

# LabVIEW™ State Diagram Toolkit User Guide

The LabVIEW State Diagram Toolkit assists in large scale application development by providing a framework in which you can build state machines in LabVIEW. Using the State Diagram Toolkit, you can create a state diagram that reflects a complex decision-making algorithm while LabVIEW simultaneously generates the block diagram code necessary to implement the state machine.

This user guide contains an introduction to state machines and exercises that help you become familiar with the State Diagram Toolkit. Throughout the exercises, you will learn how to use the State Diagram Toolkit in designing state machines.

## Contents

---

Introduction to State Machines .....	2
Testing the Primality of a Number.....	3
Defining a Prime Number Algorithm .....	4
Testing the Numbers 1, 2, and 3 .....	4
Creating the Initial State Diagram .....	4
Creating New States .....	5
Creating Terminal States .....	6
Creating Additional Transitions .....	7
Exploring the Block Diagram .....	8
Revisiting the Prime Number Algorithm.....	10
Initializing the Input n Case.....	10
Determining If n Is Equal to 1, 2, or 3.....	11
Indicating the Primality of a Number .....	13
Running the Prime Test VI.....	14
Reordering Transition Priorities .....	14
Testing Numbers Greater than 3 .....	15
Revisiting the Prime Number Algorithm.....	15
Modifying the State Diagram .....	16
Creating a Divisor d.....	17
Setting the Divisor Equal to 2.....	18

---

LabVIEW™, National Instruments™, NI™, NI Developer Zone™, and ni.com™ are trademarks of National Instruments Corporation. Product and company names mentioned herein are trademarks or trade names of their respective companies. For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/patents`.

June 2003  
323603A-01

Dividing $n$ by $d$ .....	19
Increasing the Divisor by 1 .....	20
Running the Prime Test VI.....	21
Calling a State Machine VI as a SubVI.....	23
Where to Go from Here .....	23

## Introduction to State Machines

---

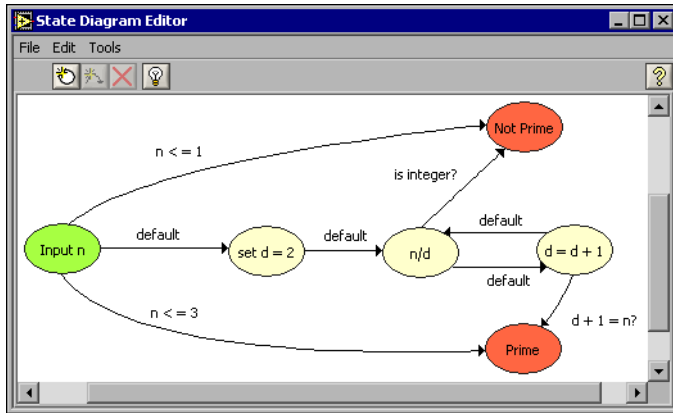
A state machine is a system with unique states and transitions that govern the execution flow of the system. In applications where distinguishable states exist, you create a state diagram to graphically represent a state machine. Within a state diagram, each state can lead to one or multiple states or can end the process flow. Each state diagram relies on user input or in-state calculations to determine the next state to execute. All applications require an initial state, followed by transition states that perform different actions. A terminal state is the final state executed and performs clean up actions.

State diagrams are useful in simplifying the design process of applications that use complex decision-making algorithms. In addition to visualizing the flow of a complex decision-making algorithm, the state diagram is a functional form of application planning. To create an effective state diagram, you must know the various states of the application and how the states relate to one another. By visualizing the various execution states of an application, you improve the overall design of the application.

The State Diagram Toolkit provides a user interface that you can use to design state diagrams. In the **State Diagram Editor** window, shown in Figure 1, the ovals represent states in the system and the arrows represent transitions between the states.

In Figure 1, the **Input  $n$**  state is the initial state and the **Not Prime** and **Prime** states are terminal states. Depending on the value of  $n$ , the state diagram determines if the number is either not prime or prime.

To determine which state executes after the **Input  $n$**  state, the state diagram evaluates the value of  $n$  using the conditions specified by the three transition arrows associated with the **Input  $n$**  state. If  $n$  is less than or equal to 1, the number is not prime. If  $n$  is less than or equal to 3, but not equal to 1, the number is prime. If the number is greater than 3, the three states—**set  $d = 2$** ,  **$n/d$** , and  **$d = d + 1$** —execute to determine the primality of a number.



**Figure 1. State Diagram Editor Window**

The exercises in this user guide will teach you how to create the state diagram in Figure 1 and write a VI that determines the primality of a number.

The State Diagram Toolkit contains two components—the **State Diagram Editor** window and the automatically generated block diagram code. The **State Diagram Editor** window provides a view of the execution flow of the application, while the block diagram defines what action occurs in each state and when each transition occurs.



**Note** In this document, a state diagram refers to the ovals and arrows in the **State Diagram Editor** window. A state machine refers to the block diagram code that implements the decision-making algorithm.

Refer to the *Application Design Patterns: State Machines* Application Note for more information about state machines. This Application Note is available on the NI Developer Zone at [ni.com/zone](http://ni.com/zone).

## Testing the Primality of a Number

The following exercises teach you how to develop a basic VI that tests the primality of a number. Throughout each exercise, you will see illustrations that provide information about the concepts associated with certain steps.

Refer to the `examples\statediagram` directory for the solution to these exercises.

## Defining a Prime Number Algorithm

A number  $n$  is prime if it is a positive integer greater than 1 and is divisible by no other positive integers other than 1 and itself. The following algorithm is just one of many ways for determining whether a number is prime.

A user picks a number  $n$ .

If  $n = 1$ , the number is not prime. (quit)

If  $n = 2$  or  $3$ , the number is prime. (quit)

If  $n > 3$ , you must test whether the number is divisible only by the number 1 and itself.

Start by setting a divisor  $d$  equal to 2.

If  $n/d$  is an integer,  $n$  is not prime. (quit)

If  $n/d$  is not an integer, increase the divisor by 1 setting  $d = d + 1$ .

Test until  $d + 1 = n$ .

If  $d + 1 = n$ , the number  $n$  is prime. (quit)



**Note** From a performance perspective, this algorithm is not the most efficient method for determining prime numbers. However, for the purposes of these exercises, this algorithm provides a straightforward example of how to use the State Diagram Toolkit.

## Testing the Numbers 1, 2, and 3

Using the **State Diagram Editor** window, you can create a state diagram that reflects the logic described in the *Defining a Prime Number Algorithm* section. On the block diagram, you then can configure the details of the algorithm to create a VI that tests whether a number is prime.

In these exercises, you will create a VI that determines the primality of the numbers 1, 2, and 3.



You can complete the exercises in approximately 45 minutes.

## Creating the Initial State Diagram

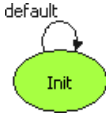
Complete the following steps to launch the **State Diagram Editor** window.



1. Display the block diagram and select the State Diagram VI, shown at left, from the **State Diagram Editor** palette.
2. Move the cursor to the upper left corner of the block diagram. Place the top left corner of the state diagram here.



**Note** You place the State Diagram VI on the block diagram similar to how you draw a While Loop on the block diagram.



- Click and drag the cursor diagonally so that the structure fills up the block diagram.

Notice how the **State Diagram Editor** window, shown in Figure 1, automatically opens. The window includes only an initial state, shown at left, represented by a green oval. An initial state accepts inputs to the state machine and is the first state to execute when you run the state machine.

The initial state also has a transition arrow attached to it. Transition arrows represent transitions between states and indicate which state executes next when a certain condition is true.

- On the front panel, place a numeric control.
- Label the numeric control *n*.

Remember that the first step in the prime number algorithm is for the user to pick a number *n*. This numeric control **n** corresponds to the number a user inputs.

- Change the representation of the control to a 32-bit signed integer (I32).
- Save this VI as `Prime Test.vi`.

## Creating New States

The first step in creating this state diagram is to specify the basic logic. By definition, the number 1 is not prime and the numbers 2 and 3 are the smallest prime numbers. If the number is greater than 3, you must use the prime test algorithm to continue testing the number. This definition indicates that you need four states—an input state, a state that indicates you need to keep testing, a not prime state, and a prime state.

Complete the following steps to build a state diagram that includes these four states.

- In the **State Diagram Editor** window, double-click the **Init** state and change the label to `Input n`.

Notice how when you select this state, the border is pink. When you select any object in the **State Diagram Editor** window, LabVIEW outlines that object in pink.



- You need to have a state that indicates when further testing is necessary. To create a new state to indicate you need to keep testing a number, click the **New State** button, shown at left.

Notice how a new state appears in the window. Unlike the initial state, this state is yellow and represents a transition state. Transition states are intermediate states that execute depending on certain conditions.

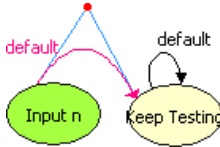


- Change the label of the new state to `Keep Testing`, shown at left.

4. Move the **Keep Testing** state to the right of the **Input n** state.

Notice how the **Keep Testing** state has a **default** transition arrow. Each new state you create has a **default** transition arrow. The **default** transition arrow indicates the state that executes by default if no other transition conditions are true. You cannot delete the **default** transition arrow.

5. Click and drag the arrowhead of the **default** transition arrow of the **Input n** state to connect the **Input n** state to the **Keep Testing** state.



Notice how when you select this arrow, the arrow is pink. When you move the arrow, blue guide lines, shown at left, appear to help you change the shape of the arrow. Drag the line point to configure the shape of the arrow.

6. Create a new state labeled `Not Prime`.
7. Create a new state labeled `Prime`.

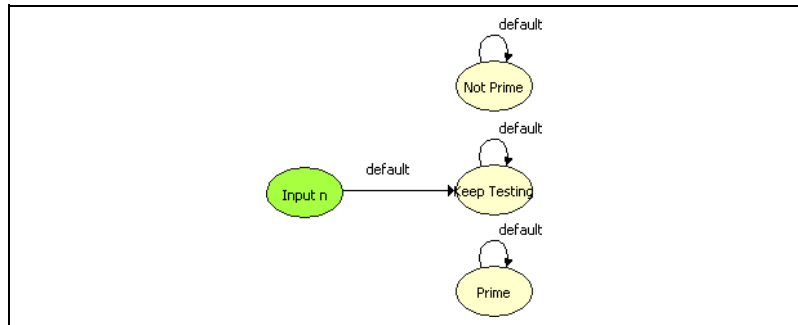


**Tip** When you click the **New State** button, LabVIEW places a new state directly under the initial state. To place a new state anywhere on the **State Diagram Editor** window, right-click a blank space and select **New State** from the shortcut menu.

## Creating Terminal States

A number is not prime, prime, or requires more testing. These are final decisions about a number that the VI makes when evaluating a number. Complete the following steps to make the **Keep Testing**, **Not Prime**, and **Prime** states terminal states.

1. In the **State Diagram Editor** window, arrange the four states so they look like the states in Figure 2.



**Figure 2.** Arranging the States



2. Right-click the **Not Prime** state and select **Make Terminal** from the shortcut menu.

Notice how the oval, shown at left, now changes to the color red. The red oval represents a terminal state that has no transition arrows leaving it.

A terminal state returns the outputs of the state machine and is the last state to execute when you run the state machine. Terminal states represent final decisions. The state machine stops after executing the code in a terminal state.

3. Configure the **Keep Testing** state as a terminal state.
4. Configure the **Prime** state as a terminal state.

## Creating Additional Transitions

Complete the following steps to create transitions that connect the **Input n** and **Not Prime** states and the **Input n** and **Prime** states.



1. In the **State Diagram Editor** window, select the **Input n** state and click the **New Transition** button, shown at left, to create a new transition arrow.

Notice how a new transition arrow appears. LabVIEW automatically attaches the head of the transition arrow to the cursor. When you use the cursor to select another state, LabVIEW creates the transition from one state to another.

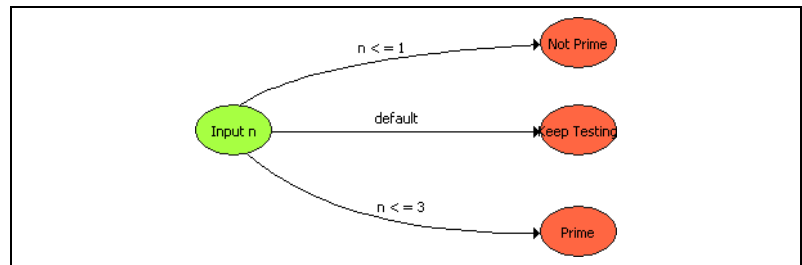
2. Click the **Not Prime** state to create the transition from the **Input n** state to the **Not Prime** state.
3. Double-click the transition arrow to change the label to  $n < = 1$ .



**Tip** You can click and drag the transition arrow labels to move them in the **State Diagram Editor** window.

4. Create a new transition from the **Input n** state to the **Prime** state and label it  $n < = 3$ .

The state diagram should look like the state diagram in Figure 3.



**Figure 3.** State Diagram for the Prime Test VI

5. Save the VI.

Keep the **State Diagram Editor** window open for use in later exercises.



**Note** If you close the **State Diagram Editor** window, you can right-click the border of the outer While Loop on the block diagram and select **Edit State Diagram** from the shortcut menu to open the **State Diagram Editor** window.

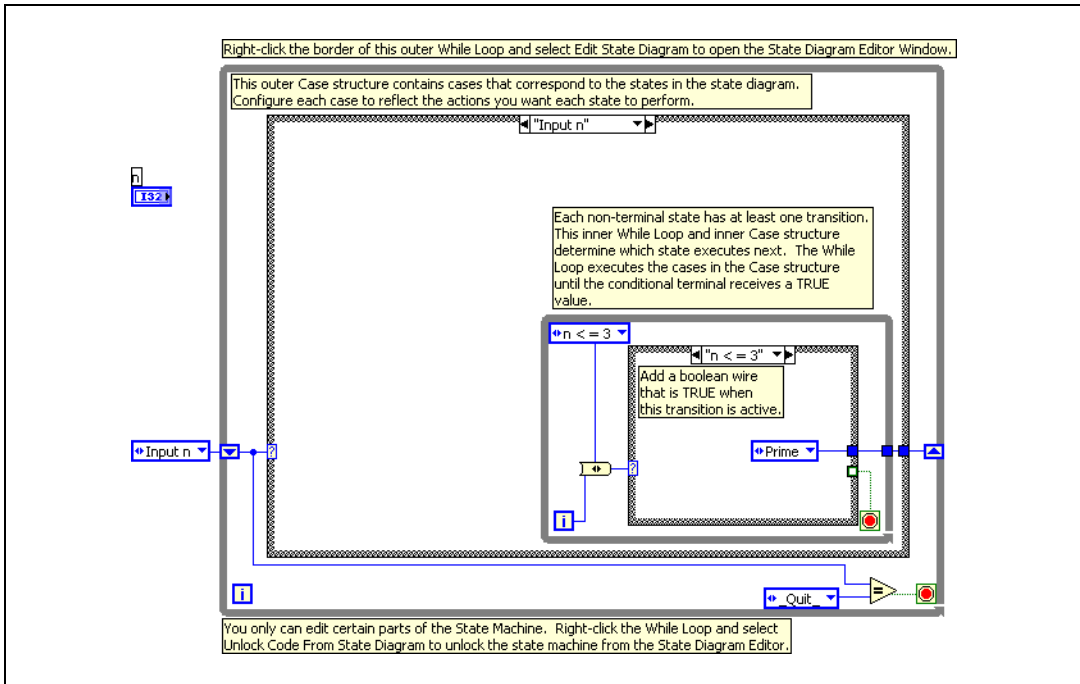
## Exploring the Block Diagram

Complete the following steps to examine the block diagram in greater detail.

1. Display the block diagram.

Notice how the block diagram already contains code generated by the State Diagram Toolkit. This automatically generated block diagram code, shown in Figure 4, uses two While Loops and two Case structures to represent the states and transitions in the **State Diagram Editor** window.

When you click the **New State** button or the **New Transition** button, the State Diagram Toolkit automatically updates the block diagram code to reflect the states and transitions you added.



**Figure 4.** Block Diagram Code for the Initial State



The outer While Loop executes the block diagram code until the state machine executes a terminal state. The outer Case structure contains the block diagram code to run for each state on the state diagram.

2. Scroll through the cases in the outer Case structure.

Notice how each case in the outer Case structure corresponds to a state in the **State Diagram Editor** window. There is one case for each state on the state diagram as well as a **\_Quit\_** case.

By default, when you create a new state diagram, the outer Case structure contains two cases, **Init** and **\_Quit\_**. The **Init** case, which you renamed **Input n** in the *Creating New States* section, corresponds to the initial state on the state diagram and is the first case to execute.

3. Select the **Not Prime** case in the outer Case structure.

Notice that all cases corresponding to terminal states contain only one case, the **default** case, in the inner Case structure. By default, this **default** case automatically calls the **\_Quit\_** case after executing.

The **\_Quit\_** case is the last case to execute. When the current case of the outer Case structure is the **\_Quit\_** case, the While Loop terminates and the VI stops running.



**Note** The **\_Quit\_** case is part of the block diagram code, but it has no corresponding state on the state diagram. This case exists to indicate to the outer While Loop when to stop the VI. When the conditional terminal on the outer While Loop receives an enumeration value of **\_Quit\_**, the VI stops.

4. Select the **Input n** case in the outer Case structure.
5. Scroll through the cases in the inner Case structure.

Notice how each case in the inner Case structure corresponds to a transition in the **State Diagram Editor** window. There is one case for each transition in the **State Diagram Editor** window.

If you have multiple transitions, represented by multiple cases in the outer Case structure, the inner While Loop and Case structure govern which case executes next. The inner While Loop executes the block diagram code until all cases in the inner Case structure complete execution.

You can add VIs and functions to the inner Case structure to specify the conditions that must be met before executing another case.



**Note** The While Loops and Case structures on the block diagram are permanent. You cannot alter the core functionality of the state machine code. If you want to modify the code, you must unlock the block diagram from the state diagram. Right-click the border of the outer While Loop and select **Unlock Code From State Diagram** from the shortcut menu. If you unlock the block diagram code, you can no longer use the **State Diagram Editor** window to edit it.

6. Select the **Input n** case in the outer Case structure and the **default** case in the inner Case structure.

By default, the inner Case structure contains only one case, the **default** case. The **default** case represents the **default** transition arrow associated with each state on the state diagram. The **default** transition arrow indicates the state that executes when no other transition conditions are true. To reflect this behavior, the **default** case always contains a TRUE constant wired to the conditional terminal of the inner While Loop.

## Revisiting the Prime Number Algorithm

The block diagram code provides the structure for the state machine, but you still need to add the logic that defines the actions that occur in each state. To build a VI that tests the numbers 1, 2, and 3, remember the first half of the algorithm defined in the [Defining a Prime Number Algorithm](#) section.

A user picks a number  $n$ .

If  $n = 1$ , the number is not prime. (quit)

If  $n = 2$  or  $3$ , the number is prime. (quit)

If  $n > 3$ , you must keep testing to determine whether the number is divisible only by the number 1 and itself.

## Initializing the Input n Case

The **Input n** case corresponds to the step where a user enters a number  $n$ . Complete the following steps to initialize the **Input n** case.

1. On the block diagram, select the **Input n** case of the outer Case structure.
2. Right-click the left border of the outer While Loop and select **Add Shift Register** from the shortcut menu.
3. Make sure the numeric control labeled **n**, which you created in the [Creating the Initial State Diagram](#) section, is outside the outer While Loop.
4. Wire the **n** terminal to the left shift register terminal on the outer While Loop.
5. Wire the left shift register terminal to the right border of the outer Case structure.

Notice how LabVIEW automatically creates a tunnel on the left border of the outer Case structure.

6. Wire the tunnel to the right shift register terminal on the outer While Loop.

Notice how LabVIEW automatically creates a tunnel on the right border of the outer Case structure.

7. Save the VI.

The **Input n** case on the block diagram should look like the block diagram in Figure 5.

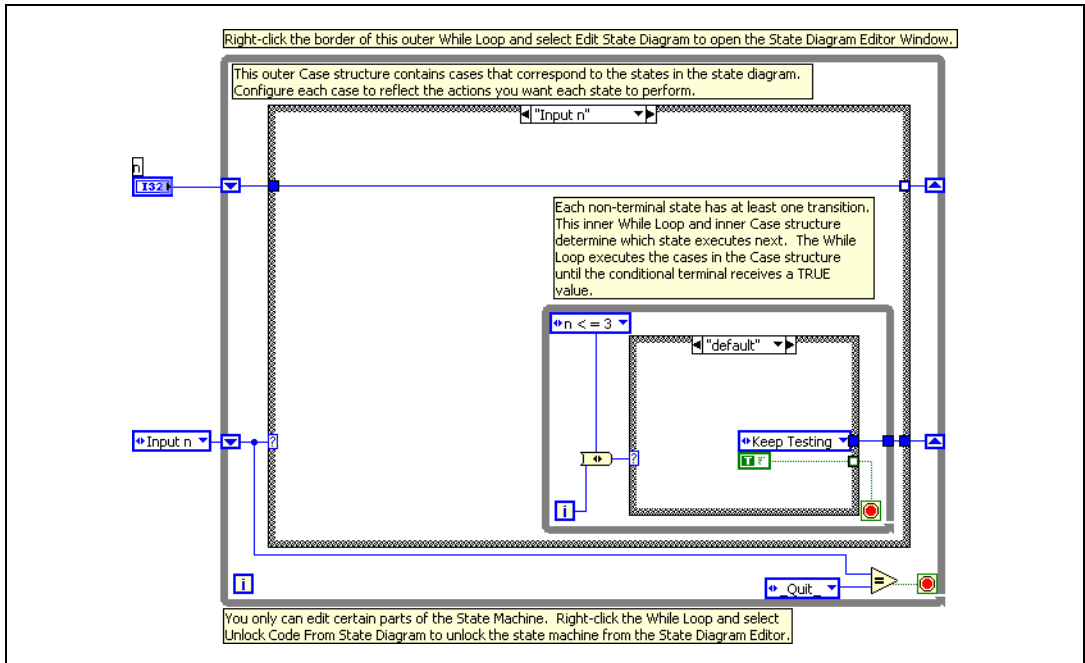


Figure 5. Initializing the **Input n** Case

## Determining If $n$ Is Equal to 1, 2, or 3

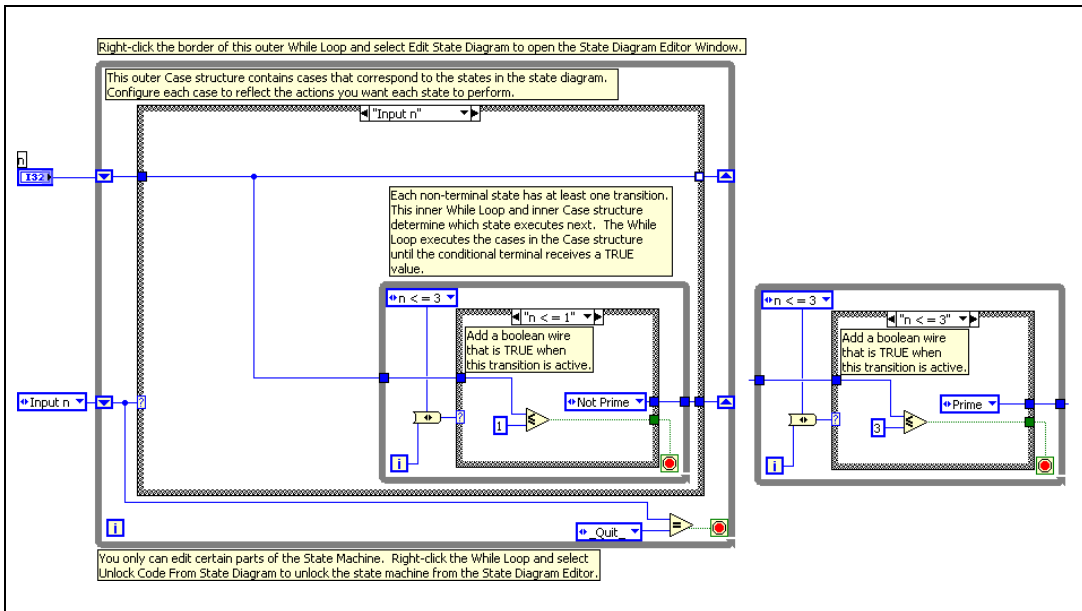
After you initialize the outer Case structure, you need to specify the conditions under which the next three cases execute. These transitions correspond to the steps that determine whether  $n$  is equal to the number 1, 2, or 3.

Complete the following steps to configure the transitions from the **Input n** case to the **Not Prime**, **Prime**, and **Keep Testing** cases.

1. Select the **Input n** case in the outer Case structure.
2. On the block diagram, select the  $n \leq 1$  case of the inner Case structure.
3. Place a Less Or Equal? function inside the  $n \leq 1$  case.

4. Wire the function so it compares the value of the **n** terminal to the number 1.
5. Wire the output of the Less Or Equal? function to the conditional terminal of the inner While Loop.  
The conditional terminal of the inner While Loop determines when to transition to the next state.
6. Select the **n ≤ 3** case of the inner Case structure.
7. Place a Less Or Equal? function in the **n ≤ 3** case.
8. Wire the function so it compares the value of the **n** terminal to the number 3.
9. Wire the output of the Less Or Equal? function to the conditional terminal of the inner While Loop.

The **Input n** case on the block diagram should look like the block diagram in Figure 6.



**Figure 6.** Block Diagram for the **Input n** Case

## Indicating the Primality of a Number

The **Not Prime**, **Prime**, and **Keep Testing** states are terminal states that correspond to cases on the block diagram. The **\_Quit\_** case has no corresponding state on the state diagram. Cases corresponding to terminal states call the **\_Quit\_** case after they execute. As the name indicates, the **\_Quit\_** case indicates to the VI when to stop running.

Complete the following steps to set up the **Not Prime**, **Prime**, **Keep Testing**, and **\_Quit\_** cases.

1. In the outer Case structure, wire the left tunnel to the right tunnel of the **n** shift register for the **Not Prime**, **Prime**, **Keep Testing**, and **\_Quit\_** cases.  
Notice how the right tunnel changes to a solid color when you properly wire all the cases in the outer Case structure.
2. Place a One Button Dialog function in the **Not Prime** case of the outer Case structure. Wire a string constant containing the message **Not Prime** to the **message** parameter.
3. Place a One Button Dialog function in the **Prime** case of the outer Case structure. Wire a string constant containing the message **Prime** to the **message** parameter.
4. Place a One Button Dialog function in the **Keep Testing** case of the outer Case structure. Wire a string constant containing the message **Keep Testing** to the **message** parameter.
5. Save the VI.

The **Not Prime**, **Prime**, **Keep Testing**, and **\_Quit\_** cases on the block diagram should look like the block diagrams in Figure 7.

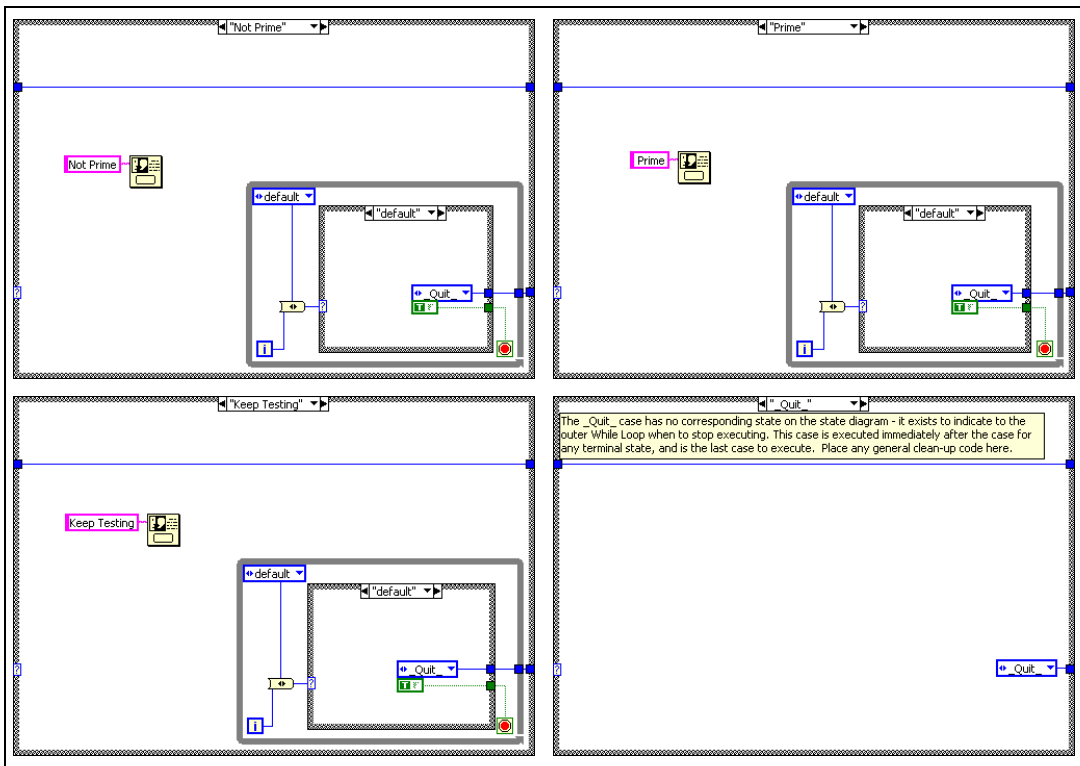


Figure 7. Block Diagram for the **Not Prime**, **Prime**, **Keep Testing**, and **\_Quit\_** Cases

## Running the Prime Test VI

Complete the following steps to test the number 1.

1. Display the front panel of the VI.
2. Enter the number 1 in the **n** numeric control.
3. Run the VI.

Notice how the VI evaluates the number 1 as prime. By definition, the number 1 is not prime. The VI incorrectly evaluates the number 1 because state diagrams depend on data flow and priorities. To correct the VI, you need to reorder the transition priorities.

## Reordering Transition Priorities

States execute in an “if...then...else if” pattern depending on the number of transition arrows leaving a state. The order of the transitions are important because the conditions defined by each transition arrow govern the execution flow of the state diagram.

LabVIEW defines the transition order based on the order in which you create the transition arrows. The last transition arrow created is the first transition arrow evaluated. When you have multiple transition arrows, the state associated with the **default** transition arrow is always the last to execute.

The state diagram currently evaluates a number as follows because of the order in which you created the transitions.

If  $n \leq 3$ ,  $n$  is prime.

If  $n \leq 1$ ,  $n$  is not prime.

If  $n > 3$ , you need to keep testing.

While this logic works for the numbers 2 and 3, the VI incorrectly evaluates 1 as prime. Complete the following steps to reorder the transition priorities so that the VI correctly evaluates the number 1.

1. In the **State Diagram Editor** window, right-click the **Input n** state and select **Reorder Transitions** from the shortcut menu.
2. Select the  **$n \leq 1$**  transition and drag the selection to the top of the list in the **Set Transition Priorities** dialog box.
3. Click the **Apply** button to apply the changes and close the **Set Transition Priorities** dialog box.
4. Display the front panel and run the VI again using the number 1.  
Notice how the VI correctly evaluates the number 1 as not prime.
5. Test the VI using the numbers 2, 3, and 4.  
Notice how the VI correctly evaluates the numbers 2 and 3 as prime. Also notice how the VI indicates that further testing is necessary to determine the primality of the number 4. The next exercise describes how to test numbers greater than 3.
6. Save the VI.

## Testing Numbers Greater than 3

In the previous exercise, the Prime Test VI notifies the user that if a number is greater than 3, more testing is necessary. In these exercises, you will add the logic to the VI to determine whether numbers greater than 3 are prime.



You can complete the exercises in approximately 45 minutes.

## Revisiting the Prime Number Algorithm

For  $n > 3$ , you need to test whether  $n$  is divisible by any factors other than 1 and itself. To perform this test, remember the second half of the algorithm defined in the [Defining a Prime Number Algorithm](#) section.

Start by setting a divisor  $d$  equal to 2.

If  $n/d$  is an integer,  $n$  is not prime. (quit)

If  $n/d$  is not an integer, increase the divisor by 1 setting  $d = d + 1$ .

Test until  $d + 1 = n$ .

If  $d + 1 = n$ , the number  $n$  is prime. (quit)

## Modifying the State Diagram

To add the logic necessary to determine the primality of a number, you need more states on the state diagram. Complete the following steps to modify the state diagram to include more states.

1. In the **State Diagram Editor** window, right-click the **Keep Testing** state and select **Make Non-Terminal** from the shortcut menu to change the state back to a non-terminal state.

As a terminal state, the **Keep Testing** state does nothing more than tell you that you need to continue testing before determining whether a number is prime. By making the state non-terminal, you can add more states to help determine the primality of a number.

2. Double-click the **Keep Testing** state and change the label to `set d = 2`.



**Note** When naming states and transition arrows, select names that are intuitive. Names that describe the condition on which that state executes help users understand the state diagram logic.

3. Create two new states labeled  $n/d$  and  $d = d + 1$ .

These two states and the **set d = 2** state are states that you will configure to calculate whether a number is divisible by any number other than 1 and itself.

4. Connect the **set d = 2** state to the  $n/d$  state with the **default** arrow originating from the **set d = 2** state.
5. Connect the  $n/d$  state to the  $d = d + 1$  state with the **default** arrow originating from the  $n/d$  state.
6. Connect the  $d = d + 1$  state to the  $n/d$  state with the **default** arrow originating from the  $d = d + 1$  state.
7. Create a new transition arrow called `is integer?` that connects the  $n/d$  state to the **Not Prime** state.

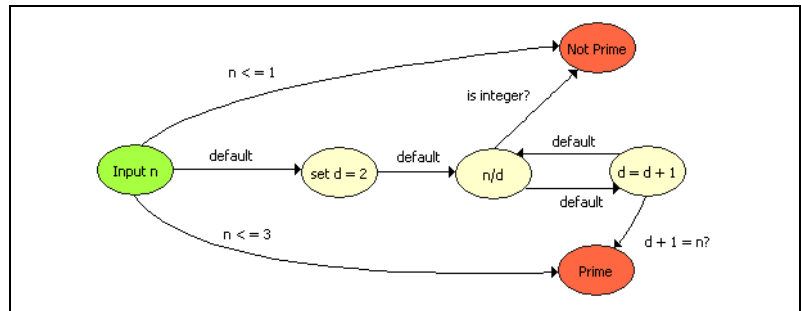


**Tip** To create a new transition arrow, you can click the state from which you want the transition to originate, press the <Shift> key, then click the state to which you want to transition. You also can right-click a non-terminal state and select **Create New Transition** from the shortcut menu to create a new transition arrow.



8. Create a new transition arrow labeled  $d + 1 = n?$  that connects the  $d = d + 1$  state to the **Prime** state.

The state diagram should look like the state diagram in Figure 8.



**Figure 8.** State Diagram for Prime Tester VI

When you added the extra states, LabVIEW added cases to the outer Case structure for each state. LabVIEW also added cases to the inner Case structure for each transition. In the following exercises, you will configure the block diagram code to determine whether numbers larger than 3 are prime.

## Creating a Divisor $d$

To determine whether a number has divisors other than 1 and itself, you need to create a divisor that represents the divisor  $d$ . Complete the following steps to create a shift register for the divisor  $d$ .

1. In the **State Diagram Editor** window, select the **set  $d = 2$**  state.
2. Display the block diagram.

Notice how the outer Case structure on the block diagram displays the state you selected in the **State Diagram Editor** window.

3. Move the One Button Dialog function and the message **Keep Testing** outside of the outer While Loop.

The message is no longer necessary in the **set  $d = 2$**  case. However, the message will be useful in a later exercise.

4. Below the  $n$  shift register, right-click the left border of the outer While Loop and select **Add Shift Register** from the shortcut menu.

The first shift register is for the number you want to test,  $n$ . The second shift register is for the divisor  $d$ . These shift registers carry the values of  $n$  and  $d$  to the next iteration if  $n/d$  is not an integer.

5. Place a numeric constant on the block diagram to the left of the shift register you just created.

6. Initialize this shift register by wiring the constant value 0 to the left terminal of the shift register.
7. Right-click the constant and select **Visible Items»Label** from the shortcut menu.
8. Change the label of the constant to d.
9. Save the VI.

## Setting the Divisor Equal to 2

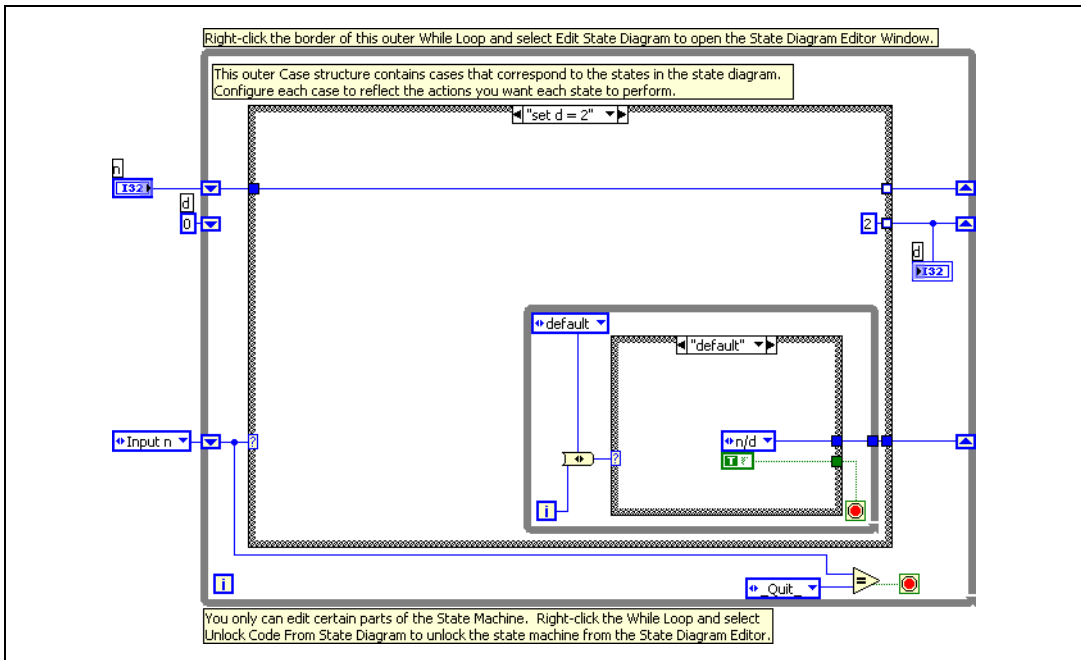
To determine whether a number has divisors other than 1 and itself, start testing with a divisor of 2, the smallest prime number. Complete the following steps to modify the **set d = 2** case.

1. Create a constant value 2 inside the **set d = 2** case and wire the constant to the right terminal of the **d** shift register.
2. Create an indicator between the outer Case structure and the outer While Loop that displays the current value of the **d** shift register.



**Tip** Create more working space on the block diagram by pressing the <Ctrl> key and using the Positioning tool to drag out a rectangle where you want more space.

The **set d = 2** case on the block diagram should look like the block diagram in Figure 9.



**Figure 9.** Block Diagram for the **set d = 2** Case

3. Select the **Input n** case of the outer Case structure.
4. Wire the left terminal of the **d** shift register to the left border of the outer Case structure.
5. In the outer Case structure, wire the left tunnel to the right tunnel of the **d** shift register.
6. For the **Not Prime**, **Prime**, and **\_Quit\_** cases, also wire the left tunnel to the right tunnel of the **d** shift register.
7. Save the VI.

## Dividing $n$ by $d$

In the  $n/d$  state, you want to test whether the number  $n$  divided by  $d$  is an integer. If  $n$  is divisible by any number between 1 and itself, the number is not prime. Complete the following steps to configure the  $n/d$  state.

1. In the **State Diagram Editor** window, select the  $n/d$  state and then the **is integer?** transition arrow.

Notice how the outer and inner Case structures on the block diagram adjust according to the state and transition arrows you selected in the **State Diagram Editor** window.



**Tip** If you have many states and transitions, selecting a specific state and transition arrow in the **State Diagram Editor** window helps you display the correct cases you want to edit on the block diagram. Instead of scrolling through multiple cases in the outer and inner Case structures to locate the state and transition, the State Diagram Toolkit automatically locates the cases for you.

2. Wire the left tunnel to the right tunnel for the **n** and **d** shift registers.
3. Place a Quotient & Remainder function on the block diagram inside the  $n/d$  case of the outer Case structure.
4. Wire the left shift register terminal of the **n** shift register to the **x** input of the Quotient & Remainder function.
5. Wire the left shift register terminal of the **d** shift register to the **y** input of the Quotient & Remainder function.
6. Place an Equal To 0? function inside the **is integer?** case of the inner Case structure.

The **is integer?** case specifies what action to take if  $n/d$  is an integer. If the Equal To 0? function returns TRUE, this transition case executes the **Not Prime** case.

7. Wire the remainder output of the Quotient & Remainder function to the input of the Equal To 0? function.
8. Wire the output of the Equal To 0? function to the conditional terminal of the inner While Loop.

- Move the One Button Dialog function and the string constant containing the message **Keep Testing** inside the **is integer?** case.
- The **n/d** case on the block diagram should look like the block diagram in Figure 10.

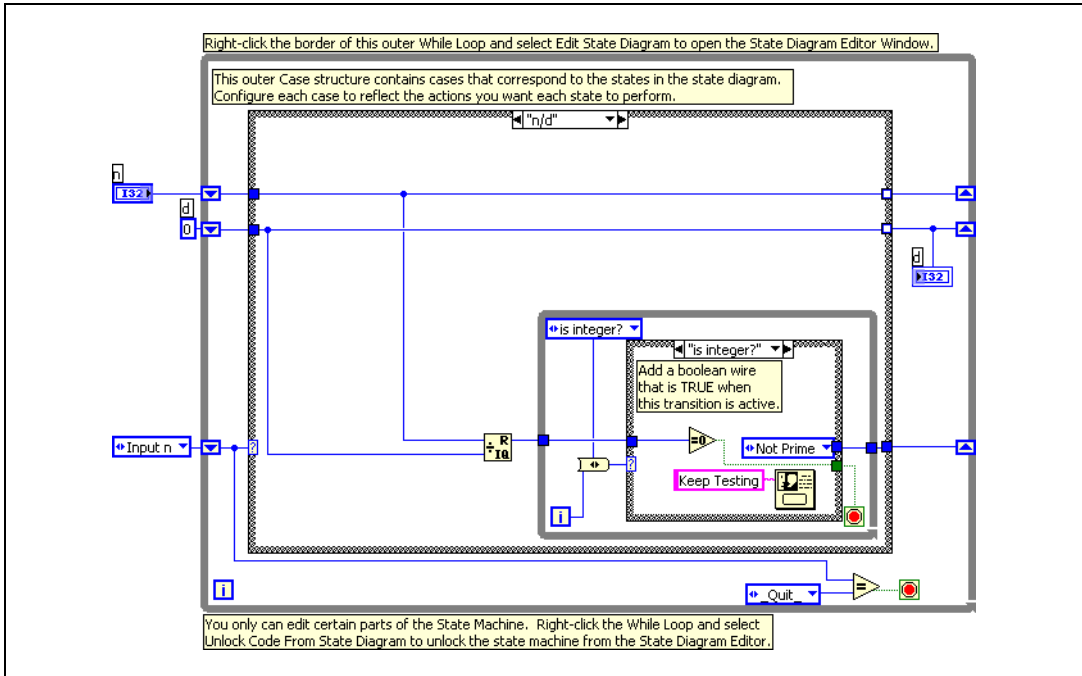


Figure 10. Block Diagram for **n/d** Case

## Increasing the Divisor by 1

If  $n/d$  is not an integer, you must increase the divisor by 1 and retest  $n/d$ . Complete the following steps to configure the  $d = d + 1$  state.

- In the **State Diagram Editor** window, select the  $d = d + 1$  state and then select the  $d + 1 = n?$  transition arrow.
- On the block diagram, wire the left tunnel to the right tunnel of the **n** shift register.
- Place an Increment function on the block diagram inside the  $d = d + 1$  case of the outer Case structure.
- Wire the left tunnel of the **d** shift register to the input of the Increment function.
- Wire the output of the Increment function to the right tunnel of the **d** shift register.
- Place an Equal? function inside the  $d + 1 = n?$  case of the inner Case structure.

7. Wire the function so it compares whether the value of the output of the Increment function is equal to the value of the  $n$  shift register.
8. Wire the output of the Equal? function to the conditional terminal of the inner While Loop.
9. Save the VI.

The  $d = d + 1$  case on the block diagram should look like the block diagram in Figure 11.

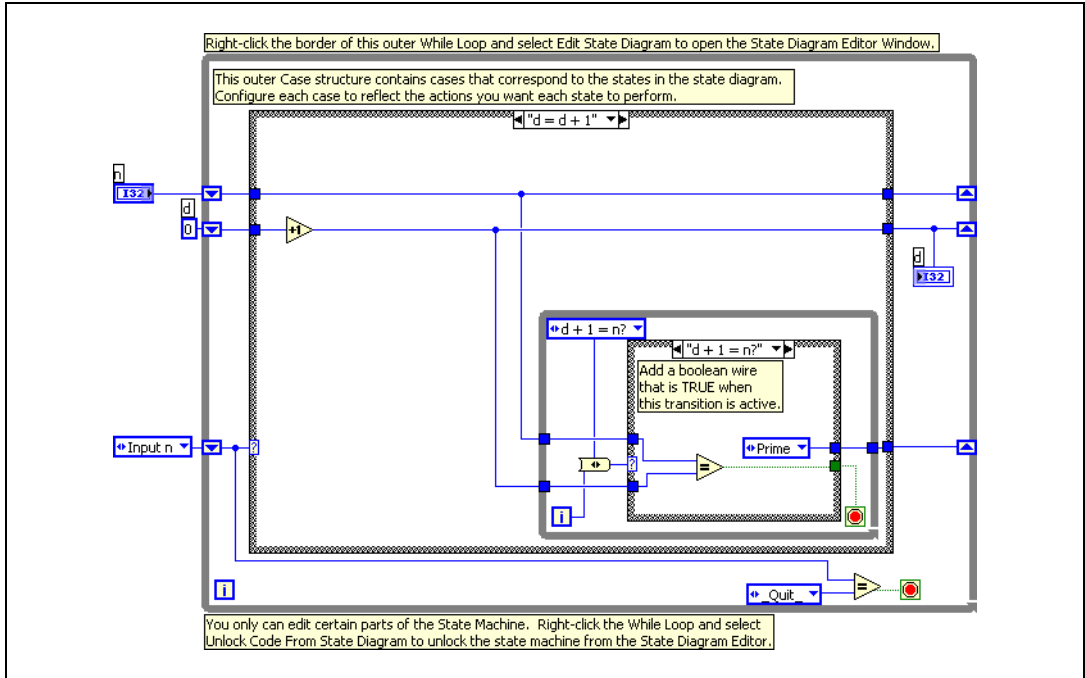


Figure 11. Block Diagram for the  $d = d + 1$  Case

## Running the Prime Test VI

You now can run the Prime Test VI to test any number. Complete the following steps to run the VI and observe how the state diagram executes.

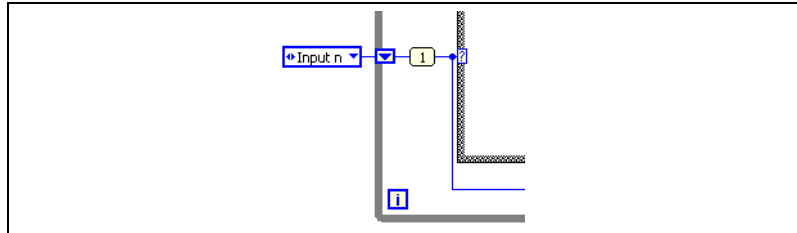
1. Display the block diagram and the **State Diagram Editor** window.
2. In the **State Diagram Editor** window, click the **Highlight Execution** button, shown at left.



The **Highlight Execution** button displays an animation of the state diagram execution when you run the VI. If the **Highlight Execution** button appears yellow, execution highlighting is enabled.



**Note** When you turn on execution highlighting in the **State Diagram Editor** window, LabVIEW automatically places a probe on the wire connecting the enumeration constant to the outer Case structure, shown in Figure 12. The probe transfers the data on the wire from the block diagram to the **State Diagram Editor** window. Using this information, the state diagram highlights the state currently executing as the VI runs.



**Figure 12.** Adding a Probe for Execution Highlighting

3. Display the front panel and **State Diagram Editor** window.
4. Enter the number 4 in the **n** numeric control.
5. Run the VI. Observe the flow of the state diagram as illustrated by execution highlighting.

Notice the divisor displayed in the **d** indicator. If the number  $n$  is greater than 3, the VI cannot automatically determine the primality of the number. Therefore, the **Keep Testing** message appears on the screen. When you click the **OK** button, the VI continues testing.

In the **State Diagram Editor** window, notice how for the number 4, LabVIEW highlights the following states:

**Input n**  
**set d = 2**  
**n/d**  
**Not Prime**

6. Enter the number 5 in the **n** numeric control.
7. Run the VI.

Again, notice the divisor displayed in the **d** indicator. For numbers greater than 3, when you click the **OK** button, the divisor **d** increases by 1 and the VI continues testing.

In the **State Diagram Editor** window, notice how for the number 5, LabVIEW highlights the following states:

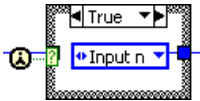
```
Input n
set d = 2
n/d
d = d + 1
n/d
d = d + 1
n/d
d = d + 1
n/d
Prime
```

By using execution highlighting, you can observe how the state diagram can simplify a complex application.

8. Save the VI.

## Calling a State Machine VI as a SubVI

After you build a state machine in LabVIEW using the State Diagram Toolkit, you can convert the VI into a subVI. Instead of continuously running the VI until it terminates, the VI executes one state and then stops. To convert a state machine VI into a subVI, in the **State Diagram Editor** window, select **Edit»Execution Mode»Single Step (Callable as SubVI)**.



The block diagram code automatically updates to contain a Case structure, shown at left, with a First Call? function wired to the selector terminal. The First Call? function returns TRUE only the first time you call the VI. The VI executes the case specified by the enumeration constant in the Case structure and then terminates. By default, the value of the enumeration constant in the **True** case is always the initial state of the state machine.

After a case executes, the VI does not run again until called. Each subsequent time you call the VI, the VI remembers which state executed the last time you called the VI and then executes the next state in the state machine.

## Where to Go from Here

---

In addition to this user guide, the State Diagram Toolkit provides example VIs that you can explore to learn more about creating state diagrams in LabVIEW.

Refer to the `examples\statediagram` directory for examples of VIs using the state diagram architecture.