it is useful to click the Go to Child toolbar button with the clock state selected. It brings up the child page with the contents of the clock state (see Figure 2-14). Notice the initial states here. Each thread of control has a current state that is also the previous state.
Press and hold down the Date button in the interface. Notice that the interface display shows the month and date while you are holding down this button. Notice also what happens in the statechart: The transition containing the **DatePressed** event turns red as does the DisplayDate state. The previous state is DisplayTime.

When you release the Date button, the interface display returns to the current time. In the statechart, the DisplayTime state and the transition containing **DateReleased** turn red. The previous state is DisplayDate.

When the controller initially enters the clock state, OneMinuteWait is both the current and the previous state until the timer indicates that one minute has passed. At that point, the transition turns red and stays red until you press the Set button, which causes the controller to leave the clock state.

To stop the model, close the windows.
2.14 Gaining Expertise in Using BetterState

Congratulations! You have just completed the BetterState Getting Started guide! We suggest that you briefly review the tasks in this tutorial and the process you undertook to create a controller.

We also suggest that you spend some time with our demo models, both in using them and in examining the code for them. You can run all the demos on BetterState Lite, regardless of the number of states in the model. The demos are all located in the following directory:

\ISIHOME\bs_6xx.x\examples

where \ISIHOME\ is the root installation directory for MATRIX\_X products. Pay particular attention to the application schedulers for these models (modelnamemain.c).

As you have no doubt realized in using this tutorial, you have used but a few of the constructs provided by BetterState! For a comprehensive understanding of BetterState, see the BetterState User's Guide.
Index

A

action
  on-entry, creating 29
  sample code 45
  supplying user code 44
  transition, assigning 36
appearance
  threads of control 41
application
  components (illustration) 47
  scheduler 12, 47
  pseudo code for main program 49
assertions 13

B

BetterState
  advanced features 6
  application areas 2
  design
    environment 7
    features 6
  diagrammatic constructs 3
  internationalization 8
  introduction and overview 6
  product definition 1
  window 17

C

chart
  animation 13
    generating code for 45
    running demo of clock 52
    running your model of clock 51
    stopping 55
    updating statechart 53
    using 50
  basic information 21
  constructs 6
  example 4
  properties
    setting initial 22
    setting others 45
See also statechart, controller
Chart Properties toolbar button 45
child page 38
  navigating to 40
circled
  bar, chart animation 52
clock model 16
  design requirements 16
  event-driven controller 23
  user interface 16
coarse state 38
BetterState 6.0
Getting Started

_**code**_
- file location 45
- generation
  - language 10, 24
  - overview 10
- integrating, overview 12
- responsibility for generating 48
- sample actions 45
- testing and debugging 12

_**concurrence**_ 6

**Concurrence toolbar button** 41

**constructs**
- diagrammatic 3
- flowchart 5
- statechart 6

**contents, tutorial** 15

**control implementation, setting** 24

**controller** 47

See also _chart, statechart_

**current object, definition** 27

---

_**D**_

**debugging**
- overview 13
- tools 50

**default state** 27, 29

**design**
- environment, sample dialogs 8
- features, BetterState 6

**digital clock**
- model 16
- user and designer 53

**drawing**
- self-looping transition 35
- state 26
- threads of control 41
- transition 30

---

_**E**_

**environment, BetterState design** 7

---

_**event**_
- for model 53
- handler 11
- handling 47
- transition 33

**event-driven control implementation** 24
- overview 11

**example**
- state transition diagram 3
- statechart 4

---

_**F**_

**finite state machines** 3

**flowchart**
- constructs 5
- definition 5
- frame 18

---

_**G**_

**Generate Code for Chart toolbar button** 46

**generating code for model** 46

**Go to Child toolbar button** 40, 54

**Go to Parent toolbar button** 39, 43

**graphical programming** 2

---

_**H**_

**handles**
- selection, definition 27

**header file, creating separate** 45

---

_**I**_

**initial state visited** 27

**integrating code, overview** 12

**interface, external world** 47

**internationalization, BetterState** 8
ISIHOME 44

L
language, code generation 10
layers 13

M
model
digital clock 16
running chart animation 53

N
navigating
child page to parent page 39
parent page to child page 40

O
object, current 27
on-entry actions, creating 29
overview, tutorial 15

P
page
child 38
definition 18
parent 38
constructing states 27
navigating to 39
priorities, transition 25
project
contents 18
name 22
properties, setting 18

R
reactive system
definition 2
designing 1
resting state 26

S
sample code, actions 45
scheduler, application 47
selection handles 27
self-looping transition, drawing 35
state
coarse 38
default 29
drawing 26
representation 26
resting 26
transition diagram, example 3
statechart
calling 53
definition 4
example 4
updating during chart animation 53
Statechart window 18
illustration 7
subpage 38
symbols
UML 19
system
event-driven 11

T
testing and debugging 12
threads of control 6
creating 41
drawing 41
representation 41
toolbar button
Chart Properties 45
Concurrence 41
Generate Code for Chart 46
Go to Child 40, 54
Go to Parent 39, 43
transition
  action, assigning 36
  definition 30
  drawing 30
  event 33
  placement 35
  representation 30
  self-looping 34, 43
drawing 35
traveling. See navigating
tutorial, overview 15

U
UML symbols, using 19
Unified Modeling Language (UML)
symbols 19
user code
  actions 44
  application scheduler 48
  interface to external world 48

V
visual settings 19

W
window
  BetterState 17
  BetterState Editor 7
  Statechart 7