7 Subsystems and Masking

In this chapter, we will explore some of the ways we can use Simulink to model more complex systems. We will build hierarchical models and develop custom Simulink blocks called masked blocks. We will also discuss conditionally executed subsystems that can make Simulink models more efficient.

7.1 INTRODUCTION

In the preceding chapters, we discussed the basics of building Simulink models for continuous, discrete, and hybrid systems. Using the procedures we covered in the proceeding chapters, it is possible to model any physical system. However, as your Simulink models become more complex, additional Simulink capabilities and programming techniques can make the models easier to develop, to understand, and to maintain. In this chapter, we'll start with a discussion of Simulink subsystems, which provide a capability within Simulink similar to subprograms in traditional programming languages. Next, you'll learn to use masking to make subsystems easier to use and understand. Last, we will discuss conditionally executed subsystems, which facilitate the development of models with multiple modes or phases of operation.

7.2 Simulink SUBSYSTEMS

Most engineering programming languages include the capability to employ subprograms. In FORTRAN, there are subroutine subprograms and function subprograms. C subprograms are called functions; MATLAB subprograms are called function M-files. Simulink provides an analogous capability called *subsystems*. There are two important reasons for using subprograms: abstraction and software reuse.

As models grow larger and more complex, they can easily become difficult to understand and maintain. Subsystems solve this problem by breaking a large model into a hierarchical set of smaller models. As a simple example, consider the automobile model of Example 5.5. The Simulink model is repeated in Figure 7.1. The model consists of two main parts: the automobile dynamics and the controller. When examining the model, it is not clear which blocks represent the automobile dynamics and which blocks constitute the controller. In Figure 7.2, we converted the automobile and controller portions of the model into subsystems. In this version, the conceptual structure is clear in the top level of the model Figure 7.2,

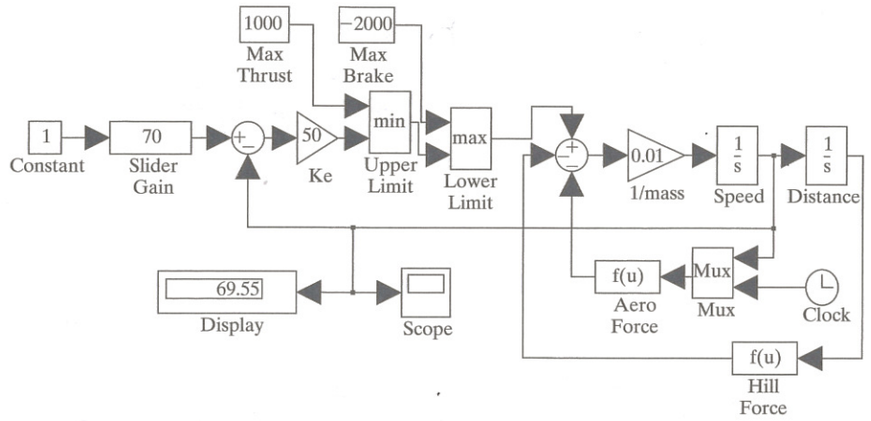


FIGURE 7.1: Automobile model with proportional speed control

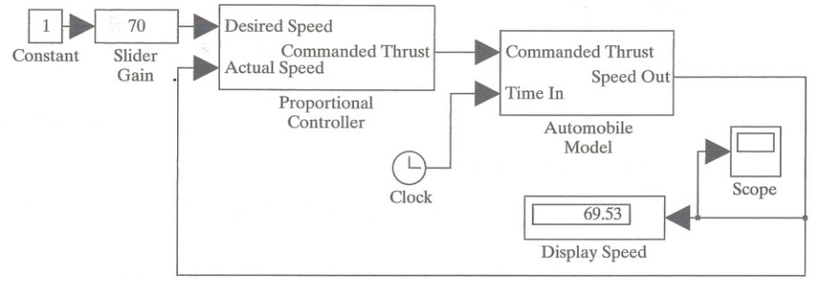
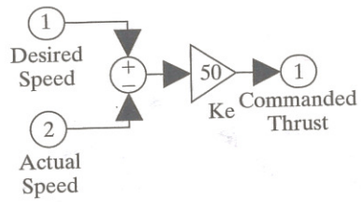


FIGURE 7.2: Hierarchical automobile model

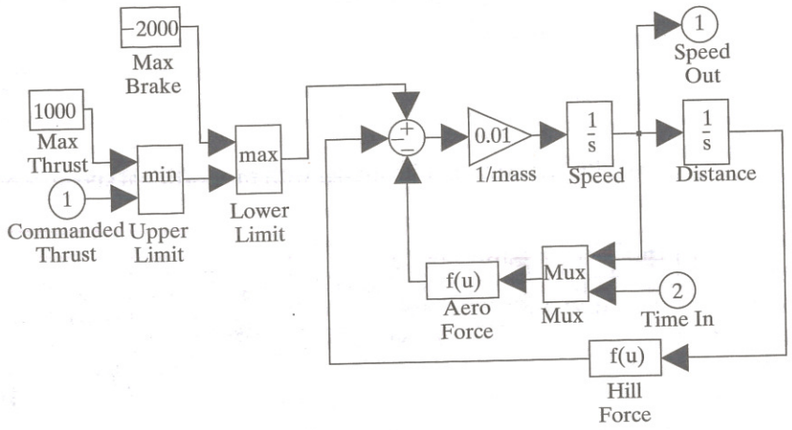
but the details of the controller and automobile dynamics are hidden in the subsystems (Figure 7.3). This hierarchical structure is an example of software abstraction.

Subsystems can also be viewed as reusable model components. Suppose that we wish to compare several different controller designs using the same automobile dynamics model. Rather than building a complete new block diagram each time, it is more convenient to build only the part of the model that is new each time—the controller. Not only does this save time building the model, but it also ensures that we are using exactly the same automobile dynamics. An important advantage of software reuse is that once we verified that a subsystem is correct, we don't have to repeat the testing and debugging process each time we use the subsystem in a new model. Subsystems greatly simplify the task of modeling physical systems that contain several instances of a particular component, such as, for example, the four tire models that would be required to model the ride characteristics of an automobile.

There are two methods to use to build Simulink subsystems. The first method is to encapsulate a portion of an existing model in a subsystem using **Edit:Create**



(a) Controller subsystem



(b) Automobile dynamics

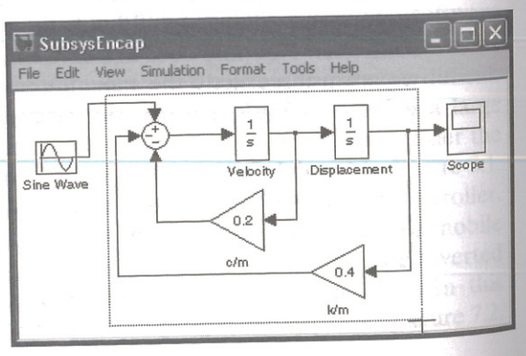
FIGURE 7.3: Hierarchical automobile model

Subsystem. The second method is to use a Subsystem block from the Ports & Subsystems block library. We'll discuss both methods.

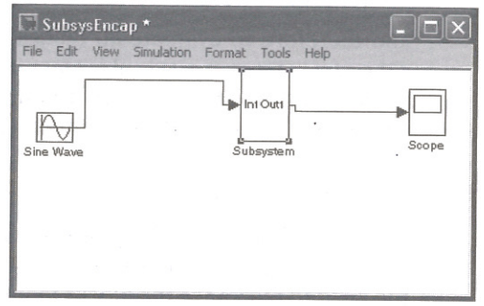
7.2.1 Encapsulating a Subsystem

To encapsulate a portion of an existing Simulink model into a subsystem, proceed as follows:

Select all the blocks and signal lines to be included in the subsystem using a bounding box. *Note that you must use a bounding box in this instance.* It is frequently necessary to rearrange some blocks so that you can enclose only the desired blocks in the bounding box.

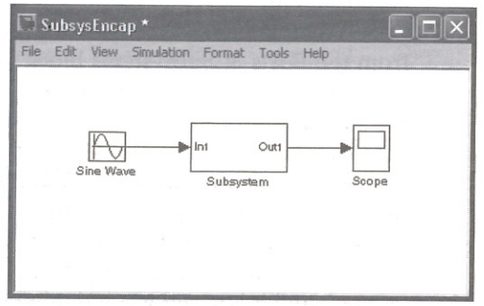


Choose **Edit:Create Subsystem** from the model window menu bar. Simulink will replace the selected blocks with a Subsystem block with an input port for each signal entering the new subsystem and an output port for each signal leaving the new subsystem.

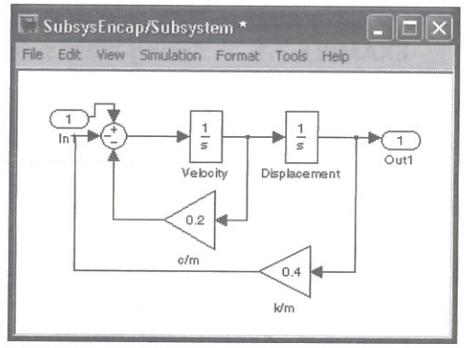


Simulink will assign default names to the input and output ports.

Resize the Subsystem block so that the port labels are readable, and rearrange the model as desired.



To view or edit the subsystem, double-click the block. A new window will appear that contains the subsystem. In addition to the original blocks, an Inport block is added for the signal entering the subsystem and an Outport block is added for the signal exiting the subsystem. Changing the labels on these ports changes the labels on the new block's icon. Click the control to close the subsystem window when you've finished editing the subsystem.



Edit:Create Subsystem does not have an inverse operation. Once you encapsulate a group of blocks into a subsystem, there is no menu choice to reverse the process. Therefore, it is a good idea to save the model before creating the subsystem. If you decide you don't want to accept the newly created subsystem, close the model window without saving, then reopen the model. To manually reverse the encapsulation of a subsystem, copy the subsystem to a new model window, open the subsystem, then copy the blocks from the subsystem window to the original model window.

By default, subsystems are *virtual*. When the Simulink model is executed, Simulink evaluates blocks according to a default evaluation order, just as if all the blocks in the subsystem were in the base model window. In some models, it may

be desirable, in terms of execution order, to treat a subsystem as a single block, and therefore to evaluate all blocks in the subsystem as a group called an *atomic subsystem*. To convert a subsystem into an atomic subsystem, select the subsystem block and then choose **Edit:SubSystem Parameters** from the model window menu bar. Select checkbox **Treat as atomic unit**.

7.2.2 Subsystem Blocks

If, when building a model, you know that you will need a subsystem, you may find it convenient to build the subsystem in a subsystem window directly. This eliminates the need to rearrange the blocks that will compose the subsystem to fit in a bounding box. It also avoids having to tidy up the model window after the subsystem is encapsulated.

To create a new subsystem using a Subsystem block, drag a Subsystem block from the Ports & Subsystems block library to the model window. Double-click the Subsystem block. The subsystem window will appear. Build the subsystem using the standard procedures for constructing a model. Use Inport blocks for all signals entering the subsystem and Outport blocks for all signals leaving the subsystem. If desired, change the labels on the Inport and Outport blocks to identify the purpose of each input and output. Close the subsystem window when you've finished building the subsystem. Note that you do not need to choose **File:Save** before closing the subsystem window; the subsystem is part of the model in which the subsystem is created, and it is saved when that model is saved.

EXAMPLE 7.1 Spring-mass subsystem

We wish to model the spring-mass system composed of carts connected as shown in Figure 7.4. We will build the model from subsystem blocks that model each cart as shown in Figure 7.5.

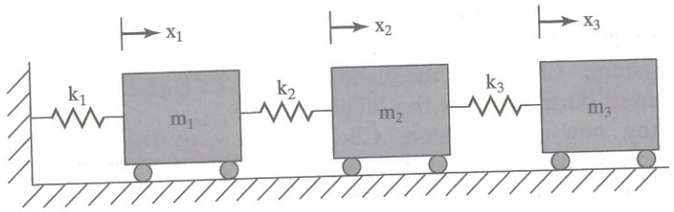


FIGURE 7.4: Spring-mass system

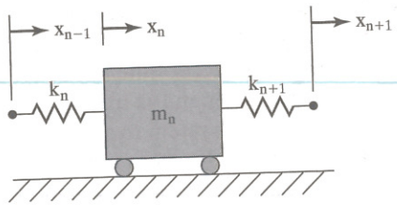


FIGURE 7.5: Single-cart model

The equation of motion for a single cart is

$$\ddot{x}_n = \frac{1}{m_n}\left[k_n\left(x_{n-1} - x_n\right) - k_{n+1}\left(x_n - x_{n+1}\right)\right]$$

Using the procedure discussed in Section 7.2.2, construct the subsystem as shown in Figure 7.6. This subsystem will model Cart 1. The inputs to the single cart subsystem are x_{n-1} (position of the cart to the left) and x_{n+1} (position of the cart to the right). The subsystem output is x_n (position of cart). Notice that each spring is referenced in two subsystem blocks: one for the cart to the right of the spring and one for the cart to the left.

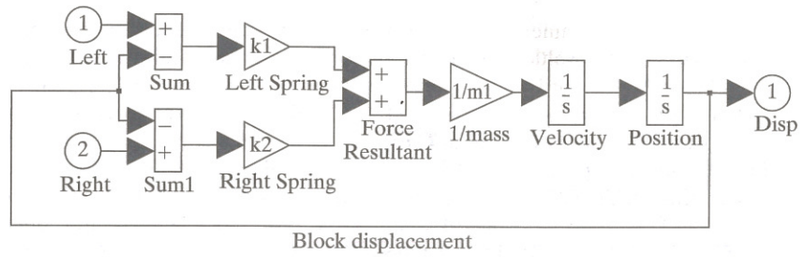


FIGURE 7.6: Cart model subsystem for Cart 1

Once the subsystem is complete, close the subsystem window. Make two copies of the subsystem block, and connect the blocks as shown in Figure 7.7.

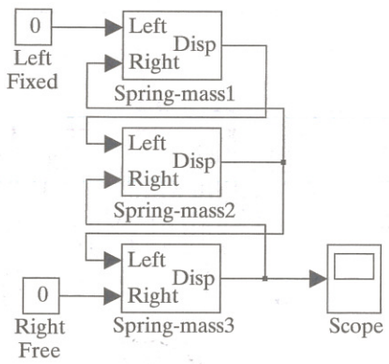


FIGURE 7.7: Three-cart model using subsystems

It is convenient in this case to enter the spring constants (`k1`, `k2`, `k3`) and cart masses (`m1`, `m2`, `m3`) as MATLAB variables, and assign values to the variables using a MATLAB script M-file (which we named `SetCartParms.m`), as shown in Listing 7.1. Execute this script M-file from the MATLAB prompt before running the simulation. (Note that the script M-file (extension `.m`) must have a name that is different from the name of the Simulink model. For example, if the Simulink model is named `examp_1.mdl`, and you name the script M-file `examp_1.m`, MATLAB will open the Simulink model when you enter the command `examp_1` at the MATLAB prompt.)

Listing 7.1: MATLAB script `SetCartParms.m` to initialize spring constants

```
% Set the spring constants and block mass values
k1 = 1 ;
k2 = 2 ;
k3 = 4 ;
m1 = 1 ;
m2 = 3 ;
m3 = 2 ;
```

The block parameters for each block in each copy of the subsystem must now be set. For Cart 1, set the value of **Gain** for the Gain block labeled `Left Spring` to `k1`, and for Gain block `Right Spring` to `k2`. Next, set **Gain** for the Gain block labeled `1/mass` to `1/m1`. Initialize the `Velocity` Integrator block to 0, and the `Position` Integrator block to 1.

For Cart 2, set the value of **Gain** for the Gain block labeled `Left Spring` to `k2`, and for Gain block `Right Spring` to `k3`. Next, set **Gain** for the Gain block labeled `1/mass` to `1/m2`. Initialize the `Velocity` Integrator block to 0 and the `Position` Integrator block to 0.

For Cart 3, set the value of **Gain** for the Gain block labeled `Left Spring` to `k3`, and for Gain block `Right Spring` to 0, because there is no right spring for this cart. Next, set **Gain** for the Gain block labeled `1/mass` to `1/m3`. Initialize the `Velocity` Integrator block to 0, and the `Position` Integrator block to 0.

We configured the Scope block to save the scope data to the workspace, and set the simulation **Start** time to 0 and **Stop** time to 100. After running the simulation, the scope data were plotted from within MATLAB, resulting in the trajectory plot for Cart 3 shown in Figure 7.8.

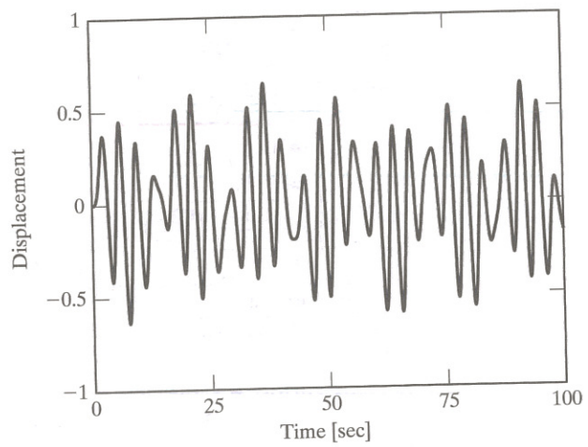
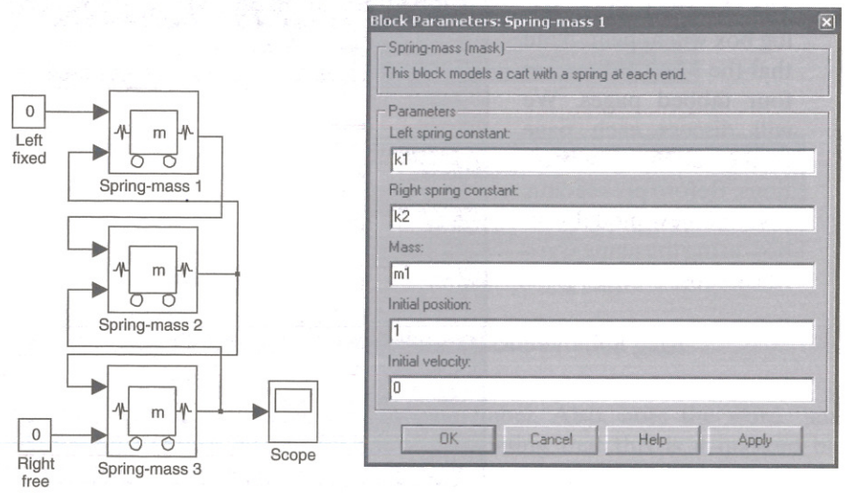


FIGURE 7.8: Trajectory of Cart 3

7.3 MASKED BLOCKS

Masking is a Simulink capability that extends the concept of abstraction. Masking permits us to treat a subsystem as if it were a simple block. A masked block may have a custom icon, and it may also have a dialog box in which configuration parameters are entered in the same way parameters are entered for blocks in the Simulink block libraries. The configuration parameters may be used directly to initialize the blocks in the underlying subsystem, or they may be used to compute data to initialize the blocks.

To understand the concept of masking, consider the model shown in Figure 7.9(a). This model is equivalent to the model in Example 7.1, but it is easier to use. Double-clicking the block labeled Spring–mass 1 opens the dialog box shown in Figure 7.9(b). Instead of opening the dialog box for each Gain block and each Integrator to set the block parameters, you can enter all the parameters for each subsystem in the subsystem's dialog box. The dialog box in Figure 7.9(b) "masks" a subsystem that is nearly identical to the subsystem in Figure 7.6.



(a) Simulink model (b) Dialog box for `Spring-mass 1`

FIGURE 7.9: Three-cart model using masked subsystems

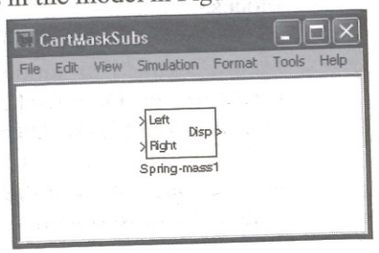
In this section, we will explain the steps in creating a masked subsystem. The examples will show how to create the spring-mass masked subsystem. Additional examples will illustrate other masking features.

The process of producing a masked block can be summarized as follows:

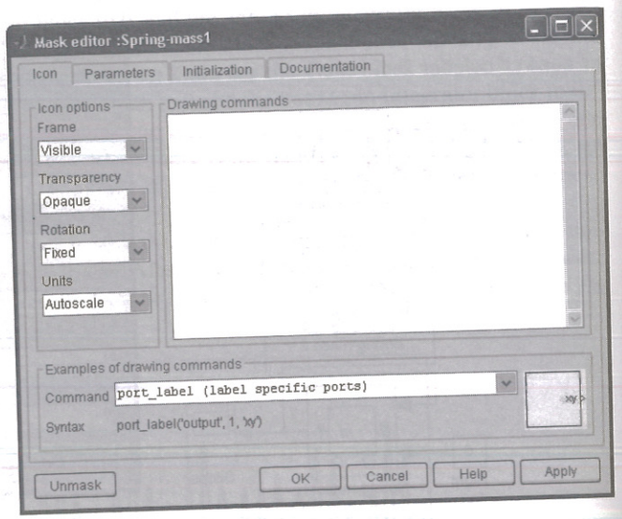
1. Build a subsystem using the procedures discussed in Section 7.2.
2. Select the subsystem block, then choose **Edit:Mask Subsystem** from the model window menu bar.
3. Using the Mask Editor, set up the mask documentation, dialog box, and optionally, build a custom icon.

7.3.1 Converting a Subsystem into a Masked Subsystem

The first step in creating a masked subsystem is to create a subsystem using the procedures described in Section 7.2. To illustrate the process, let's build a spring-mass masked block starting with one of the subsystems in the model in Figure 7.7. Open the model shown in Figure 7.7. Next, open a new model window. Drag a copy of the block labeled `Spring-mass 1` to the new model window. Select the block, then choose **Edit:Mask Subsystem** from the new model window menu bar.



The Mask Editor dialog box will appear. Note that the Mask Editor has four tabbed pages. We will discuss each page in the following subsections. Before proceeding, save the new model window using the name CartMaskSubs.



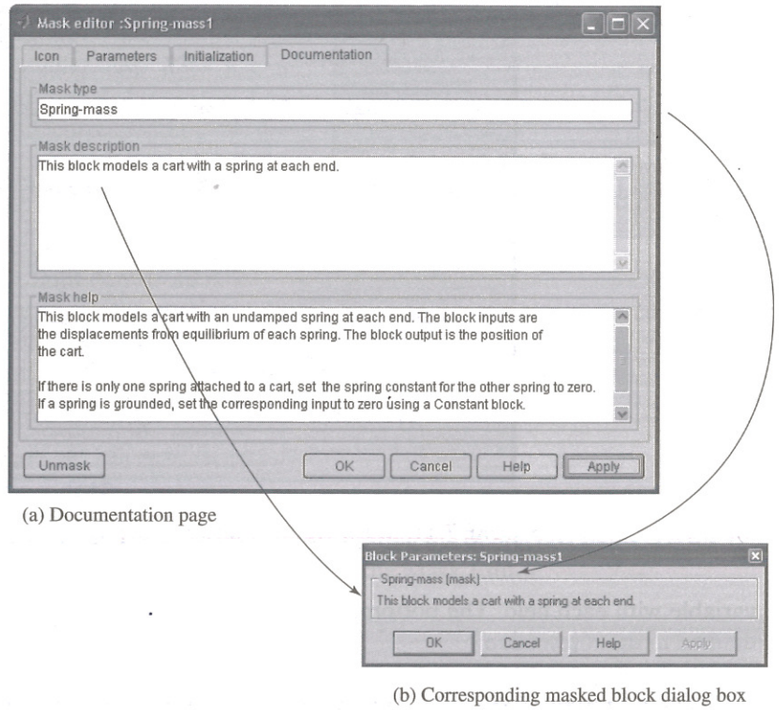
7.3.2 Mask Editor Documentation Page

The Documentation page is illustrated in Figure 7.10(a). The page consists of three fields, which in Figure 7.10(a) were filled in for the spring-mass block. All fields in the Documentation page are optional.

After filling in the fields as shown, select **Close**. Then, double-click the spring-mass block, opening the dialog box shown in Figure 7.10(b). You can return to the Mask Editor by selecting the block, then choosing **Edit:Edit Mask** from the model window menu bar.

We will discuss each field in the Documentation page next.

Mask Type Field. The contents of the first field, **Mask type**, will be displayed as the block type in the masked block's dialog box. Notice that there are two labels in the upper-left corner of the block dialog box [Figure 7.10(b)]. The label in the window title bar (here `Spring-mass1`) is the label of the currently selected block. The label inside the dialog box (here `Spring-mass`) is the block type. Every instance of this new block will have the same block type, but each instance



(a) Documentation page

(b) Corresponding masked block dialog box

FIGURE 7.10: Mask Editor **Documentation** page

in a particular model must have a different label. Also, note the word "mask" in parentheses appended to block type, indicating that this is a masked block. (Compare the masked block dialog box to the dialog box for a block from a block library.)

Block description field. The second field, **Block description**, is displayed in a bordered area at the top of the masked block's dialog box. This area should contain a brief description of the block's purpose and any needed reminders concerning the use of the block.

Block help. The third field, **Block help**, will be displayed by the MATLAB Help system when the masked block's dialog box **Help** button is pressed. This field should contain detailed information concerning the use, configuration, and limitations of the masked block and the underlying subsystem.

7.3.3 Parameters Page

The **Parameters** page (Figure 7.11) is used to define parameters for blocks in the subsystem underlying the masked block. The **Parameters** page can be divided into two sections. The top section defines masked block dialog fields and the order in which they are displayed in the masked block dialog box and associates a MATLAB

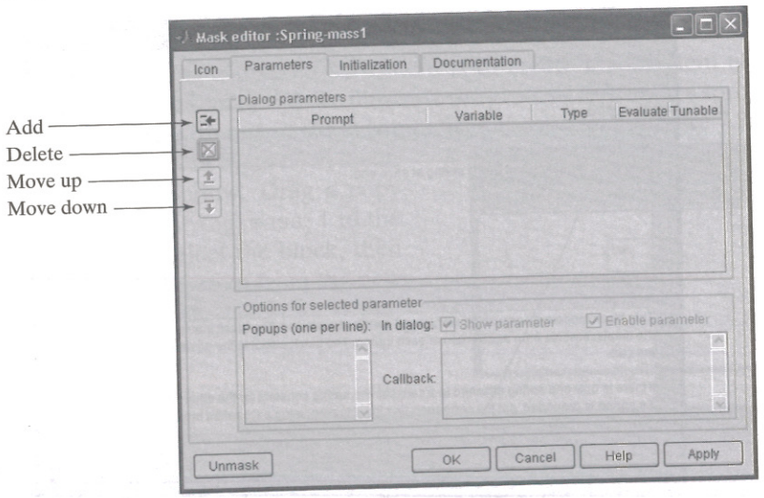


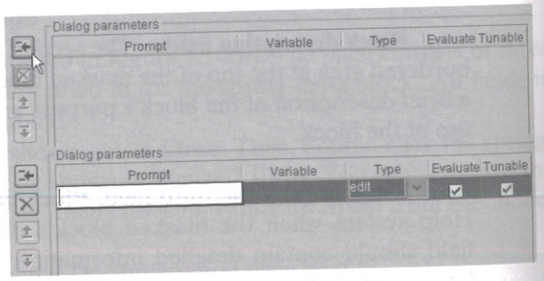
FIGURE 7.11: Mask Editor **Parameters** page

variable with each field. The bottom section contains certain options for each field defined in the top section.

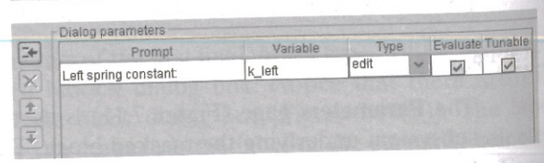
Dialog Parameters. The top section of the Mask Editor **Parameters** page is used to create, edit, and delete dialog box fields and associate a MATLAB variable with each field. This section consists of a scrolling list of dialog box fields, buttons to add, delete, and move fields, and five columns used to configure the masked block dialog box fields and associated MATLAB variables. To add the first prompt field in the Spring-mass block dialog box, proceed as follows:

Select the block, then open the Mask Editor by choosing **Edit:Edit Mask** from the model window menu bar. Select the **Parameters** page.

Click the **Add** button.

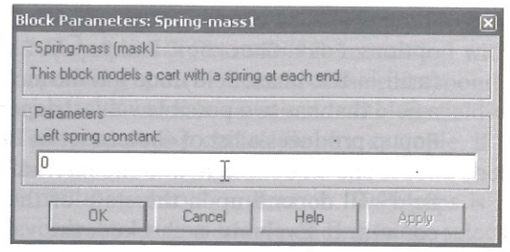


A blank line will be added to the parameter list.

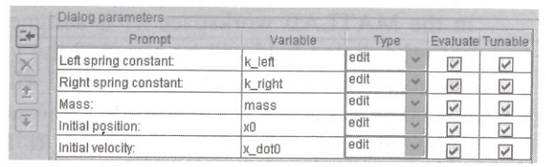


In the **Prompt** field, enter `Left spring constant`. In the **Variable** field, enter `k_left`. Notice that the prompt and the variable name are displayed in the parameter list.

Choose **OK**. Double-click the Spring-mass subsystem. The block dialog box now has a prompt. The value entered in the field corresponding to the prompt will be assigned to MATLAB variable `k_left`.

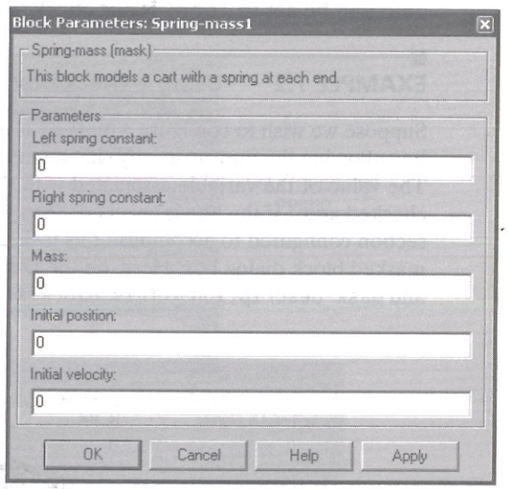


Add prompt `Right spring constant` with variable `k_right`, prompt `Mass` with variable `mass`, prompt `Initial position` with variable `x0`, and prompt `Initial velocity` with variable `x_dot0`.



That completes the process of creating the dialog box fields. To see the results, press **OK** to save the changes and exit the Mask Editor.

Double-click the Spring-mass block, opening the block dialog box, which now includes all five prompts.



Now, we will discuss in more detail the buttons and fields in the dialog box prompt section of the Mask Editor. First, notice that there are four buttons on the left side of the dialog box prompt section. These buttons are used to add, delete, and arrange dialog box prompts. To create a new field, click the line in the scrolling parameter list of the item you wish to precede the new field. Then, click the **Add** button. A blank line will appear in the scrolling list. Clicking the **Delete** button deletes the selected field. The **Up** and **Down** buttons move the selected field in the appropriate direction in the list of fields. So, if you wish for the new field to be the top field, click on first parameter, add a line, and then move the new line to the top of the list.

The third column of the parameter list, **Type**, is a drop-down list with three options: `Edit`, `Checkbox`, and `Popup`. `Edit` produces a field in which data is entered (a fill-in-the-blank field), and it is the most common type of field. `Checkbox` generates a field that has two possible values, depending on whether or not the box is checked. `Popup` produces a list of choices set in **Popup** in the Options section.

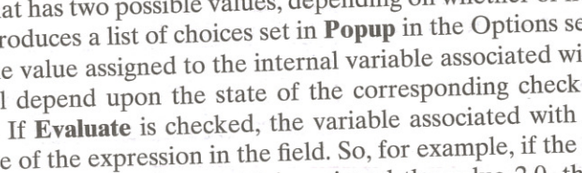
The value assigned to the internal variable associated with a block dialog box field will depend upon the state of the corresponding check box in the **Evaluate** column. If **Evaluate** is checked, the variable associated with the field will contain the value of the expression in the field. So, for example, if the field contains `k1`, and in the MATLAB workspace `k1` is assigned the value 2.0, the variable associated with the dialog field will be assigned the value 2.0. If **Evaluate** is not checked, the variable associated with the dialog field will contain the character string `'k1'`.

The last column, **Tunable**, determines whether the parameter may be changed during the execution of a simulation. If **Tunable** is checked, the parameter may be changed during a simulation, otherwise, the parameter stays constant during a simulation.

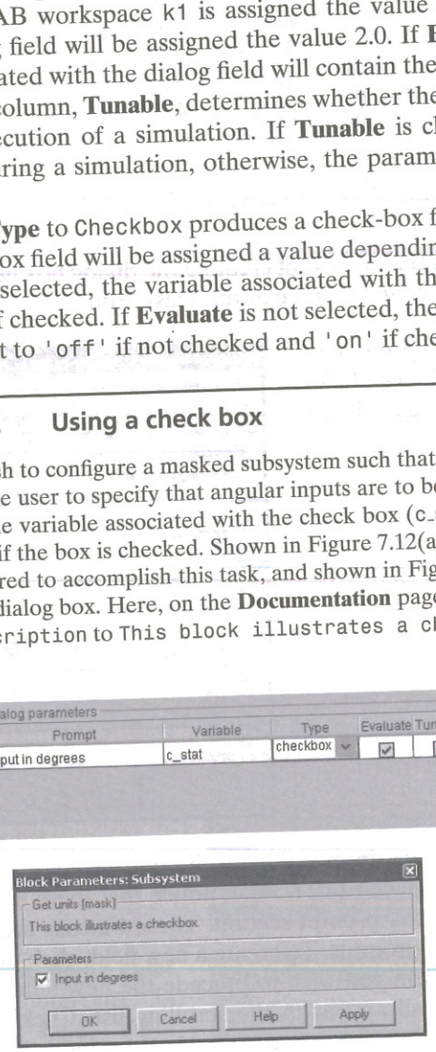
Setting **Type** to `Checkbox` produces a check-box field. The variable associated with a check-box field will be assigned a value depending on the setting of **Evaluate**. If **Evaluate** is selected, the variable associated with the field will be set to 0 if not checked or 1 if checked. If **Evaluate** is not selected, the variable associated with the field will be set to `'off'` if not checked and `'on'` if checked.

EXAMPLE 7.2 Using a check box

Suppose we wish to configure a masked subsystem such that its block dialog box has a check box allowing the user to specify that angular inputs are to be in degrees rather than radians. The value of the variable associated with the check box (`c_stat`) is to be 0 if the box is not checked and 1 if the box is checked. Shown in Figure 7.12(a) is the Mask Editor Parameters section configured to accomplish this task, and shown in Figure 7.12(b) is the corresponding masked block dialog box. Here, on the **Documentation** page, **Mask type** is set to `Get units` and `Mask description` to `This block illustrates a check box.`



(a) **Type** set to checkbox



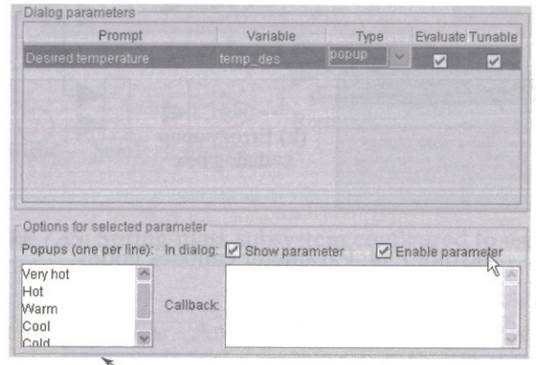
(b) Dialog box with checkbox

FIGURE 7.12: Mask Editor **Documentation** page

If **Type** is set to `Popup`, the field **Popups** in the options section is used to define a list of choices, as will be shown in Example 7.3. The variable associated with a popup field will be assigned a value depending upon the setting of the corresponding **Popups** check box. If **Popups** is checked, the variable associated with the field will be set to the ordinal number of the selected popup choice. So, for example, if the first choice is selected, the value of the variable will be set to 1. If the second choice is selected, the value of the variable will be set to 2, and so on. If **Popups** is not checked, the variable will contain the character string corresponding to the selected choice. Enter the popup choices in field **Popups**, one to a line.

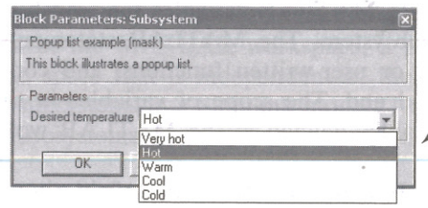
EXAMPLE 7.3 Using a popup control

Suppose we wish to create a masked block that produces an output signal defined by a popup list containing choices `Very hot`, `Hot`; `Warm`, `Cool`, and `Cold`. To do this, open a new model, then drag a Constant block into the model window. Select the Constant block using a bounding box, then choose **Edit:Create Subsystem** from the model window menu bar. Select the new subsystem, then choose **Edit:Mask Subsystem** from the model window menu bar. Set **Mask type** to `Popup example`, and **Block description** to `This block illustrates a popup list.` Configure the **Parameters** page as shown in Figure 7.13(a). Choose **OK**. Double-click the masked block, then click on the popup control. The block dialog box should be as shown in Figure 7.13(b).



(a) Prompt and popup configuration

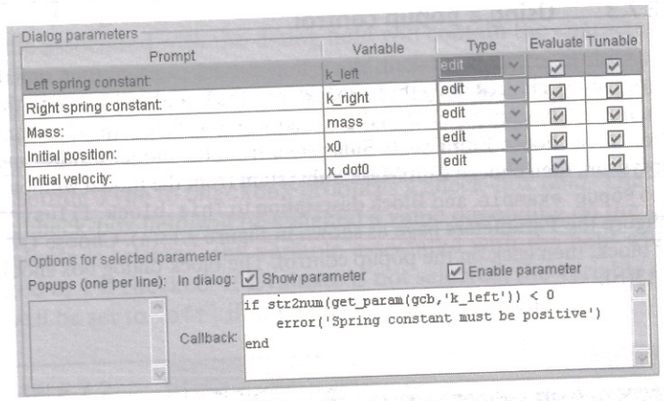
Popups defines the menu choices



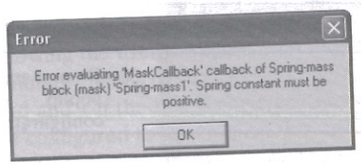
(b) Block dialog box with popup menu

FIGURE 7.13: Popup menu

Callback Field. The **Callback** field allows you to associate with the selected parameter a block of MATLAB code called a callback that is executed when the dialog parameter is entered. A detailed discussion of callbacks is presented in Chapter 9. Although callbacks can be used for many purposes in Simulink models, the masked block parameter callback is most useful for performing error and consistency checking on parameters. For example, the callback illustrated in Figure 7.14(a) will test the value entered for the left spring constant for the spring-mass block (`k_left`) and produce the error dialog in Figure 7.14(b) if a negative value is entered.



(a) Callback code



(b) Error popup dialog box

FIGURE 7.14: Parameter callback code

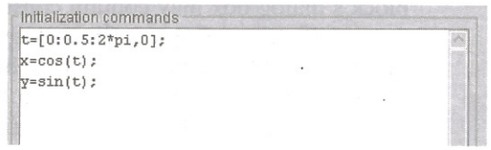
7.3.4 Initialization Page

The **Initialization** page provides a list of variables associated with the block parameters and field **Initialization commands**. This field can contain one or more MATLAB statements that assign values to MATLAB variables that can be used to configure blocks in the masked subsystem. The MATLAB statements may use any MATLAB operator, built-in or user-written functions, and control flow statements such as `if`, `while`, and `end`. The scope of variables in the **Initialization commands** field is local; variables defined in the MATLAB workspace are not accessible.

Each command in the **Initialization commands** field should normally be terminated with a semicolon (;). If you omit the semicolon for a command, the results of the command will be displayed in the MATLAB window whenever the command is executed. This provides a convenient means for debugging the commands.

EXAMPLE 7.4 Configuring Initialization commands

The Spring-mass block icon will need two wheels. To prepare to draw the wheels, create two vectors: one containing the x coordinates of a small circle and the other the y coordinates. These variables will be used in the Icon page to draw the wheels.



Configuring Subsystem Blocks. The blocks in the masked subsystem must be configured to use the variables defined on the **Initialization** and **Parameters** pages. To configure the blocks in the Spring-mass subsystem, select the subsystem, then choose **Edit:Look Under Mask** from the model window menu bar. Double-click the Gain block labeled `Left spring`, and set **Gain** to `k_left`. Likewise, set the value of **Gain** for Gain block `Right spring` to `k_right` and for Gain block `1/mass` to `1/mass`. Set **Initial condition** for `Integrator Velocity` to `x_dot0` and `Integrator Position` to `x0`. The subsystem should now appear as shown in Figure 7.15. Close the subsystem, and save the model.

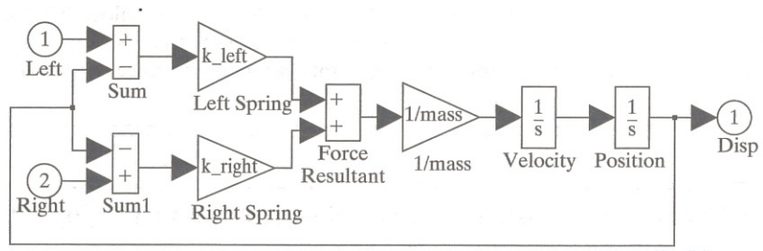


FIGURE 7.15: Spring-mass subsystem configured to use masked block variables

Local Variables. An important difference between masked subsystems and nonmasked subsystems is the scope of variables in the dialog boxes for the blocks in a subsystem. Blocks in nonmasked subsystems may use any MATLAB variable currently defined in the MATLAB workspace. This feature was used to initialize the subsystems in Example 7.1. The blocks in a masked subsystem cannot access variables in the MATLAB workspace; a masked subsystem has its own internal name space that is independent of the MATLAB workspace and all other masked subsystems in a Simulink model. This is an extremely valuable feature of masked subsystems, as it eliminates the possibility of unintentional variable name conflicts.

A masked subsystem's internal variables are created and assigned values using the Mask Editor dialog fields and initialization commands. Each dialog field in a masked block's dialog box defines an internal variable accessible only within the masked subsystem. Additional internal variables may be defined in the **Initialization commands** field of the **Initialization** page.

The connections between the MATLAB workspace and a masked subsystem are the contents of the masked block's dialog box fields. An input field in a masked block's dialog box may contain constants or expressions using variables defined in the MATLAB workspace. The value of the contents of the input field is assigned to the masked subsystem internal variable associated with the input field. This internal variable may be used to initialize a block in the masked subsystem, or it may be used to define another internal variable defined in the **Initialization commands** field.

Consider the model shown in Figure 7.9, with the masked subsystem configured as shown in Figure 7.15. The spring constant for the left spring in each instance of the masked subsystem is `k_left`. However, the contents of `k_left` in each instance is different. `k_left` for the first Spring-mass block should be set (using the Spring-mass block dialog box) to `k1`. `k_left` for the second block should be set to `k2` and for the third Spring-mass block to `k3`.

EXAMPLE 7.5 Using internal variables in a masked subsystem

To illustrate the use of internal variables in a masked subsystem, consider the assignment of a value to the spring constant of the right spring in the cart subsystem of Figure 7.9. Earlier, the contents of the cart subsystem dialog field **Right spring constant** were associated with internal variable `k_right`. In Figure 7.16, we see that the Gain block labeled Right spring is set to `k_right`. Thus, when the M-file script `SetCartParms.m` is executed, the value of the **Gain** for the Gain block `Right spring` in this particular instance of the cart subsystem is set to 2.

EXAMPLE 7.6 Setting an internal variable using a popup list

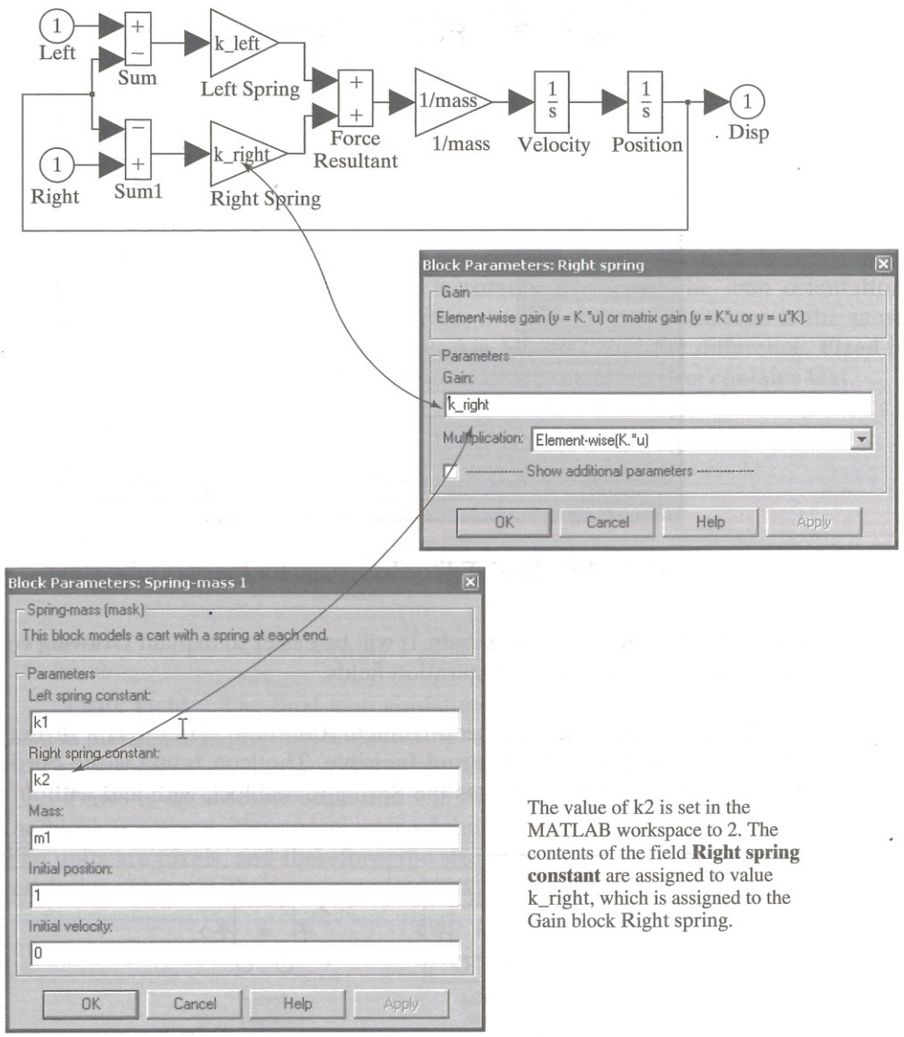
For the masked subsystem in Example 7.3, define a variable `temp_val` as follows:

Menu Choice	temp_val
`Very hot`	120
`Hot`	100
`Warm`	85
`Cool`	70
`Cold`	50

To accomplish this task, place the following statements in the **Initialization commands** field:

```
temp_list = [120,100,85,70,50] ;
temp_val = temp_list(temp_des) ;
```

The first statement creates a vector of temperatures. The second statement uses `temp_des` [the variable associated with the pop-up list in Example 7.3 (Figure 7.13)] as an index into the vector. Choose **Close**, and then, with the masked block still selected, choose **Edit:Look Under Mask**, and set Constant block dialog box field **Constant value** to `temp_val`.



The value of k2 is set in the MATLAB workspace to 2. The contents of the field **Right spring constant** are assigned to value k_right, which is assigned to the Gain block Right spring.

FIGURE 7.16: Right spring constant Gain block initialization

7.3.5 Mask Editor Icon Page

The **Icon** page allows you to design custom icons for masked blocks. The **Icon** page used to create the custom icon for the cart block in Figure 7.7 is shown in Figure 7.17. (Recall that x and y were defined in the **Initialization commands** field in Example 7.4.) The page consists of six fields. **Drawing commands** is a multiple line field in which you may enter one or more MATLAB statements to draw and label the icon. The four fields to the left of **Drawing commands** configure the block icon. The final field, **Command**, is a drop-down list of available MATLAB commands that

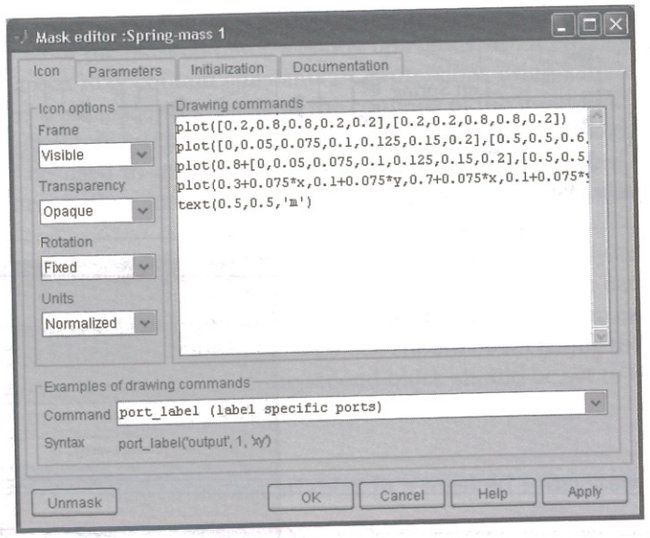


FIGURE 7.17: Mask Editor Icon page for the cart subsystem

may be used in **Drawing commands**. It will be easier to explain **Drawing commands** if we first discuss the icon configuration fields.

Frame Field. The first icon configuration field, **Frame**, is a drop-down list containing two choices: **Visible** and **Invisible**. The icon frame is the border of the block icon. Figure 7.18 illustrates the Spring-mass block with and without the icon frame.

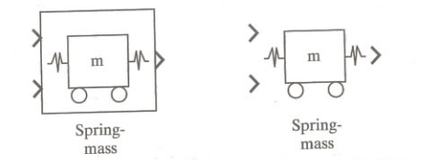


FIGURE 7.18: Icon frame visibility

Transparency Field. The second icon configuration field, **Transparency**, is a drop-down list containing two choices: **Transparent** and **Opaque**. Shown in Figure 7.19 is the Spring-mass block with **Transparency** set to both options. Note that when **Transparent** is selected, the labels on the Inport and Outport blocks in the subsystem underlying the mask are visible. Selecting **Opaque** hides the labels.

Rotation Field. The third icon configuration field, **Rotation**, is a drop-down list containing two choices: **Fixed** and **Rotates**. This field determines the behavior of the block icon when **Format:Flip block** and **Format:Rotate block** are selected.

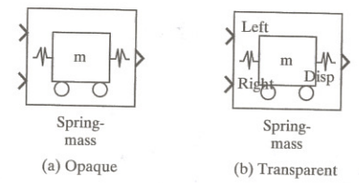


FIGURE 7.19: Icon transparency

If **Fixed** is selected, when the block is rotated or flipped the icon orientation doesn't change. When **Rotates** is selected, the orientation of the icon is the same as the orientation of the block. Illustrated in Figure 7.20 is the difference. **Fixed** is frequently desirable, particularly in cases in which the block icon contains text.

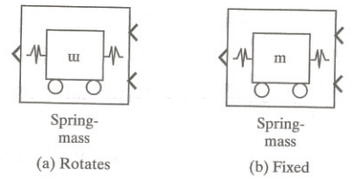


FIGURE 7.20: Icon rotation

Units Field. The final icon configuration field, **Units**, determines the scale used in plotting icon graphics and locating text on the icon. The field is a drop-down list consisting of three choices: **Autoscale**, **Pixels**, and **Normalized**.

Pixels is an absolute scale and will result in an icon that isn't resized when the block is resized. The coordinates of the lower-left corner of the icon are (0,0). The units are pixels, and therefore, the size of the icon will depend on the display resolution. Figure 7.21 shows a masked block with **Units** set to **Pixels**.

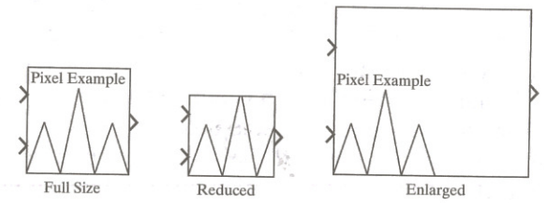


FIGURE 7.21: Units set to Pixels

Autoscale adjusts the size of the icon to exactly fit in the block's frame (even if the frame is invisible). Figure 7.22 shows the masked block in Figure 7.21 reset to **Autoscale**. Note that the text on the icon does not change size when the block is resized.

Normalized specifies that the drawing scale is 0.0 to 1.0 in both the horizontal and vertical axes. The coordinates of the lower left corner of the icon (in its default

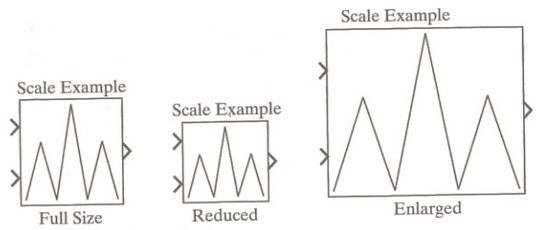


FIGURE 7.22: Units set to Autoscale

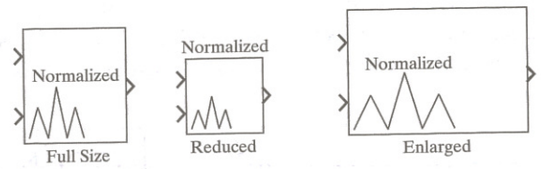


FIGURE 7.23: Units set to Normalized

orientation, not rotated or flipped) are defined to be (0,0), and the coordinates of the upper right corner of the icon are (1,1). When the block is resized, the coordinates are also resized. Text does not change size when the block is resized. Figure 7.23 illustrates a masked block with **Units** set to **Normalized**.

Drawing Commands. Several different MATLAB statements may be entered in Drawing commands to customize a block's icon. Table 7.1 lists these commands.

Three of the commands display text on the icon. The simplest, `disp(string)`, displays `disp(string)` centered on the icon. This command is useful for placing a simple descriptive title in the center of the icon. The `text(x,y,string)` command permits you to locate a string anywhere on the icon, using the icon coordinate system as specified in the Drawing coordinates field. The third text command, `fprintf(string,list)` is identical to the MATLAB `fprintf` statement. (Enter `help fprintf` at the MATLAB prompt for details on `fprintf`.) Using `fprintf`, you can build labels that use variables defined by the block's dialog fields and the statements in the field **Initialization commands** on the **Initialization** page. Embed a new-line character (`\n`) in the string to produce a label with multiple lines. Like the `disp` command, `fprintf` places the label at the center of the icon.

A character string used to display text on the icon may be a literal string, or it may be a MATLAB string variable. A literal string is a sequence of printable characters enclosed in single quotes. For example, to place the label "Special Block" in the center of a block's icon, use the command:

```
disp('Special Block')
```

A string variable is a MATLAB variable that represents a character string instead of a number. MATLAB provides several functions for building and manipulating string variables. These functions can be used in the field **Initialization commands** on the **Initialization** page to create strings for use in the text display

TABLE 7.1: Icon Drawing Commands

Command	Description
`disp(string)`	Display *string* in the center of the icon.
`text(x,y,string)`	Display *string* starting at `(x,y)`.
`text(x,y,string, 'HorizontalAlignment', halign, 'VerticalAlignment', valign)`	Display *string* relative to `(x,y)`. The two alignment options allow you to control the positioning of the text relative to the location `(x,y)`. Either or both option pairs may be used. If option `'HorizontalAlignment'` is specified, parameter *halign* must immediately follow it and must be a string or string variable containing `'left'`, `'right'`, or `'center'`. If option `'VerticalAlignment'` is specified, parameter *valign* must immediately follow it and must be a string or string variable containing `'base'`, `'bottom'`, `'middle'`, `'cap'`, or `'top'`.
`fprintf(string,list)`	Display the results of the `fprintf` statement at the center of the icon.
`plot(x_vector,y_vector)`	Draw a plot on the icon.
`dpoly(num,denom)`	Display a transfer function centered on the icon.
`dpoly(num,denom'z')`	Display a discrete transfer function in ascending powers of z.
`dpoly(num,denom'z-')`	Display a discrete transfer function in descending powers of z.
`droots(zeros,poles, gain)`	Display a transfer function in zero-pole-gain format.
`port_label(type, number, label)`	Display a label on a port. *type* must be `'input'` or `'output'`. *number* is the input or output port number. *label* is a text string or string variable containing the desired port label. Note that if there are any icon drawing commands, the port labels on the subsystem input and output ports are not displayed. A separate `port_label` command must be used for each port for which a label is desired.
`image(object, [x, y, width, height])`	Display image at position `(x, y)` relative to the lower-left corner of the icon.
`image(object, [x, y, width, height])`	Display image at position `(x, y)` relative to the lower-left corner of the icon.
`image(object, [x, y, width, height], rotation)`	The additional parameter *rotation* must be a string or string variable containing either `'on'` or `'off'`. If `'on'`, the image rotates if the block is rotated. If `'off'` (the default), the image does not rotate.

TABLE 7.1: Icon Drawing Commands (Cont)

Command	Description
`patch (x, y)`	Draws a closed polygon with vertices defined by the *x* and *y* vectors. The *x* and *y* vectors must be of the same length and must have at least three elements. The polygon will be drawn in the current foreground color.
`patch (x, y, [red green blue])`	Overrides the current foreground color with specified values of red, green, blue.
`color(color)`	Set the drawing color for subsequent commands to the specified color.

commands. A particularly useful string function is `sprintf`. `sprintf` is similar to `fprintf`, but it writes to a character string instead of the screen or a file. An excellent reference for a detailed discussion of MATLAB string variables and functions is the text by Hanselman and Littlefield [1].

EXAMPLE 7.7 Displaying a parameter value on a block icon

Suppose we wish to change the icon shown in Figure 7.9 such that the cart mass is displayed on the icon just above the wheels, and the units of mass are kilograms. Add the following command to the **Initialization commands** field on the **Initialization page** of the Mask Editor:

```
b_label=sprintf('%1.1f kg',mass);
```

Then, enter the following command in the **Drawing commands** field of the **Icon** page:

```
ext(0.25,0.4,b_label);
```

The block icon will be as shown in Figure 7.24.

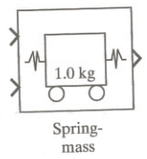


FIGURE 7.24: Displaying the cart mass on its icon

The `plot(x_vector,y_vector)` command displays graphics on the block icon. This command is similar to the MATLAB `plot` command but has fewer options. In particular, the Mask Editor `plot` command does not support options to set line styles or colors and will not plot two-dimensional arrays. The command expects pairs of vectors specifying sequences of *x* and *y* coordinates. There may be more than one pair of vectors in a single plot command, and there may be more than one plot command for an icon. Figure 7.17 shows the commands used to create the cart icon.

EXAMPLE 7.8 Drawing on a block icon

Suppose we wish to display sine and cosine functions on an icon. Placing the following commands in the **Initialization commands** field will produce the necessary vectors:

```
x_vector = [0:0.05:1];
y_sin = 0.5 + 0.5*sin(2*pi*x_vector) ;
y_cos = 0.5 + 0.5*cos(2*pi*x_vector) ;
```

Next, place the following command in **Drawing commands**:

```
plot(x_vector,y_sin,x_vector,y_cos) ;
```

The block will appear as shown in Figure 7.25.

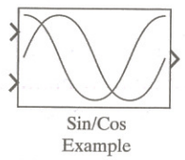


FIGURE 7.25: Drawing curves on a block icon

EXAMPLE 7.9 Drawing logic gate icons

The Simulink Logical Operator block can be configured to implement AND and OR gates, but the block icon is a rectangle with the logical operator it implements displayed on the icon. By placing copies of the Logical Operator block into masked subsystems, we can produce AND and OR blocks that more closely resemble the conventional icons for these gates.

To produce the AND gate block, start with a Logical Operator block configured to implement AND. Select the block using a bounding box, then choose **Edit:Create subsystem** from the model window menu bar. Select the subsystem, then choose **Edit:Mask Subsystem** from the model window menu bar. Open the Mask Editor, and place the following command in the **Initialization commands** field on the **Initialization** page:

```
t=-pi/2:0.1:pi/2;
```

Change to the **Icon** page, and set **Frame** to **Invisible** and **Units** to **Normalized**. Place the following commands in **Drawing commands**:

```
plot([0.5,0,0,0.5],[0,0,1,1],0.5+0.5*cos(t),0.5+0.5*sin(t));
port_label('input', 1, 'a');
port_label('input', 2, 'b');
port_label('output', 1, 'ab');
```

To produce the OR gate block, the process is similar, using the following in **Drawing commands**:

```
plot([0,0],[0,1],t,0.5*t.^2,t,1-0.5*t.^2);
port_label('input', 1, 'a');
port_label('input', 2, 'b');
port_label('output', 1, 'a+b');
```

The logic gate blocks that result are shown in Figure 7.26.

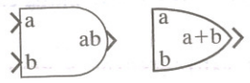


FIGURE 7.26: Logic gate masked blocks

The drawing commands `image(object)`, `image(object, [x, y, width, height])`, and `image(object, [x, y, width, height], rotation)` allow you to place color images on block icons. *object* is a MATLAB image usually read using the MATLAB command `imread(ImageFile)`, where *ImageFile* is a file containing an image in a format compatible with MATLAB.

EXAMPLE 7.10 Displaying a .tif image on a block icon

Suppose we wish to display a photograph of an aircraft in tagged image format, `phantom.tif`, on a block icon for a masked subsystem that models the aircraft flight dynamics. Assuming the file phantom.tif is in the current MATLAB path, we can place the following statement in **Drawing commands**:

```
image(imread('phantom.tif'));
```

The icon will appear as shown in Figure 7.27. Note that the image is not stored with the Simulink model—it must be loaded from the file each time the model is opened.

FIGURE 7.27: Placing an image on a block icon

The drawing commands `dpoly(num,denom)` and `droots(zeros,poles, gain)`, display a transfer function on the blocks icon.

`dpoly` displays the transfer function in polynomial form. The arguments *num* and *denom* are vectors containing the coefficients of the transfer function numerator and denominator in descending powers of s. `dpoly` will also display discrete transfer functions in descending powers of z or ascending powers of $1/z$. To display the transfer function in descending powers of z, use the command `dpoly(num,denom,'z')`. To display the transfer function in ascending powers of $1/z$, use `dpoly(num,denom,'z-')`.

`droots` displays a transfer function in factored pole-zero form. *zeros* is a vector containing the zeros of the transfer function (roots of the numerator), and *poles* is a vector containing the poles of the transfer function (roots or the denominator). *gain* is a scalar.

7.3.6 Looking Under and Removing Masks

There are two additional masking commands that permit you to view a subsystem underlying a mask and to delete the mask.

To examine the subsystem underlying a masked block, select the masked block, then choose **Edit:Look under mask**.

To convert a masked block into an unmasked block or subsystem, select the block, then open the Mask Editor. Press the **Unmask** button at the bottom of the Mask Editor. If you change your mind about removing the mask, select the block and choose **Edit:Mask Subsystem**. The masking information will be preserved until you close the model. Once you close the model after removing the mask, it is not possible to restore the mask.

7.3.7 Using Masked Blocks

Once you have created a masked block, it can be copied to a model window in a manner identical to that used to copy a block from a Simulink block library. For example, to build the model shown in Figure 7.9(a), open a new model window, then drag three copies of the Spring-mass block to the new model window. Add the Constant and Scope blocks, and connect the blocks with signal lines as shown. Configure the blocks as shown in Table 7.2

Now the model is complete. Configure the Scope block and simulation parameters as in Example 7.1, then save the model. In the MATLAB workspace, run the

TABLE 7.2: Cart Model Configuration Parameters

Field	Spring-mass 1	Spring-mass 2	Spring-mass 3
Left spring constant	k1	k2	k3
Right spring constant	k2	k3	0
Mass	m1	m2	m3
Initial position	1	0	0
Initial velocity	0	0	0

script M-file (`set_x4a.m`) to assign values to the spring constants and masses. This model will produce results identical to those of the model in Example 7.1.

7.3.8 Creating a Block Library

A *block library* is a special Simulink model that serves the same purpose as a subroutine library in a conventional programming language. When a block is copied from a block library to a model, the copy remains linked to the version of the block in the block library. If the version of the block in the block library is changed, the change is effective each place the block is used. A block in a block library is called a *library block*. A copy of library block in a model is called a *reference block*. Each reference block has its own data (the dialog box fields), but the functionality is defined by the library block.

A reference block cannot be changed. So, for example, if you select a reference block for which the library block is a masked subsystem, then choose **Edit** from the model window menu bar (the **Edit:Edit Mask** menu choice will not be present). **Edit:Look Under Mask** will be present. But if you look under the mask, and then try to edit the underlying subsystem, Simulink will display an error message.

Creating a Block Library. Create a block library by selecting **File:New:Library** from any model window menu bar or the Simulink block library menu bar. (Recall that you can open the Simulink block library by right-clicking the Simulink Library icon in the Simulink Library Browser window and then clicking the single menu choice Open the 'Simulink' Library.) Copy the desired blocks to the new library, and then save the block library. Once the library has been saved, the blocks in the library are library blocks, and blocks copied from the library are reference blocks.

You can create nested block libraries by adding subsystems to a block library window. For example, each of the block libraries (Sources, Sinks, etc.) in the Simulink block library is a subsystem that contains a set of unconnected blocks. To create a subsystem block library, drag an empty Subsystem block to the block library window. Open the Subsystem block, add the desired blocks to the subsystem window, and delete the default inport and outport blocks and signal lines. A subsystem library can be masked, as are the block libraries in the Simulink block library. The masked subsystem can have a custom icon, but it must not have any dialog box fields (the **Parameters** page should be empty), and the **Block description** field on the **Documentation** page must be blank. Otherwise, double-clicking the block library icon would open a block dialog box, instead of opening a subsystem window from which to copy blocks.

The final step in creating a block library is to add the directory in which the library is stored to the MATLAB path. Reference blocks can find only library blocks that are in the MATLAB path or the current directory (the current MATLAB directory).

Modifying a Block Library. Once a block library has been saved and closed, it is locked and cannot be changed. If it is necessary to change a block library, either to add more blocks or to edit a block, the block library must be unlocked. To unlock

a block library, choose **Edit:Unlock Library** from the block library window menu bar. The library will be unlocked as long as it remains open. To relock a block library, close the library, then reopen it.

When a block in a library is changed, the change will be applied to each corresponding reference block when the model containing the reference block is opened or run, or when **File:Update Diagram** is selected from the model window menu bar.

To quickly find the library block corresponding to a reference block, select the reference block, and then choose **Edit:Link Options:Go to Library Block** from the model window menu bar.

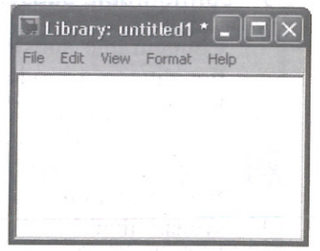
Unlinking a Block. A reference block can be converted into a normal block by breaking the link to the library block. To remove the link, select the block, and then choose **Edit:Link Options:Disable Link** from the model window menu bar. Breaking a link affects only the instance of the reference block that was selected. The library block and all other corresponding reference blocks are unaffected.

Troubleshooting Links. If Simulink is not able to find a library block corresponding to a reference block, the reference block is displayed with a red dashed border, and an error message is displayed. To correct the problem, delete the reference block from the model and then reinstall the reference block. An alternative is to double-click the reference block. A dialog box will appear, prompting you for the path to the library block.

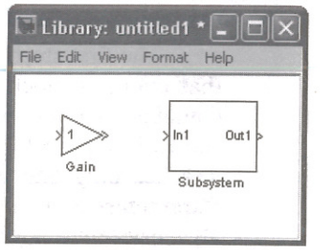
EXAMPLE 7.11 Creating a custom block library

In this example, we will create a custom block library. The library will contain a Gain block and a block library containing the Spring-mass block.

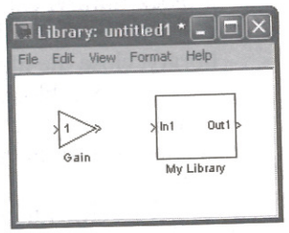
Open a new library window by choosing **File:New:Library**.



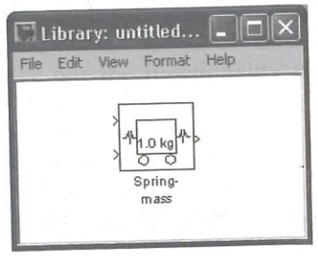
Copy a Gain block from the Math Operations block library and a Subsystem block from the Ports & Subsystems block library.



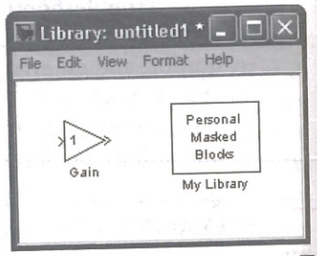
Label the subsystem `My Library`.



Double-click subsystem `My Library`, opening the subsystem window. Delete the Inport and Outport blocks and signal line, then drag a copy of the Spring-mass block to the subsystem window, and rename the block `Spring-mass`. Then, close the subsystem window. Select the subsystem and choose **Edit:Mask Subsystem** from the model window menu bar.



In the **Drawing commands** field on the **Icon** page, enter `disp('Personal \nMasked\nBlocks') ;`. Leave all other fields in the Mask Editor blank. Choose **OK**. Resize My Library to fit the block icon. Save the model using the name `PersonalLibrary`. Then, close the subsystem window. Select the subsystem and choose **Edit:Mask Subsystem** from the model window menu bar.



7.3.9 Configurable Subsystems

Configurable subsystems provide a convenient means of adding optional functionality to a model. For example, suppose we wish to implement the automobile model of Figure 7.1 such that the user can choose proportional control or proportional-derivative (PD) control. We could place both controllers in a Configurable Subsystem and then choose between the two controllers before running the simulation. To illustrate the procedure, we will build a configurable subsystem that has two inputs, (a and b) and a single output c. The subsystem is to be configurable to produce either $c = a + b$ or $c = k * a$, where k is a parameter.

The first step in building a configurable subsystem is to build a block library that contains blocks that provide the desired functions. In this case, the library contains two subsystems. The library and the subsystem blocks it contains are shown in Figure 7.28. Notice that subsystem `a + b` has two inputs, and subsystem `k*a` has only one input. `k*a` is masked and has a single prompt **Gain:** on the **Parameters** page and corresponding variable `k`. Save the block library after adding all subsystems.

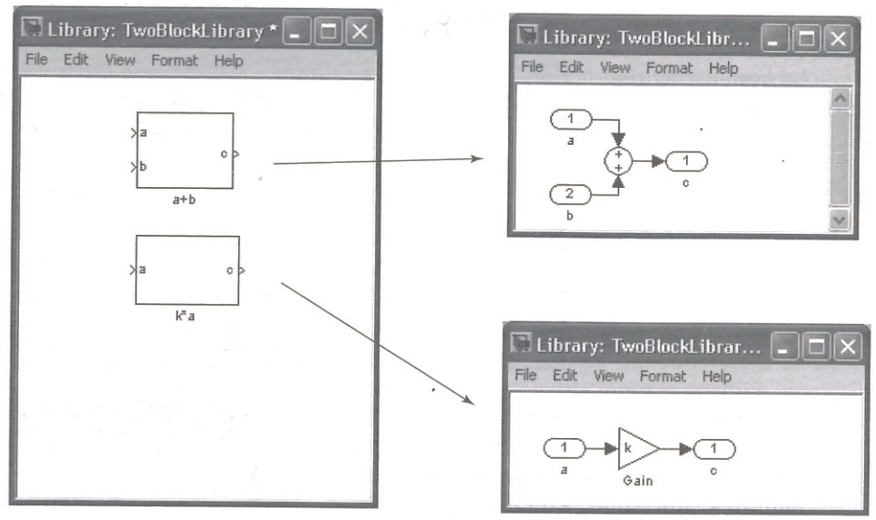
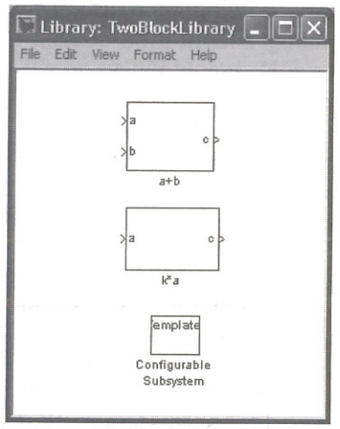
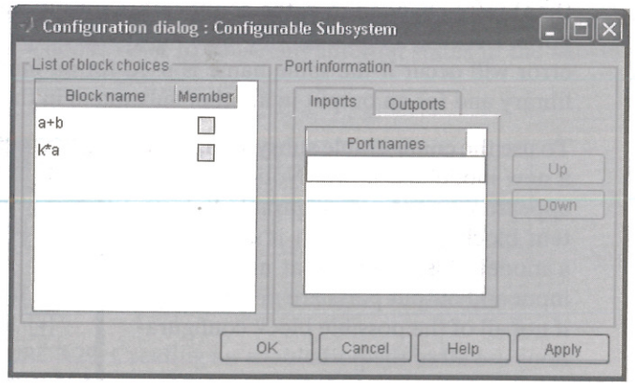


FIGURE 7.28: Block library for configurable subsystem

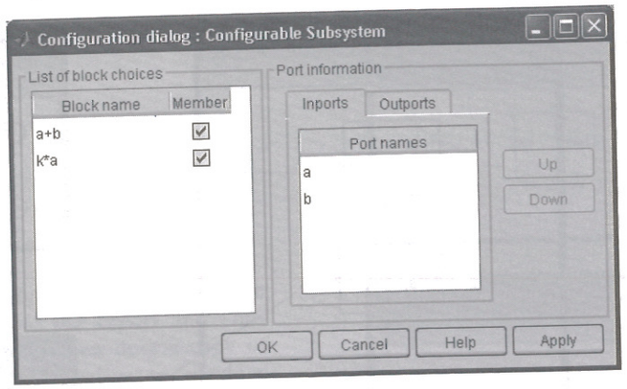
Drag a Configurable Subsystem block from the Signals & Systems block library to the new block library window.



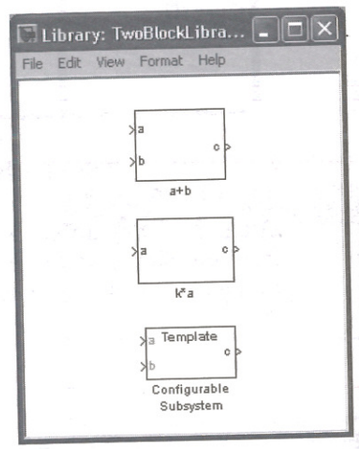
Double-click the Configurable Subsystem block. The Configuration dialog box will appear as shown. Notice that there is a check box for each subsystem in the block library.



Click both check boxes to add the subsystems to the configurable subsystem.

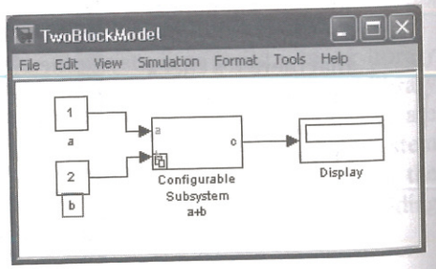


Click **OK**. After resizing, the Configurable subsystem displays the port labels. This completes the construction of the configurable subsystem. You can build additional configurable subsystems in a block library by adding Configurable Subsystem blocks and configuring them as needed.



When you create a set of subsystems for a block library that will be used with configurable subsystems, you must pay special attention to the naming of the input and output ports on the various subsystems. The configurable subsystem block will have one input port for each input port name in the block library and one output port for each output port name. In this example, the same names were used for both subsystems in the block library. But if the output port on the second subsystem (`k*a`) were changed to `d`, the configurable subsystem block would have two output ports. Because a block cannot have input and output ports with the same name, an error will occur if the same name is used for an input in one subsystem in a block library and for an output in another subsystem in the same block library.

To use the configurable subsystem, build a new model as shown. Notice that the appearance of the Configurable Subsystem block changes when it's copied to a model. Also notice that all available input and output ports are present, even if not all of the possible block configurations use all of the ports.



To choose a subsystem, select the Configurable Subsystem block, and choose **Edit:Block** choice. All available subsystems are listed in the drop-down list. The **Edit** menu also provides options for setting subsystem parameters and properties. If you double-click the Configurable Subsystem block, it will open the currently selected subsystem or, if masked, the masked block dialog box.

7.4 CONDITIONALLY EXECUTED SUBSYSTEMS

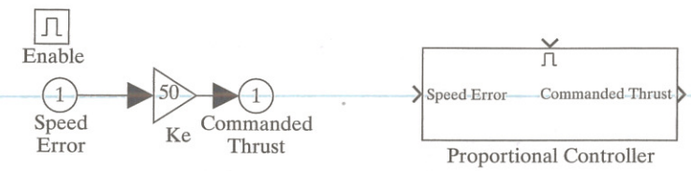
The Ports & Subsystems block library provides two blocks that cause subsystems to execute conditionally. The Enabled subsystem block causes a subsystem to execute only if a control input is positive. The Triggered subsystem block causes a subsystem to execute once when a trigger signal is received. Placing both the Enabled subsystem and Triggered subsystem blocks in a subsystem causes the subsystem to execute once when a trigger signal is received, only if an enable input is positive.

7.4.1 Enabled Subsystems

Enabled subsystems provide a means for modeling systems that have multiple operating modes or phases. For example, the aerodynamics of a jet fighter in the landing configuration are quite different from the aerodynamics of the same airplane in supersonic flight. The digital flight control system for such an airplane likely uses different control algorithms in the different flight regimes. A Simulink model of the airplane and its control system might need to include both flight regimes. It is possible to model such a system using logic blocks or switch blocks. However, that would require every block in the model to be evaluated each simulation time step, including blocks not currently contributing to the system's behavior. If we convert the various flight dynamics and control algorithm subsystems into enabled subsystems, only the subsystems that are active during a particular simulation step will be evaluated during that step. This can provide a significant computational savings.

A subsystem is converted into an enabled subsystem by adding an Enable block from the Ports & Subsystems block library to the subsystem. The Enabled Subsystem block provides a subsystem preconfigured with an Enable block. In Figure 7.29, a simple proportional controller converted into an enabled subsystem is illustrated.

The Enable block's dialog box is shown in Figure 7.30. The dialog box has three fields. The first field, **States when enabling**, is a drop-down menu with two options: **reset** and **held**. Choose **reset** to cause any internal states in the subsystem



(a) Enabled subsystem (b) Enabled subsystem block

FIGURE 7.29: Creating an enabled subsystem

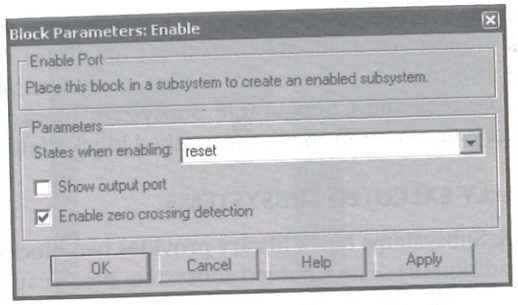


FIGURE 7.30: Enable block dialog box

to be reset to the specified initial conditions each time the block is enabled. If you choose **held**, when the block is reenabled, it will resume with all internal states at the values they held when the block was last executed. The second field, **Show output port**, is a check box. When selected, the Enable block will have an output port. This output port passes through the signal received at the Enable input port when the block is enabled. The third field, **Enable zero crossing detection** is a check box used to enable or disable zero crossing detection for the enable signal.

It is also important to configure the Outport blocks of an enabled subsystem. The dialog box for the Outport block (Figure 7.31) has three fields. The first field, **Port number**, determines the order in which the ports are displayed on the subsystem block icon. The second field, **Output when disabled**, is a drop-down menu with two options: **reset** and **held**. Choose **reset** to cause the output to reset to the value in the third field, **Initial output**. Choose **held** to cause the output to remain at the last value output before the subsystem was disabled.

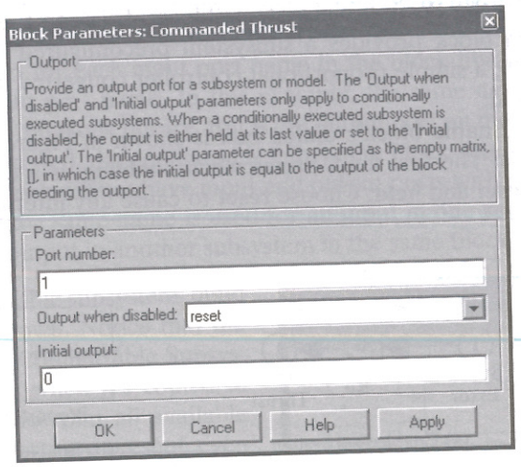


FIGURE 7.31: Enabled subsystem Outport block dialog box

An enabled subsystem is enabled when the signal at the Enable input port is positive. The input signal may be a scalar or a vector. If the signal is a vector, the subsystem is enabled if any element of the vector is positive.

EXAMPLE 7.12 Using an enabled subsystem

To illustrate the use of enabled subsystems, suppose we wish to modify the automobile speed control of Example 6.6 such that it has two modes of operation depending on the speed error. If the absolute value of speed error

$$v_{err} = |\dot{x}_{desired} - \dot{x}|$$

is less than a threshold value, say 2 ft/sec, and the absolute value of the rate of change speed error, $\dot{v}_{err}$, is also less than a threshold value, say 1 ft/sec^2, we wish to switch to proportional-integral (PI) control. Once PI control is enabled, it is to remain enabled as long as v_{err} is less than a larger threshold value, 5 ft/sec. Otherwise, proportional control is to be enabled.

The Simulink model is shown in Figure 7.32. The mode selector subsystem, shown in Figure 7.33, produces two outputs. `Choose PI` is set to 1.0 if the conditions for PI control are satisfied, and 0.0 otherwise. `Choose P` is always the logical inverse of `Choose PI`.

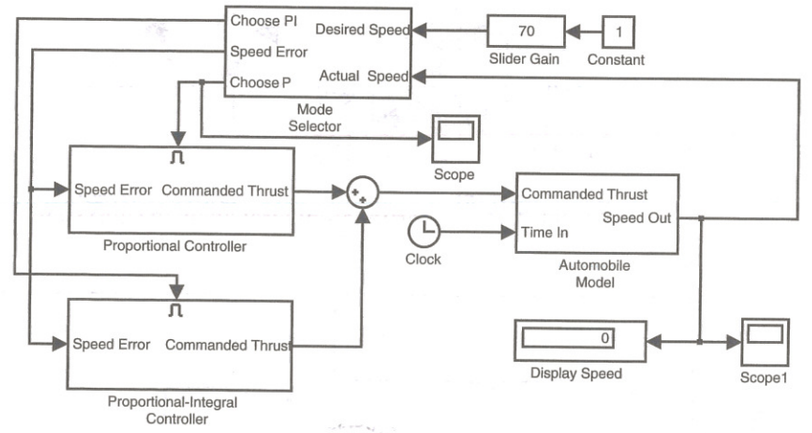


FIGURE 7.32: Simulink model of automobile with a dual-mode speed control

The proportional controller subsystem is illustrated in Figure 7.29. The PI subsystem block is shown in Figure 7.34. The Enable block and Outport block for both controller subsystems are configured to reset.

Executing the simulation results in the speed trajectory shown in Figure 7.35(a). The Choose P signal for this simulation is plotted in Figure 7.35(b). Initially, proportional control is enabled. Because the rate of change of speed error is less than the threshold value, as soon as the speed error decreases below 2 ft/sec, PI control is enabled.

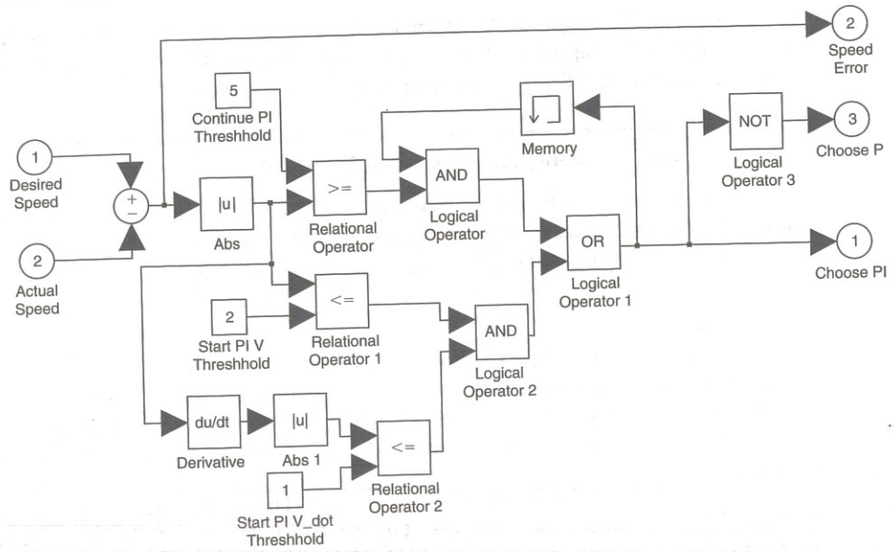


FIGURE 7.33: Mode selector subsystem

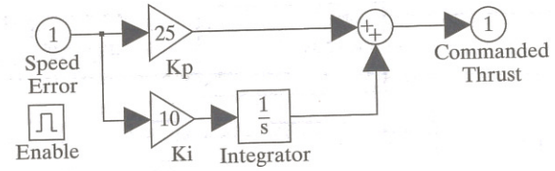
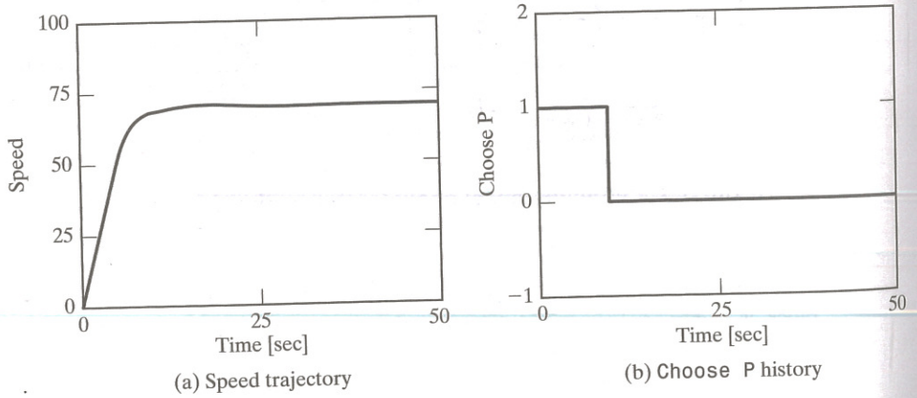


FIGURE 7.34: Proportional-integral enabled subsystem

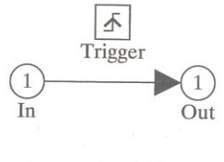


(a) Speed trajectory (b) `Choose P` history

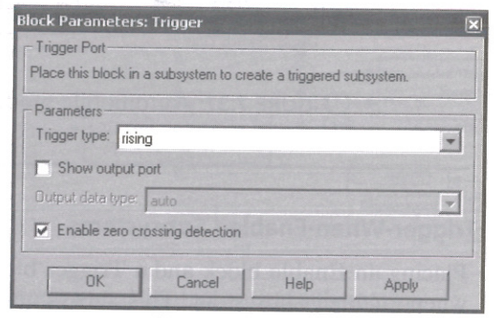
FIGURE 7.35: Speed trajectory and `Choose P` signal history for dual-mode controller model

7.4.2 Triggered Subsystems

A triggered subsystem is executed once each time a trigger signal is received. A triggered subsystem and the trigger dialog box are shown in Figure 7.36. The first field, **Trigger type**, is a drop-down menu with three choices: `rising`, `falling`, and `either`. If `rising` is selected, a trigger signal is defined as the trigger input crossing zero while increasing. If `falling` is selected, the trigger signal is defined as the trigger input crossing zero decreasing. If `either` is selected, the trigger signal is defined as the trigger input crossing zero increasing or decreasing. The second field in the Trigger block dialog box, **Show output port**, is a check box. If the box is checked, the Trigger block will have an output port that passes through the trigger signal. Field **Output data type** is a drop-down menu that is available only if an output port is enabled. The menu can be set to `Auto`, `double`, or `int8`. If `Auto` is selected (the default choice), the data type will be determined by the block to which the output port is connected. The final field is check box **Enable zero crossing detection**, which enables zero crossing detection for the selected block only.



(a) Triggered subsystem



(b) Trigger dialog box

FIGURE 7.36: From Workspace block and dialog box

A triggered subsystem holds its output value after the trigger signal is received. The initial output value of a triggered subsystem is set using the subsystem's Outport blocks.

The trigger signal may be a scalar or a vector. If the signal is a vector, the subsystem is triggered when any element of the vector satisfies the **Trigger type selection**.

EXAMPLE 7.13 Using a triggered subsystem

The triggered subsystem illustrated in Figure 7.36 passes its input to its output when a trigger signal is received and holds its output at that value until another trigger signal is received. The Outport block is initialized to 0. In Figure 7.37, we added this subsystem to the automobile speed control model and routed the subsystem output to a Display block from the Sinks block library. The trigger input is connected to the enable signal for the PI controller. The Display block displays 0 until the PI controller is activated, and afterwards, it displays the most recent activation time.

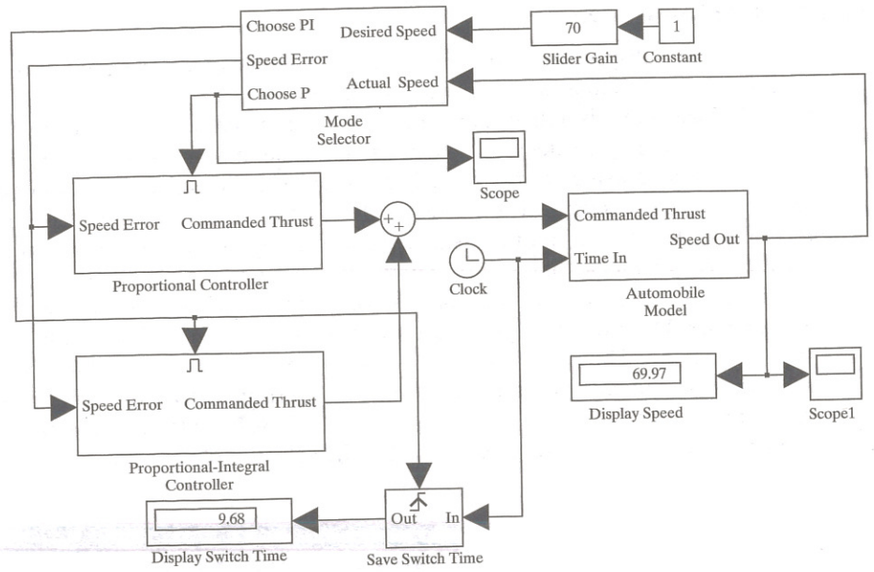


FIGURE 7.37: Automobile model with triggered subsystem

7.4.3 Trigger-When-Enabled Subsystems

Placing an Enable block and a Trigger block in a subsystem produces a triggered-when-enabled subsystem. This subsystem will have both a trigger input and an enable input. The subsystem behaves the same as a triggered subsystem, but the trigger signal is ignored unless the enable signal is positive.

7.4.4 Discrete Conditionally Executed Subsystems

All three types of conditionally executed subsystems may contain continuous blocks, discrete blocks, or continuous and discrete blocks. Discrete blocks in an enabled subsystem execute based on their sample time. They use the same time reference as the rest of the Simulink model; time in the subsystem is referenced to the start of the simulation and not the activation of the subsystem. Consequently, an output dependent upon a discrete block won't necessarily change at the instant the subsystem is enabled.

Discrete blocks in triggered subsystems must have their sample times set to -1, indicating that they inherit their sample time from the driving signal. Note that discrete blocks that include time delays (z^{-1}) change state once each time the subsystem is triggered.

7.5 LOGICAL SUBSYSTEMS

Simulink provides two types of logical conditional subsystems: if subsystems and switch-case subsystems. Both types of logical subsystems employ two blocks: test

blocks and action subsystem blocks. The test blocks perform specified logical tests on one or more input signals and activate one action subsystem block depending on the test result. Thus, at any instant, at most one of the action subsystems associated with a test block is active, and the rest are dormant.

The action subsystems are standard subsystems with an additional action port and corresponding action port block. An action subsystem is built in a manner identical to an enabled subsystem.

EXAMPLE 7.14 Using an if subsystem

In this example, we will modify the dual-mode automobile speed control to use an if subsystem. Proportional control will be enabled if the speed error is more than five miles per hour, and PI control will be enabled otherwise. The top-level Simulink model is shown in Figure 7.38. Notice that there is one if block and two if action subsystem blocks. The block dialog box for the if block is shown in Figure 7.39.

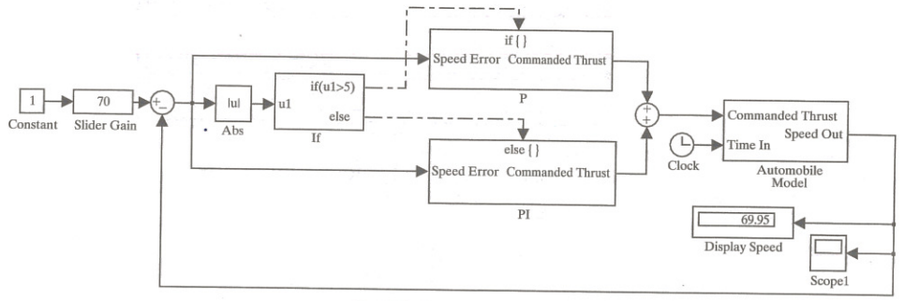


FIGURE 7.38: Dual-mode speed control with if subsystem

The proportional control if action subsystem is shown in Figure 7.40. Notice that the subsystem has an additional action port block.

7.6 ITERATIVE SUBSYSTEMS

Iterative subsystems provide a means with which to add simple control flow blocks to Simulink models. For complex control flow problems, Stateflow is more appropriate. There are two types of iterative subsystems: For Iterator subsystems and While Iterator subsystems. A For Iterator subsystem executes a specified number of iterations each time the subsystem is activated. A While Iterator subsystem executes repeatedly until the condition input is false. For Iterator subsystems and While Iterator subsystems are found in the Ports & Subsystems block library.

7.6.1 For Iterator Subsystem

Figure 7.41(a) shows the For Iterator block contained in the For Iterator Subsystem from the Ports & Subsystems block library. The For Iterator block controls the number of iterations each time the subsystem is invoked. The For Iterator block dialog box is shown in Figure 7.41(b). Set field **Source of number of iterations** to

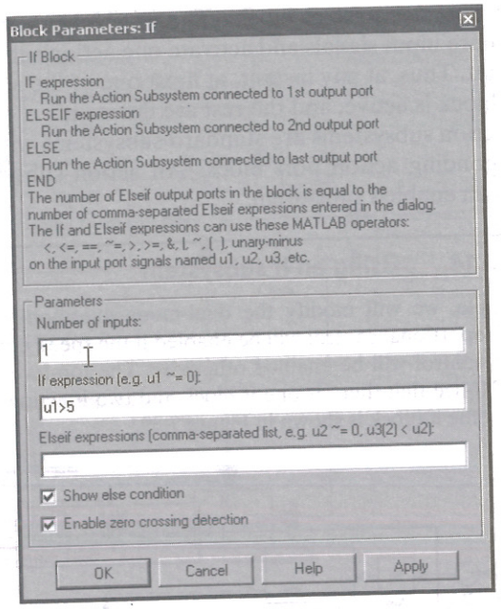


FIGURE 7.39: If block dialog box

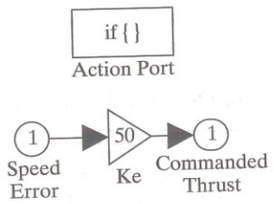


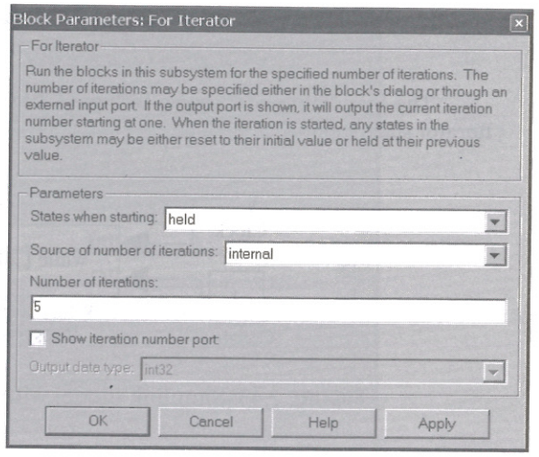
FIGURE 7.40: Proportional control if action subsystem

`internal` to specify the number of iterations in field **Number of iterations** or to `external` to add a number of iterations input port to the subsystem and For Iterator block.

7.6.2 While Iterator Subsystem

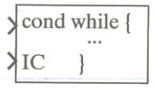
The While Iterator subsystem is similar to the For Iterator subsystem. The While Iterator block [Figure 7.42(a)] can be configured via the block dialog box [Figure 7.42(b)] to implement while behavior or do-while behavior. If while behavior is implemented, and the value at the initial condition port is zero, no iterations will be performed, and the block output will be as determined by the subsystem output port configuration. If the first iteration is executed, it will continue until the signal entering the `cond` input port is zero or the specified maximum number of iterations are complete. Selecting the do-while configuration forces the first iteration to occur.

For Iterator

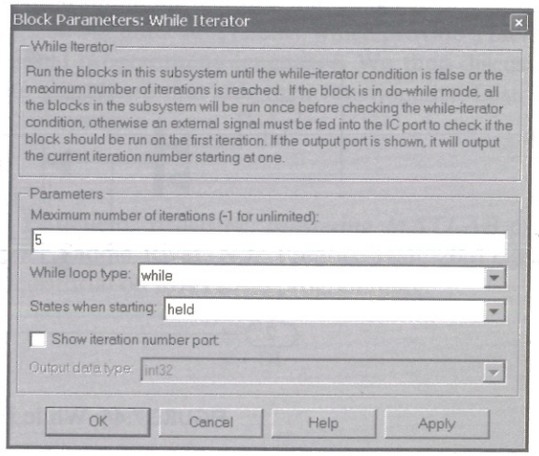


(a) For Iterator Subsystem (b) For Iterator block dialog box

FIGURE 7.41: For Iterator subsystem



While Iterator



(a) While Iterator Subsystem (b) While Iterator block dialog box

FIGURE 7.42: While Iterator subsystem

EXAMPLE 7.15 Using the While Iterator Subsystem

To illustrate the use of iterative subsystems, suppose a block output is an implicit function of the block input.

$$yu + \exp y - 3 = 0$$

where u is the input signal, and y is the output signal. We can compute the output by solving the implicit function iteratively using a Newton–Raphson technique. (For details, see Section 13.3.1.) For each iteration, we compute the new estimate of the output y^+ as a

function of the previous estimate y^- and the input:

$$y^+ = y^- - \left(y^- u + \exp y^- - 3\right) / \left(u + \exp y^-\right)$$

The Simulink model shown in Figure 7.43 implements this algorithm using a While Iterator subsystem.

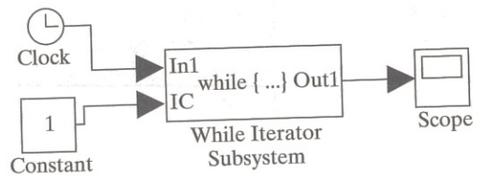


FIGURE 7.43: Simulink model with While Iterator Subsystem

The While Iterator subsystem (Figure 7.44) contains two Function blocks. Function block `Update` computes the next value of y using the expression

```
u(2) - (u(1)*u(2) + exp(u(2)) - 3)/(u(1) + exp(u(2)))
```

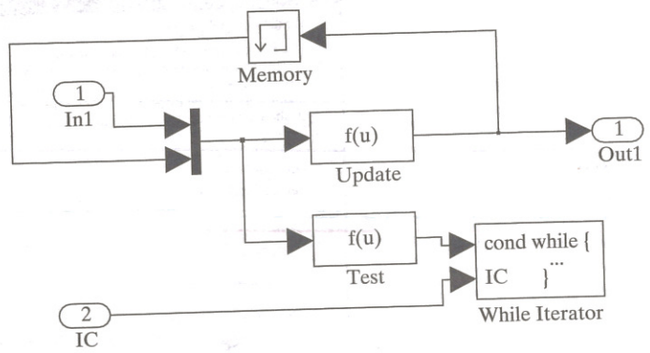


FIGURE 7.44: While Iterator subsystem

and Function block `Test` outputs a value of 1.0 until the error squared falls below a preset threshold (here 10^{-8}) using the expression

```
((u(1)* u(2) + exp(u(2)) - 3) ^ 2) > 1E - 8
```

The purpose of the memory block in the iterative subsystem is to supply the result of the previous iteration (here y^-) to the current iteration. The memory block is initialized to a guess for the output for the initial invocation. For subsequent invocations, the initial value of Memory block output is the final value from the previous invocation.

The While Iterator block dialog box is shown in Figure 7.45. We set field **States when starting** to `Held` so that each time the Iterator subsystem is invoked, the starting guess for the block output is the final value from the previous invocation.

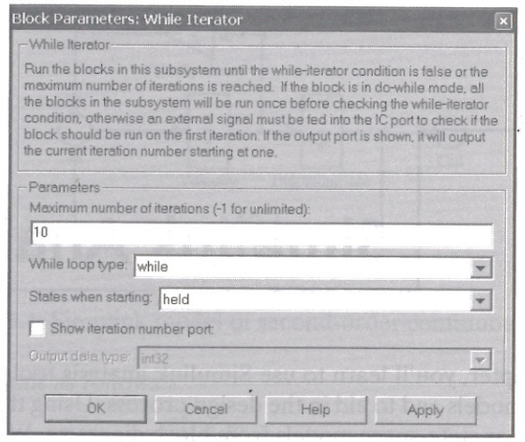


FIGURE 7.45: While Iterator block dialog box

7.7 SUMMARY

In this chapter, we described several Simulink features that make it practical to model complex systems. We discussed Simulink subsystems, which provide a facility similar to subprograms in traditional programming languages. We then discussed masking, which allows us to create subsystems that hide their functionality. Last, we discussed conditionally executed subsystems.

REFERENCE

[1] Hanselman, Duane C., and Littlefield, Bruce R., *Mastering MATLAB 6: A Comprehensive Tutorial* (Upper Saddle River, N.J.; Prentice Hall, 2001). This text provides a comprehensive tutorial on MATLAB programming.