MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

Creating Graphical User Interfaces

Version 6

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

For contact information about worldwide offices, see the MathWorks Web site.

Printing History:			
	November 2000	Online only	New for MATLAB 6.0 (Release12)
	June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
	July 2002	Online only	Revised for MATLAB 6.5 (Release 13)

Contents

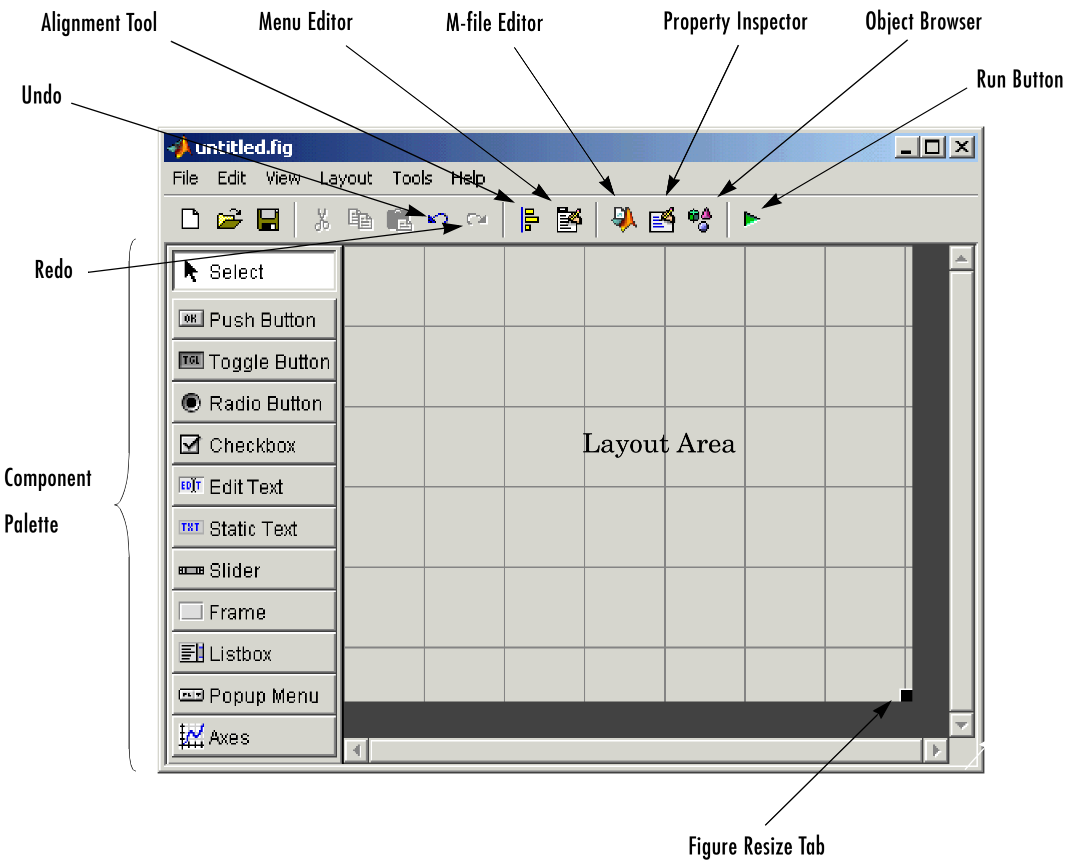
3 MATLAB GUIs

Getting Started with GUIDE

GUIDE — GUI Development Environment

GUIDE, the MATLAB Graphical User Interface development environment, provides a set of tools for creating GUIs. These tools greatly simplify the process of laying out and programming a GUI. This section introduces you to GUIDE and the layout tools it provides.

When you open a GUI in GUIDE, it is displayed in the Layout Editor, which is the control panel for all of the GUIDE tools. The Layout Editor enables you to lay out a GUI quickly and easily by dragging components, such as push buttons, pop-up menus, or axes, from the component palette into the layout area. The following picture shows the Layout Editor.



Once you lay out your GUI and set each component's properties, using the tools in the Layout Editor, you can program the GUI with the M-file Editor. Finally, when you press the **Run** button on the toolbar, the functioning GUI appears outside the Layout Editor window.

GUIDE Toolset

For more information on the full set of GUIDE development tools, see the following sections:

- “Laying Out GUIs — The Layout Editor” on page 4-2 — add and arrange objects in the figure window.
- “Aligning Components in the Layout Editor” on page 4-9 — align objects with respect to each other.
- “Setting Component Properties — The Property Inspector” on page 4-14 — inspect and set property values.
- “Viewing the Object Hierarchy — The Object Browser” on page 4-16 — observe a hierarchical list of the Handle Graphics objects in the current MATLAB session.
- “Creating Menus — The Menu Editor” on page 4-17 — create a menu bar or a context menu for any component in your layout.
- “Setting the Tab Order — The Tab Order Editor” on page 4-27 — change the order in which components are selected by tabbing.

GUI FIG-Files and M-Files

GUIDE stores GUIs in two files, which are generated the first time you save or run the GUI:

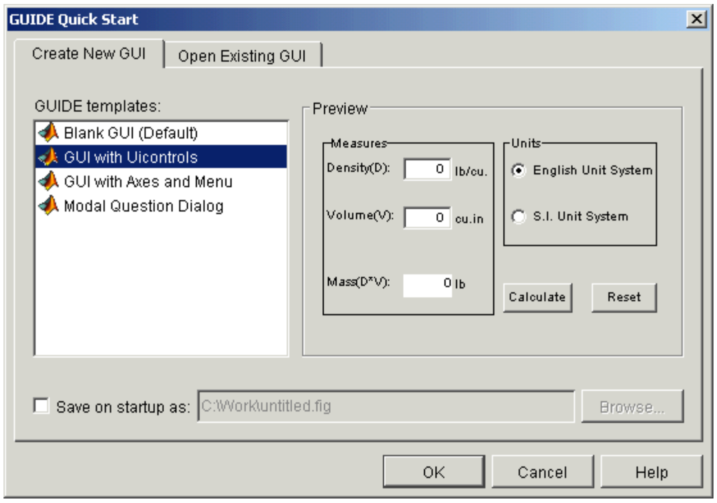
- FIG-file — a file with extension `.fig` that contains a complete description of the GUI figure layout and the components of the GUI: push buttons, menus, axes, and so on. When you make changes to the GUI layout in the Layout Editor, your changes are saved in the FIG-file.
- M-file — a file with extension .m that contains the code that controls the GUI, including the callbacks for its components. This file is referred to as the *GUI M-file*. When you first run or save a GUI from the Layout Editor, GUIDE generates the GUI M-file with blank stubs for each of the callbacks. You can than program the callbacks using the M-file editor.

Note In the documentation for releases prior to Release 13, the GUI M-file was referred to as the “application M-file.”

Using GUIDE Templates

GUIDE provides templates for several basic types of GUIs. You can modify these templates to make your own GUIs. The advantage of using the templates is that you can create GUIs more quickly and easily.

To view the templates, enter `guide` at the MATLAB prompt. This displays the GUIDE Quick Start dialog, as shown in the following figure.



When you select a template in the left pane, a preview of it appears in the right pane. Clicking **OK** opens the template in the Layout editor. See “Using GUIDE Templates” on page 3-6 for more information about GUIDE templates.

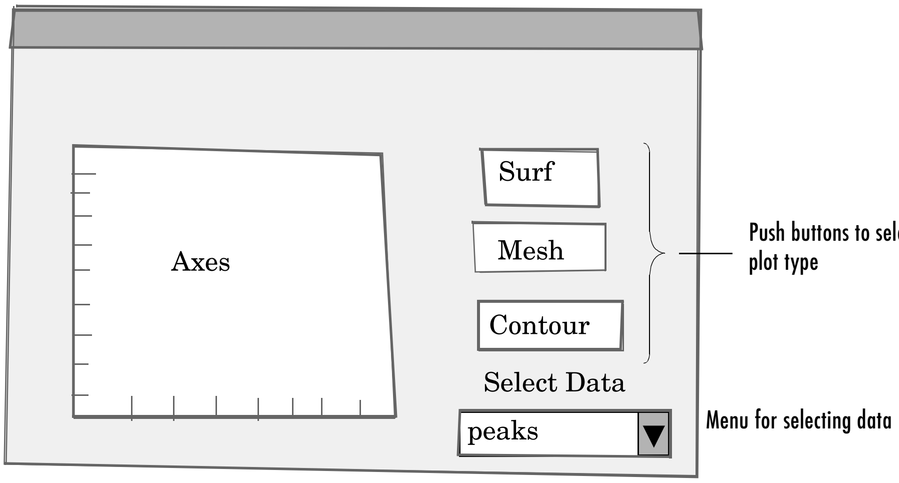
The next section, “Getting Started with GUIDE” on page 1-1, provides a detailed example of how to create a GUI using GUIDE.

Example: Creating a GUI

This section presents an example that shows how to use GUIDE to create graphical user interfaces (GUIs).

Designing the GUI

The GUI used in this example contains an axes that displays either a surface, mesh, or contour plot of data selected from the pop-up menu. The following picture shows a sketch that you might use as a starting point for the design.



The pop-up menu contains three strings — “peaks,” “membrane,” and “sinc,” which correspond to MATLAB functions. The user can select the data to plot from this menu.

Laying Out the GUI

This section illustrates how to lay out GUI components (i.e., user interface controls, such as push buttons, pop-up menus, static text, etc.) in the GUI. We recommend that you create the GUI for yourself, as this is the best way to learn how to use GUIDE.

The section explains how to:

- Open a completed version layout and code for the example
- Open a new GUI in the Layout Editor
- Set the GUI figure size
- Add the components
- Align the objects

Layout and Code for the Example

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Layout Editor with completed GUI layout
- MATLAB Editor with completed M-file

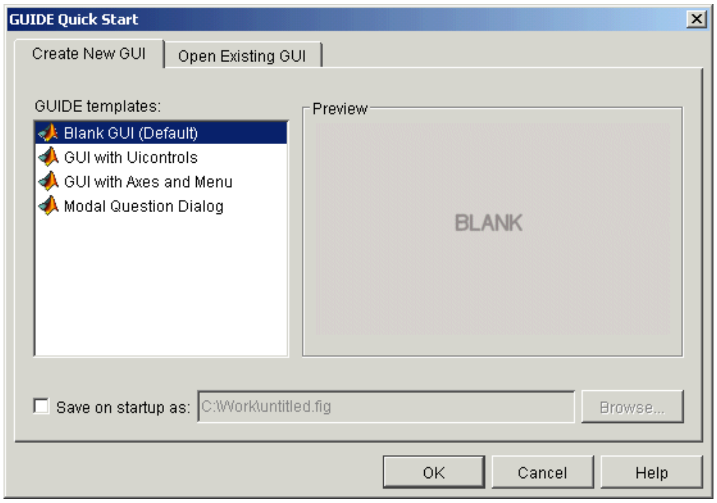
View an Animated Demo

The following link displays an animated version of this example.

Show GUIDE demonstration

Open a New GUI in the Layout Editor

Open GUIDE by typing `guide` at the MATLAB prompt. This displays the **Quick Start** dialog shown in the following figure:

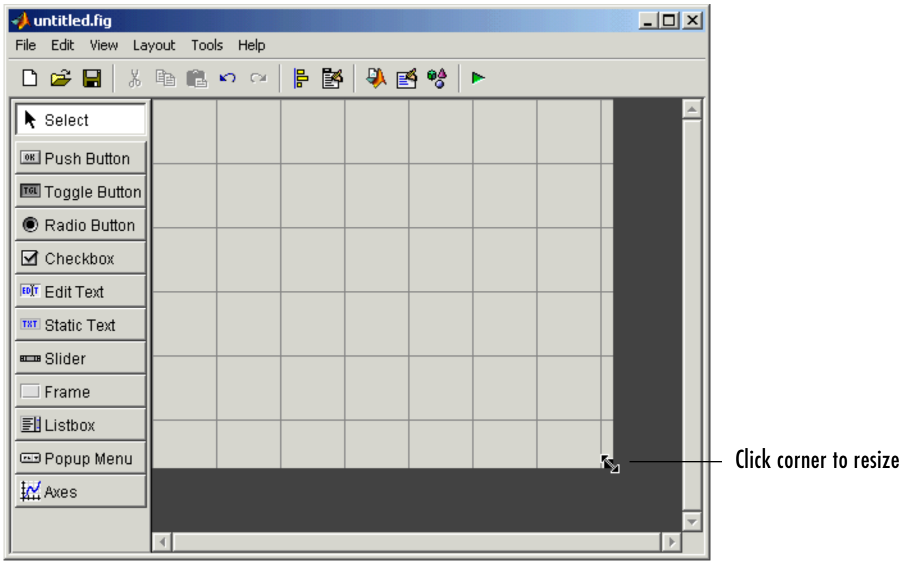


If GUIDE is already open, you can display a similar dialog, without the **Open Existing GUI** tab, by selecting **New** from the **File** menu.

In the **Quick Start** dialog, select the **Blank GUI (default)** template. Click **OK** to display the blank GUI in the Layout Editor, as shown in the following figure. You can choose to save your GUI immediately under a different name by selecting **Save on startup**. Otherwise, GUIDE prompts you to do so the first time you run the GUI.

Set the GUI Figure Size

Specify the size of the GUI by resizing the grid area in the Layout Editor. Click on the lower-right corner and resize the grid until it is about 4-by-3 inches.



If you want to set the position or size of the GUI to an exact value, do the following:

1 Select **Property Inspector** from the **View** menu.

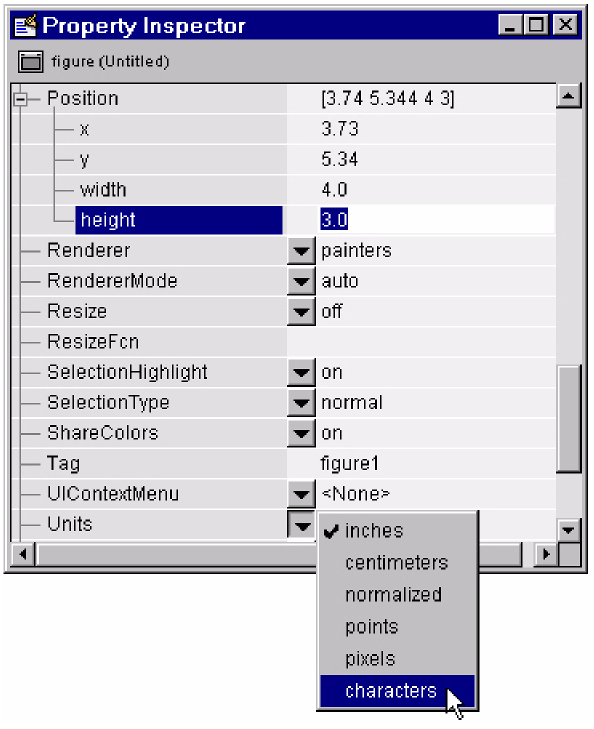
2 Select the button next to `Units` and then select `inches` from the pop-up menu

3 Click the **+** sign next to `Position`.

4 Type the `x` and `y` coordinates of the point where you want the lower left corner of the GUI to appear, and its width and height, as shown in the following figure.

5 Reset the `Units` property to `characters`.

Note Setting the `Units` property to `characters` gives the GUI a more consistent appearance across platforms.

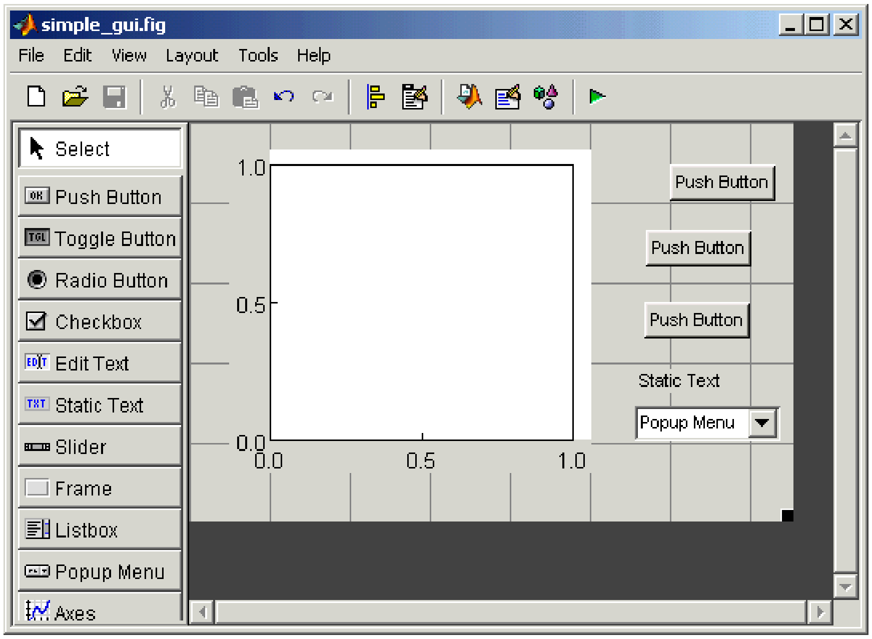


Position	[3.74 5.344 4 3]
x	3.73
y	5.34
width	4.0
height	3.0
Renderer	painters
RendererMode	auto
Resize	off
ResizeFcn	
SelectionHighlight	on
SelectionType	normal
ShareColors	on
Tag	figure1
UIContextMenu	<None>
Units	

Add the Components

Select the components to add from the palette and drag them into the layout area. You can resize components from any corner handle while it is selected.

Add three push buttons, a static text, a pop-up menu, and an axes. Arrange them as shown in the following figure. Resize the axes component to about 2-by-2 inches by selecting it with the mouse, and then clicking and dragging the lower-right corner.



Align the Objects

You can align components with respect to one another with the Alignment Tool. For example, to align the three push buttons:

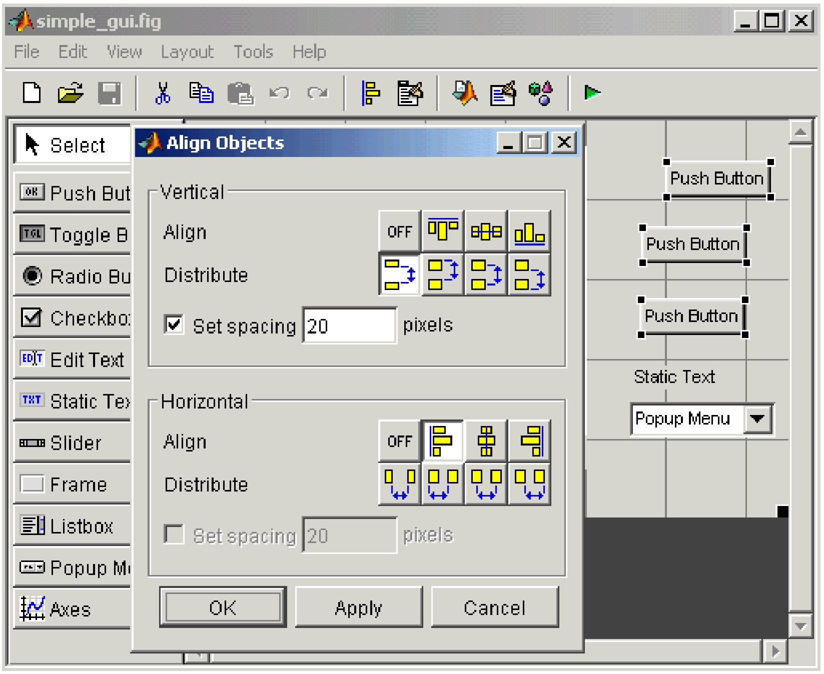
1 Select all three push buttons by pressing **Ctrl** and clicking them.

2 Select **Align Objects** from the **Tools** menu to display the Alignment Tool.

3 Make the following settings in the Alignment Tool, as shown in the following figure.

   - 20 pixels spacing between push buttons in the vertical direction.
   - Left-aligned in the horizontal direction.

4 Click **OK**.



Align Objects

Vertical
Align: OFF
Distribute
✓ Set spacing 20 pixels

Horizontal
Align: OFF
Distribute
Set spacing 20 pixels

OK | Apply | Cancel

To learn more about the Layout Editor, see “Laying Out GUIs — The Layout Editor” on page 4-2

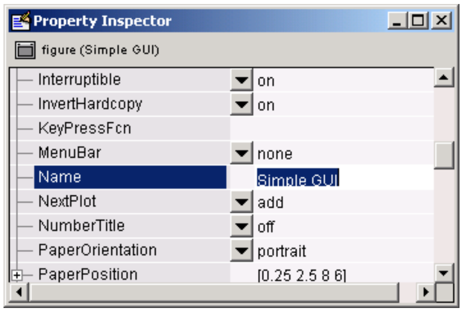
Programming the GUI

After laying out the GUI, the next step is to program it. This section explains how to do so. The section covers

- Setting properties for the GUI components
- Opening the GUI M-file
- The opening function
- Sharing data between callbacks
- Adding code to the opening function
- Adding code to the callbacks
- Using the Object Browser to identify callbacks

Set Properties for the GUI Components

To set the properties of each GUI component, select the **Property Inspector** from the **View** menu to display the Property Inspector. When you select a component in the Layout Editor, the Property Inspector displays the component's properties. If no component is selected, the Property Inspector displays the properties of the GUI figure, as shown in the following figure.



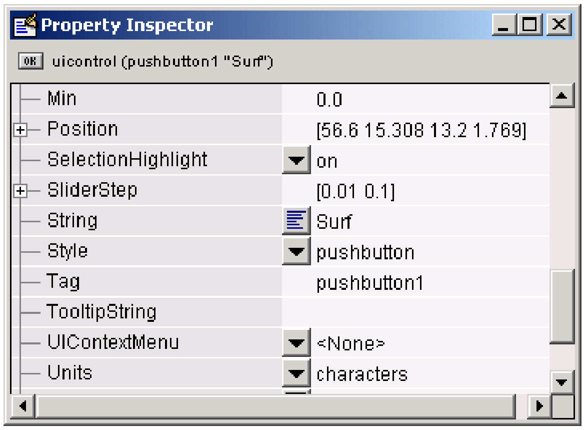
Interruptible	on
InvertHardcopy	on
KeyPressFcn	
MenuBar	none
Name	Simple GUI
NextPlot	add
NumberTitle	off
PaperOrientation	portrait
PaperPosition	[0.25 2.5 8 6]

Name Property

Scroll down the list of properties until you come to `Name`. In the field next to `Name`, type `Simple GUI`, as shown in the following figure. This is the title that is displayed at the top of the GUI.

String Property

You can set the label in a GUI component by its `String` property. For example, to set the label of the top push button, select the push button in the Layout Editor and then scroll down in the Property Inspector until you come to `String`. In the field to the right of String, change `Push Button` to `Surf`, as shown in the following figure.

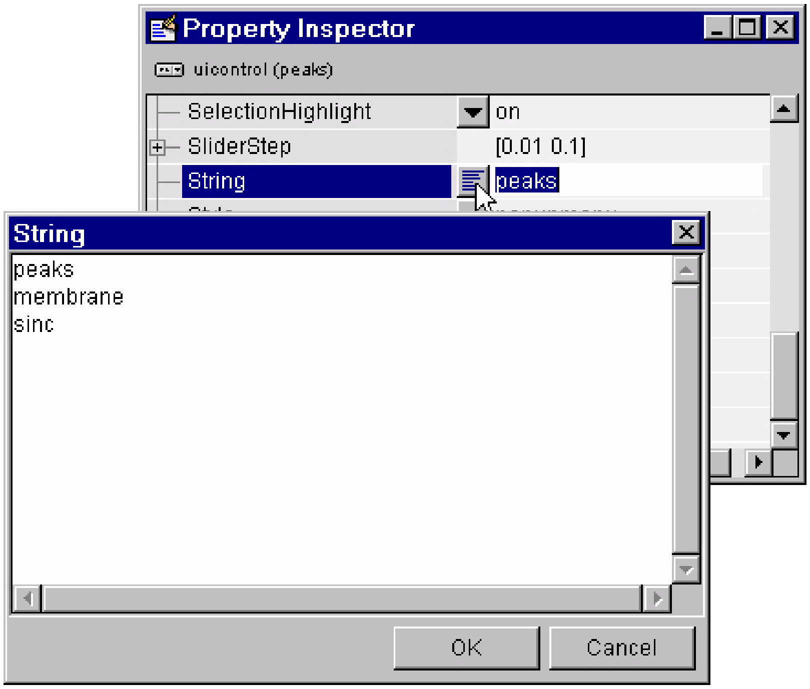

Property Inspector

uicontrol (pushbutton1 "Surf")

Min	0.0
Position	[56.6 15.308 13.2 1.769]
SelectionHighlight	on
SliderStep	[0.01 0.1]
String	Surf
Style	pushbutton
Tag	pushbutton1
TooltipString	
UIContextMenu	<None>
Units	characters

You can view the change by clicking the Layout Editor. Similarly, change the `String` property of the middle push button to `Mesh`, the bottom push button to `Contour`, and the `Static Text` to `Select Data`.

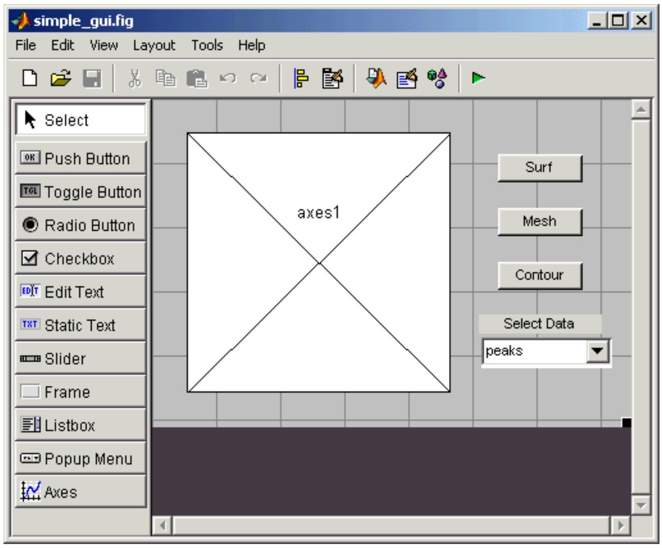
Pop-up Menu Items. To set pop-up menu items, select the popup menu in the Layout Editor. In the Property Inspector, click the icon next to `String`. This opens the `String` property edit box. Delete `Popup Menu` in the `String` property edit box, and type `peaks`, `membrane`, and `sinc` on three separate lines, as shown in the following figure:



SelectionHighlight on
SliderStep [0.01 0.1]
String peaks

peaks
membrane
sinc

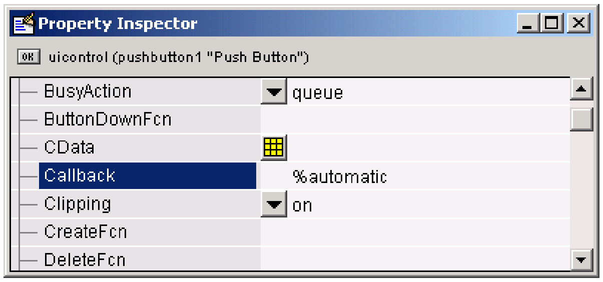
When you click on the Layout Editor, the GUI appears as in the following figure.



Callback and Tag Properties

The `Callback` property specifies the *callback* — the function in the GUI M-file that gets called when a user activates the component, for example, by clicking a push button.

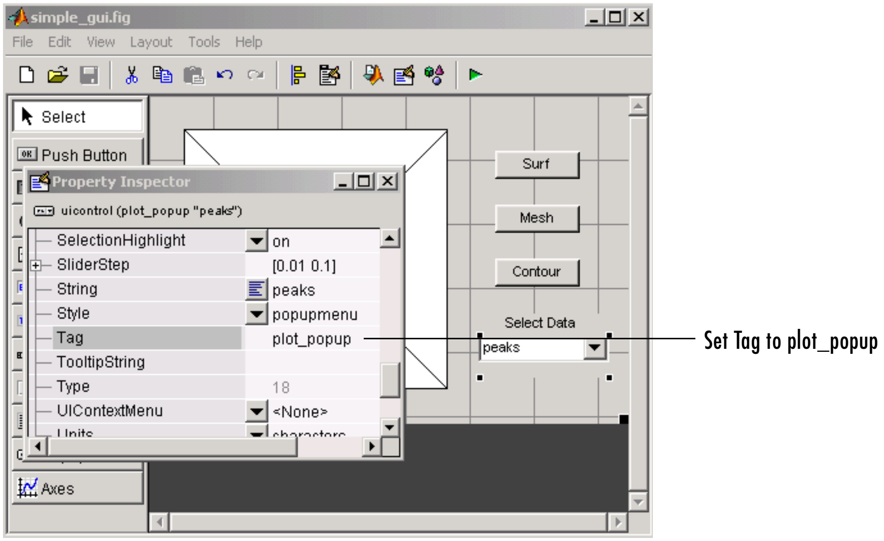
When you first add a component to the layout, its `Callback` property is set to the string `%automatic`, as shown in the following figure.



uicontrol (pushbutton1 "Push Button")

BusyAction	queue
ButtonDownFcn	
CData	
Callback	%automatic
Clipping	on
CreateFcn	
DeleteFcn	

When you save or run the GUI, GUIDE converts this string to one that calls a function in the generated M-file. GUIDE uses the component's `Tag` property to to associate the component with the function. By default, Tag is set to a generic label such as pushbutton1. You might want to change the Tag property of the pop-up menu to a more descriptive label, such as plot_popup, as shown in the following figure, before you save or run the GUI for the first time:



```
SelectionHighlight   on
SliderStep           [0.01 0.1]
String               peaks
Style                popupmenu
Tag                  plot_popup
TooltipString
Type                 18
UIContextMenu        <None>
```

When you save or run the GUI, GUIDE sets the pop-up menu Callback property to plot_popup_Callback. If you later change the Tag, GUIDE updates the Callback property to match the new Tag - see “Changing Component Tag Properties” on page 3-44.

Similarly, change the push button tags to surf_pushbutton, mesh_pushbutton, and contour_pushbutton.

To learn more, see “Setting Component Properties — The Property Inspector” on page 4-14

Opening the GUI M-File

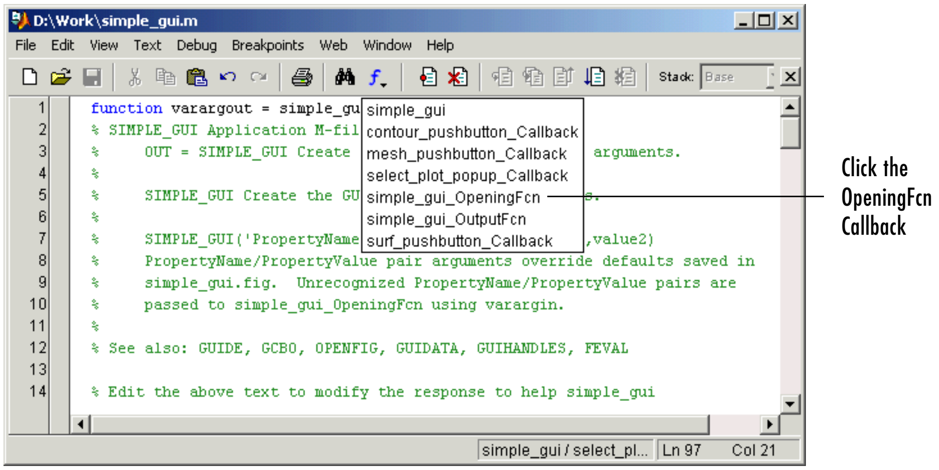
When you first save or run the GUI, GUIDE generates an M-file that contains all the callbacks. See “GUI FIG-Files and M-Files” on page 1-5. For each component that has a callback associated with it, GUIDE generates a framework for the callback, containing just a function definition. You must then add code to the callbacks to make them work. In this example, you add code to the callbacks for the three push buttons and the pop-up menu. For more information, see “Understanding the GUI M-File” on page 5-2.

You can edit the M-file code by clicking the M-file Editor icon on the toolbar. If you previously saved or ran the GUI, pressing the M-file Editor icon displays the M-file in the MATLAB editor. Otherwise, GUIDE does two things:

- Generates the M-file associated with the GUI.
- Opens the **Save GUI as** dialog.

Type a name for the FIG-file in the **File name** field. GUIDE assigns the same name to the M-file. When you click **Save**, GUIDE saves the M-file and opens it in the M-file Editor.

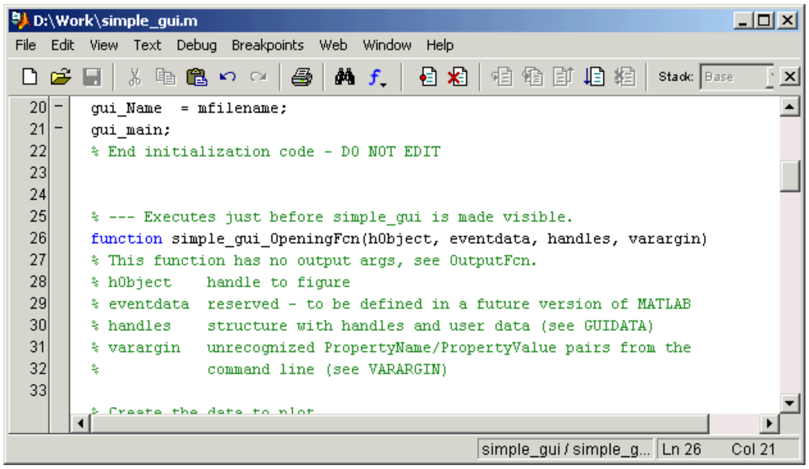
You can view a list of the callbacks in the M-file by clicking the function icon on the toolbar, as shown in the following figure.


D:\Work\simple_gui.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
simple_gui
contour_pushbutton_Callback
mesh_pushbutton_Callback
select_plot_popup_Callback
simple_gui_OpeningFcn
simple_gui_OutputFcn
surf_pushbutton_Callback
simple_gui / select_pl... Ln 97 Col 21
Click the OpeningFcn Callback

```
function varargout = simple_gu
% SIMPLE_GUI Application M-fil
%      OUT = SIMPLE_GUI Create                                 arguments.
%
%      SIMPLE_GUI Create the GU                                s.
%
%      SIMPLE_GUI('PropertyName                                ,value2)
%      PropertyName/PropertyValue pair arguments override defaults saved in
%      simple_gui.fig.  Unrecognized PropertyName/PropertyValue pairs are
%      passed to simple_gui_OpeningFcn using varargin.
%
% See also: GUIDE, GCBO, OPENFIG, GUIDATA, GUIHANDLES, FEVAL

% Edit the above text to modify the response to help simple_gui
```

Clicking a callback in the pop-up menu moves the cursor in the editor to the first line of the callback. For example, click `simple_gui_OpeningFcn` to display the opening function, as shown in the following figure.


D:\Work\simple_gui.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
simple_gui / simple_g... Ln 26 Col 21

```
gui_Name  = mfilename;
gui_main;
% End initialization code - DO NOT EDIT


% --- Executes just before simple_gui is made visible.
function simple_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   unrecognized PropertyName/PropertyValue pairs from the
%            command line (see VARARGIN)

% Create the data to plot
```

You can add code to the opening function to create data for the GUI, or perform other tasks, as described in the next section.

Opening Function

The code in the opening function is executed just before the GUI is made visible to the user. You can add code to the opening function to perform tasks that need to be done before the user has access to the GUI, for example, to create data or to read data from an external source. In this example, you add code that creates three data sets in the opening function, using the MATLAB functions `peaks`, `membrane`, and `sinc`. This code is described in the section “Adding Code to the Opening Function” on page 2-16.

Note that GUIDE names the opening function with the name of the M-file prefixed to `_OpeningFcn`. In this example, the M-file is named `simple_gui.m`, so that the opening function is named `simple_gui_OpeningFcn`.

Sharing Data Between Callbacks

You can share data between callbacks by storing it in the MATLAB `handles` structure. For example, to store data contained in a vector `X` in the `handles` structure, you

1 Choose a name for the field of the `handles` structure where you want to store the data, for example, `handles.my_data`

2 Set the field equal to `X` with the following command:

```
handles.my_data = X;
```

3 Save the `handles` structure with the `guidata` function:

```
guidata(hObject, handles)
```

Here, `hObject` is the handle to the object that executes the callback.

Note To save any changes that you make to the handles structure, you must add the command `guidata(hObject, handles)` following the code that implements the changes.

To retrieve `X` in another callback, use the command

```
X = handles.my_data;
```

You can access the data in the `handles` structure in any callback because `hObject` and `handles` are input arguments for all the callbacks generated by GUIDE.

For more detailed information on the `handles` structure, see

- "Sharing Data with the Handles Structure" on page 5-2
- "Managing GUI Data with the Handles Structure" on page 5-8

Adding Code to the Opening Function

To create data for the GUI to plot, add the following code to the opening function immediately after the comments following the function declaration.

```
% --- Executes just before simple_gui is made visible.
function simple_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to untitled (see VARARGIN)
```

Add this code

Autogenerated code

```
% Create the data to plot
handles.peaks=peaks(35);
handles.membrane=membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc = sin(r)./r;
handles.sinc = sinc;
handles.current_data = handles.peaks;
surf(handles.current_data)
```

The first six executable lines create the data using the MATLAB functions `peaks`, `membrane` and `sinc` to generate the data.

The next line, `handles.current_data = handles.peaks`, sets the `current_data` field of the handles structure equal to the data for peaks. The value of `handles.current_data` changes each time a user selects a different plot from the pop-up menu — see "Pop-up Menu Callback" on page 2-19.

The last line displays the surf plot for peaks, which appears when the GUI is first opened.

GUIDE automatically generates two more lines of code in the opening function, which follow the code that you add:

- `handles.output = hObject` saves the handle to the GUI for later access by the output function. While this command is not necessary in this example, it is useful if you want to return the GUI handle to the command line.
- `guidata(hObject, handles)` saves the `handles` structure.

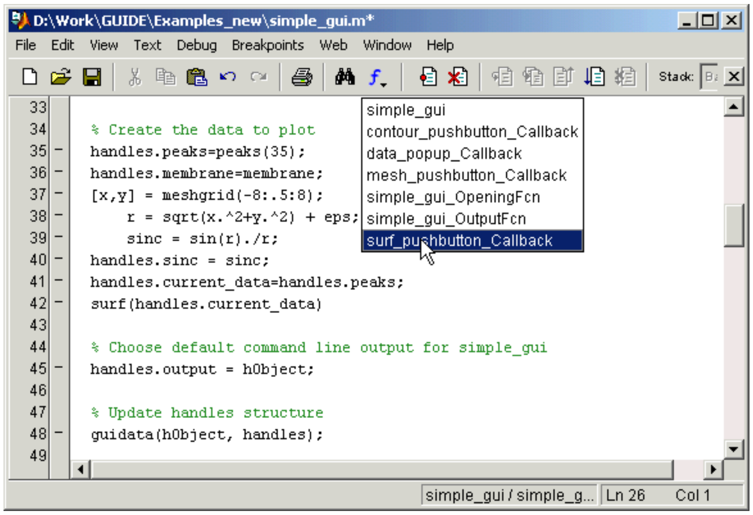
Adding Code to the Callbacks

When the GUI is completed and running, and a user clicks a component of the GUI, MATLAB executes the callback specified by the component's `Callback` property. The name of the callback is determined by the component's `tag` property. For example, the callback for the **Surf** push button is `surf_pushbutton_Callback`.

The following section describes how to add the code for the callbacks.

Push Button Callbacks

Each of the push buttons creates a different type of plot using the data specified by the current selection in the pop-up menu. Their callbacks get data from the `handles` structure and then plot it. To add code to the surf push button callback, click `surf_pushbutton_Callback` in the callback pop-up menu.



```

% Create the data to plot
handles.peaks=peaks(35);
handles.membrane=membrane;
[x,y] = meshgrid(-8:.5:8);
    r = sqrt(x.^2+y.^2) + eps;
    sinc = sin(r)./r;
handles.sinc = sinc;
handles.current_data=handles.peaks;
surf(handles.current_data)

% Choose default command line output for simple_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

```

Add the code after the comments following the function definition, as shown below:

Surf push button callback:

```
% --- Executes on button press in surf_pushbutton.
function surf_pushbutton_Callback(hObject, eventdata, handles)
% hObject    handle to surf_pushbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Add this code

Autogenerated code

```
% Display surf plot of the currently selected data
```

```
surf(handles.current_data);
```

You can add similar code to the **Mesh** and **Contour** push button callbacks after the autogenerated code.

Add this code to the **Mesh** push button callback:

```
% Display mesh plot of the currently selected data
mesh(handles.current_data);
```

Add this code to the **Contour** push button callback:

```
% Display contour plot of the currently selected data
contour(handles.current_data);
```

Pop-up Menu Callback

The pop-up menu enables users to select the data to plot. Every time a user selects one of the three plots, the pop-up menu callback reads the pop-up menu `Value` property to determine what item is currently displayed and sets `handles.current_data` accordingly. Add the following code to the `plot_popup_Callback` after the comments following the function definition.

```
% --- Executes on selection change in data_popup.
function plot_popup_Callback(hObject, eventdata, handles)
% hObject    handle to surf_pushbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

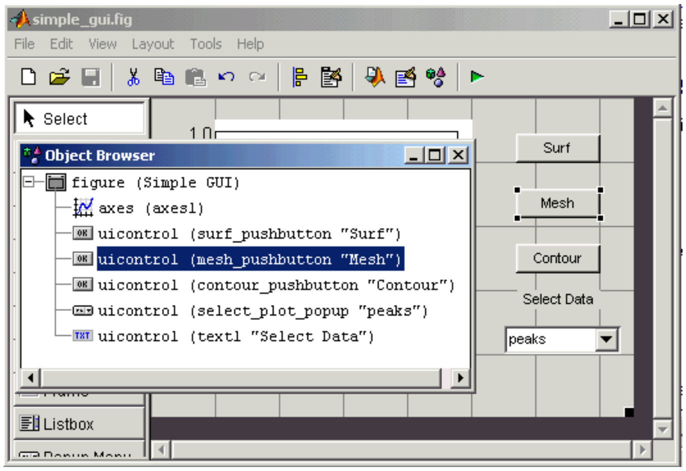
Add this code

Autogenerated code

```
val = get(hObject,'Value');
str = get(hObject, 'String');
switch str{val};
case 'peaks' % User selects peaks
   handles.current_data = handles.peaks;
case 'membrane' % User selects membrane
   handles.current_data = handles.membrane;
case 'sinc' % User selects sinc
   handles.current_data = handles.sinc;
end
guidata(hObject,handles)
```

Using the Object Browser to Identify Callbacks

In this example, it is easy to keep track of the GUI component that corresponds to each callback. But in a more complicated GUI, keeping track of callbacks can be more difficult. To identify the component corresponding to a callback, select **Object Browser** from the **View** menu in the Layout Editor. This displays the Object Browser as shown in the following figure. The Object Browser lists the `tag` and `string` properties of each component of the GUI. Selecting the name of a component in the list also selects the component in the Layout Editor. For example, in the following figure, the `uicontrol (mesh_pushbutton  Mesh )` is selected in the Object Browser. The tag `mesh_pushbutton` corresponds to the callback `mesh_pushbutton_Callback`. Note that the corresponding component, the mesh push button, is also selected in the Layout Editor.

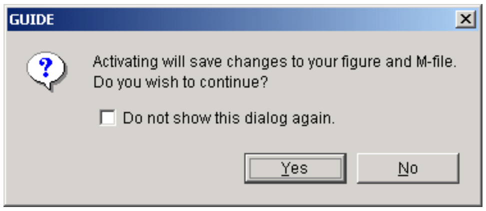


```
figure (Simple GUI)
  axes (axes1)
  uicontrol (surf_pushbutton "Surf")
  uicontrol (mesh_pushbutton "Mesh")
  uicontrol (contour_pushbutton "Contour")
  uicontrol (select_plot_popup "peaks")
  uicontrol (text1 "Select Data")
```

To learn more about programming GUIs in GUIDE, see “Programming GUIs” on page 5-1

Saving and Running the GUI

After writing the callbacks, you can run the GUI by selecting **Run** from the **Tools** menu or clicking the **Run** button on the GUIDE toolbar. If you have not saved the GUI recently, GUIDE displays the following dialog box.

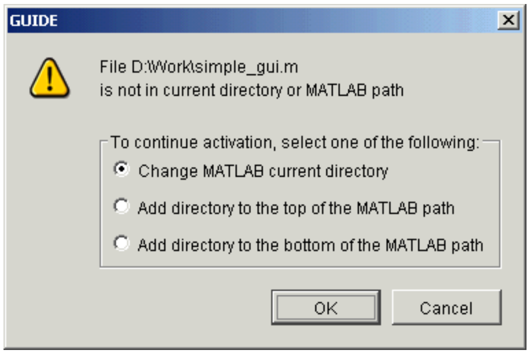

GUIDE

Activating will save changes to your figure and M-file.
Do you wish to continue?

☐ Do not show this dialog again.

If this happens, click **Yes** and then save the GUI files to a writable directory.

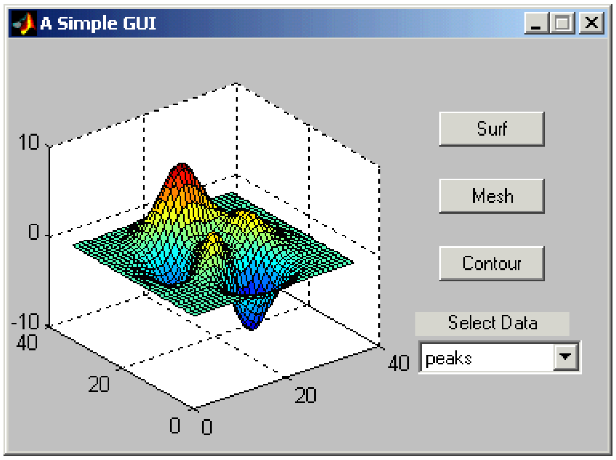
If the directory where you save the GUI is not on the MATLAB path, GUIDE opens the following dialog, giving you the option of changing the current working directory to the directory containing the GUI files, or adding that directory to the MATLAB path.


GUIDE

File D:\Work\simple_gui.m
is not in current directory or MATLAB path

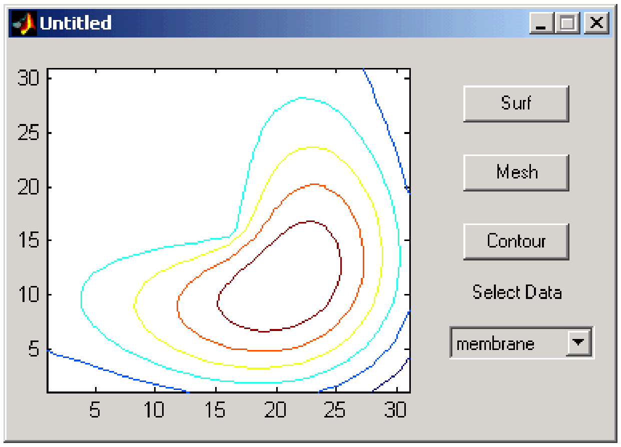
To continue activation, select one of the following:
● Change MATLAB current directory
○ Add directory to the top of the MATLAB path
○ Add directory to the bottom of the MATLAB path

Click **OK** to change the current working directory. GUIDE then opens the GUI as shown in the following figure.



Note The name of the FIG-file saved by the Layout Editor and the generated M-file must match. See "Renaming Application Files and Tags" if you want to rename files after first activating the GUI.

Next, select **membrane** in the pop-up menu and click the **Contour** push button. The GUI should look like the following figure.



Try experimenting with this GUI by adding another data set in the opening function, and a push button that displays a plot of the data set. Make sure to add the name of the new data set to the pop-up menu as well.

For more examples of creating GUIs with GUIDE, see the following sections:

- "Application Examples" on page 6-1
- "Using GUIDE Templates" on page 3-6

MATLAB GUIs

This section provides an overview of creating GUIs with GUIDE.

What Is a GUI?

A graphical user interface (GUI) is a user interface built with graphical objects — the *components* of the GUI — such as buttons, text fields, sliders, and menus. If the GUI is designed well-designed, it should be intuitively obvious to the user how its components function. For example, when you move a slider, a value changes; when you click an **OK** button, your settings are applied and the dialog box is closed. Fortunately, most computer users are already familiar with GUIs and know how to use standard GUI components.

By providing an interface between the user and the application's underlying code, GUIs enable the user to operate the application without knowing the commands would be required by a command line interface. For this reason, applications that provide GUIs are easier to learn and use than those that are run from the command line.

The sections that follow describe how to create GUIs with GUIDE. This includes laying out the components, programming them to do specific things in response to user actions, and saving and opening the GUI.

Creating GUIs with GUIDE

MATLAB implements GUIs as figure windows containing various uicontrol objects. You must program each object to perform the action you intend it to do when a user activates the component. In addition, you must be able to save and run your GUI. All of these tasks are simplified by GUIDE, the MATLAB graphical user interface development environment.

This section covers the following topics:

- “GUI Development Environment” on page 3-3
- “Editing Version 5 GUIs with Version 6 GUIDE” on page 3-4

GUI Development Environment

Creating a GUI involves two basic tasks:

- Laying out the GUI components
- Programming the GUI components

For an example of these tasks, see “Example: Creating a GUI” on page 2-1.

GUIDE primarily is a set of layout tools. However, GUIDE also generates an M-file that contains code to handle the initialization and launching of the GUI. This M-file provides a framework for the implementation of the *callbacks* — the functions that execute when users activate components in the GUI.

GUIDE Generated Files

While it is possible to write an M-file that contains all the commands to lay out a GUI, it is much easier to use GUIDE to lay out the components interactively. When you save or run the GUI, GUIDE automatically generates two files:

- A FIG-file — a file with a `.fig` file name extension, which contains a complete description of the GUI figure and all of its children (uicontrols and axes), as well as the values of all object properties. You make changes to the FIG-file by editing the GUI in the Layout Editor. See “Laying Out GUIs — The Layout Editor” on page 4-2 for more information.
- An M-file — a file with a `.m` file name extension, which contains the functions that run and control the GUI and the callbacks. This file is referred to as the *GUI M-file*. For a detailed explanation of programming the M-file, see “Programming GUIs” on page 5-1.

Note that the M-file does not contain the code that lays out the uicontrols; this information is saved in the FIG-file.

Features of the GUI M-file

GUIDE simplifies the process of creating GUIs by automatically generating the GUI M-file directly from your layout. GUIDE generates callbacks for each component in the GUI that requires a callback. Initially, GUIDE generates just a function definition line for each callback. You can add code to the callback to make it perform the operation you want.

The M-file contains two other functions where you might also need to add code:

- Opening function — performs tasks before the GUI becomes visible to the user, such as creating data for the GUI. GUIDE names this function `my_gui_OpeningFcn`, where `my_gui` is the name of the GUI.
- Output function — outputs variables to the command line, if necessary. GUIDE names this function `my_gui_OutputFcn`, where `my_gui` is the name of the GUI.

See “Understanding the GUI M-File” on page 5-2 for more details.

Editing Version 5 GUIs with Version 6 GUIDE

In MATLAB Version 5, GUIDE saved GUI layouts as MAT-file/M-file pairs. In MATLAB Version 6, GUIDE saves GUI layouts as FIG-files. GUIDE also generates an M-file to program the GUI callbacks. However, this M-file does not contain layout code as did M-files created in Version 5.

Use the following procedure to edit a Version 5 GUI with Version 6 GUIDE:

1. Display the Version 5 GUI.

2. Obtain the handle of the GUI figure. If the figure’s handle is hidden (i.e., the figure’s `HandleVisibility` property is set to `off`), set the root `ShowHiddenHandles` property to `on`:

   ```
   set(0,'ShowHiddenHandles','on')
   ```

   Then get the handle from the root’s `Children` property:

```
hObject = get(0,'Children');
```

This statement returns the handles of all figures that exist when you issue the command. For simplicity, ensure that the GUI is the only figure displayed.

3 Pass the handle as an argument to the `guide` command:

```
guide(hObject)
```

Saving the GUI in Version 6 GUIDE

When you save the edited GUI with Version 6 GUIDE, MATLAB creates a FIG-file that contains all the layout information. The original MAT-file/M-file combination is no longer used. To display the revised GUI, run the M-file generated by GUIDE.

Updating Callbacks

Ensure that the `Callback` properties of the uicontrols in your GUI are set to the desired callback string or callback M-file name when you save the FIG-file. If your Version 5 GUI used an M-file that contained a combination of layout code and callback routines, then you should restructure the M-file to contain only the commands needed to initialize the GUI and the callback functions. The M-file generated by Version 6 GUIDE can provide a model of how to restructure your code.

Note By default, GUIDE generates an M-file having the same name as the FIG-file saved with the Layout Editor. When you run a GUI from the Layout Editor, GUIDE executes this M-file to run the GUI.

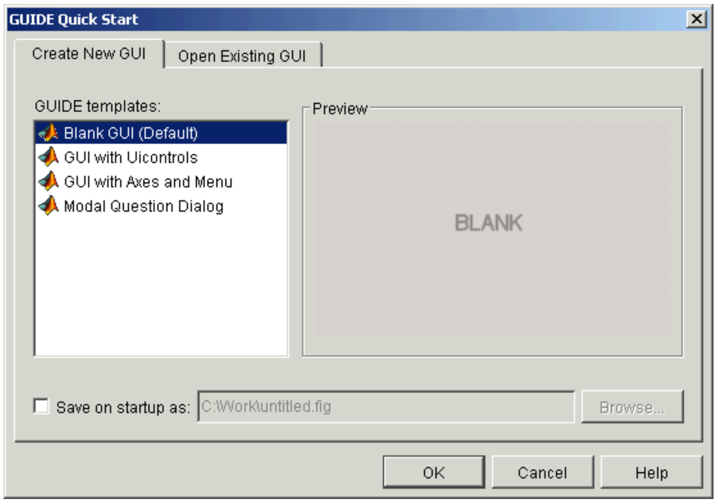
Using GUIDE Templates

GUIDE provides several templates, which simple examples that you can modify to create your own GUIs. The templates are fully functional GUIs: their callbacks are already programmed. You can view the code for these callbacks to see how they work, and then modify the callbacks for your own purposes.

You can access the templates in two ways:

- Start GUIDE by entering `guide` at the MATLAB prompt.
- If GUIDE is already open, select **New** from the **File** menu in the Layout Editor.

Starting GUIDE displays the **GUIDE Quick Start** dialog as shown in the following figure.



The **Quick Start** dialog gives you two options:

- Select the **Open Existing GUI** tab and open a GUI that you have already created.
- Select the **Create New GUI** tab and open one of the templates.

The preceding figure shows the **Quick Start** dialog with the **Create New GUI** tab selected. Selecting a template in the left pane displays a preview in the right pane. Clicking **OK** opens the GUI template in the Layout Editor. If you select **Save on startup as** and type in name in the field to the right, GUIDE saves the GUI before opening it in the Layout Editor. If you choose not to save the GUI at this point, GUIDE prompts you to save it the first time you run the GUI.

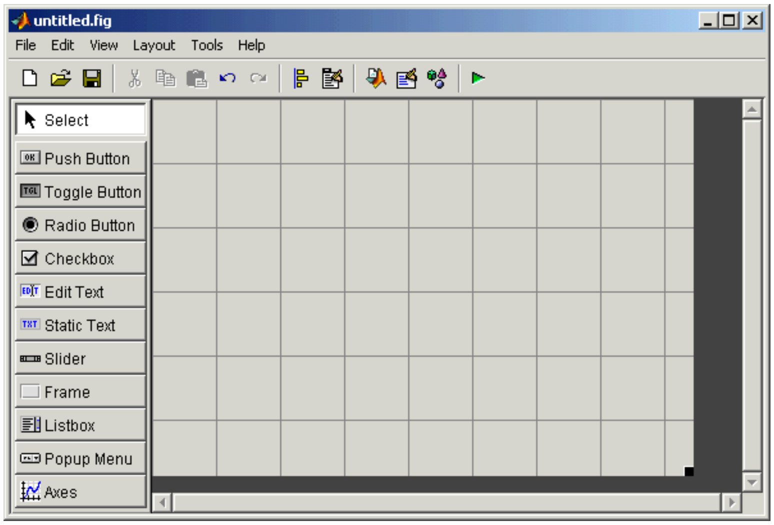
GUIDE provides four templates, which are described in the following sections:

- "Blank GUI" on page 3-7
- "GUI with UIcontrols" on page 3-8
- "GUI with Axes and Menu" on page 3-9
- "Modal Question Dialog" on page 3-10

To view the M-file for any of these templates, open the template in the Layout Editor and click the M-file editor icon on the toolbar.

Blank GUI

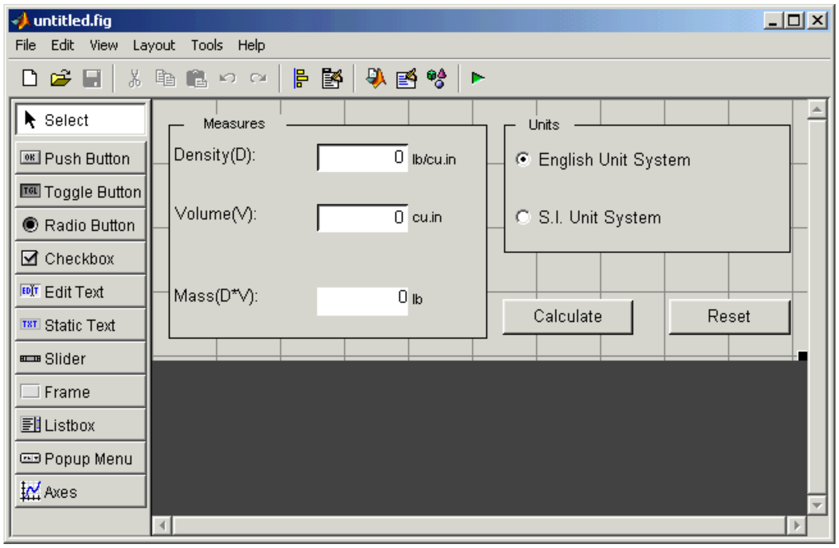
The blank GUI template displayed in the Layout Editor is shown in the following figure.



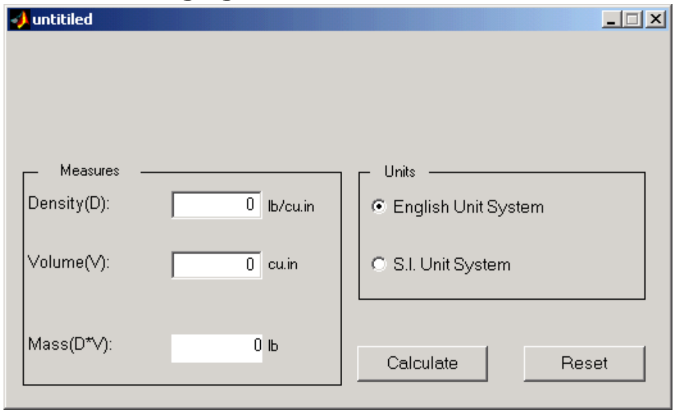
Select the blank GUI if the other templates are not suitable starting points for the GUI you are creating, or if you prefer to start with an empty GUI.

GUI with UIcontrols

The following figure shows the GUI with UIcontrols template displayed in the Layout Editor.



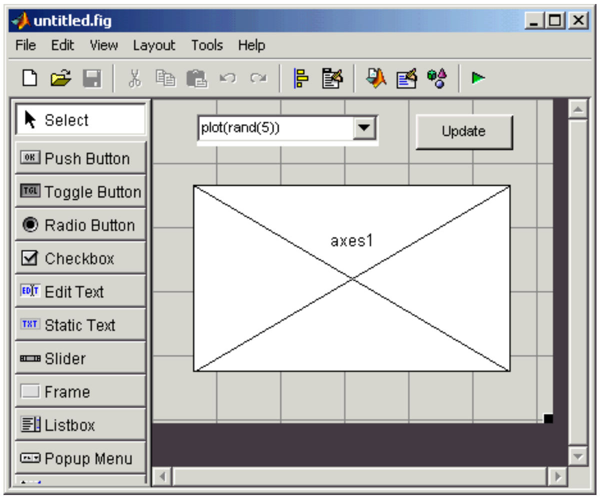
When you run the GUI by clicking the **Run** button, the GUI appears as shown in the following figure.



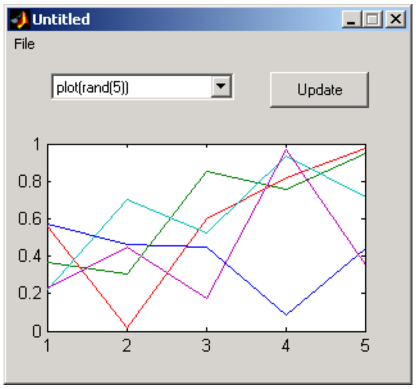
When a user enters values for the density and volume of an object, and clicks the **Calculate** button, the GUI calculates the mass of the object and displays the result next to `Mass(D*V)`.

GUI with Axes and Menu

The GUI with axes and menu template is shown in the following figure.



When you run the GUI by clicking the **Run** button on the toolbar, the GUI displays a plot of five random numbers generated by the MATLAB `rand(5)` command, as shown in the following figure.



You can select other plots in the pop-up menu. Clicking the **Update** button displays the currently selected plot on the axes.

The GUI also has a menu with three items:

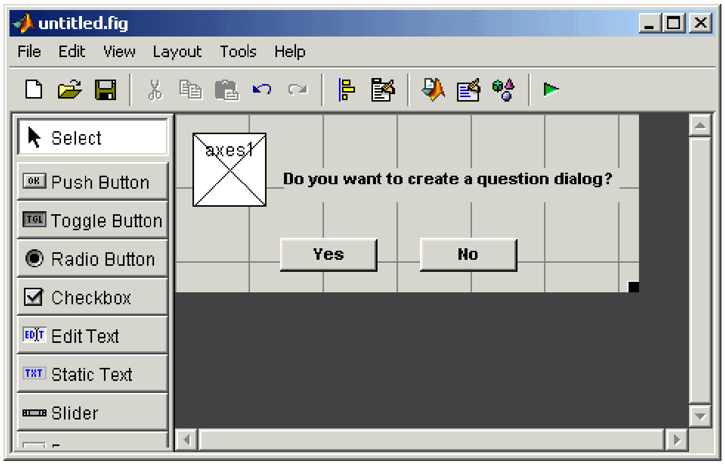
- Selecting **Open** displays a dialog from which you can open files on your computer.
- Selecting **Print** executes the `printdlg` command, which opens the **Print** dialog:

  ```
  printdlg(handles.figure1)
  ```

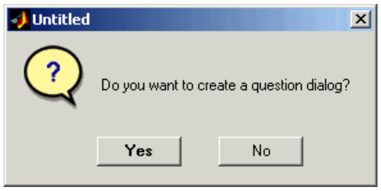
  Note that `handles.figure1` contains the current plot. Clicking **Yes** in the **Print** dialog prints the plot.
- Selecting **Close** closes the GUI.

Modal Question Dialog

The modal question dialog template displayed in the Layout Editor is shown in the following figure.



Running the GUI displays the dialog shown in the following figure:



The GUI returns the text string `Yes` or `No`, depending on which button you press. The GUI is *blocking*, which means that the current M-file stops executing until the GUI restores execution. You can make a GUI blocking by adding the following command to the opening function:

```
uiwait(handles.figure1);
```

To restore access to other MATLAB windows once a button is pressed, add the following command to callbacks for both the **Yes** and **No** push buttons:

```
uiresume(handles.figure1);
```

The GUI is also *modal,* which means that the user cannot interact with other MATLAB windows until clicking one of the buttons. See “Using Modal Figure Windows” on page 5-20 for more information on making a GUI modal.

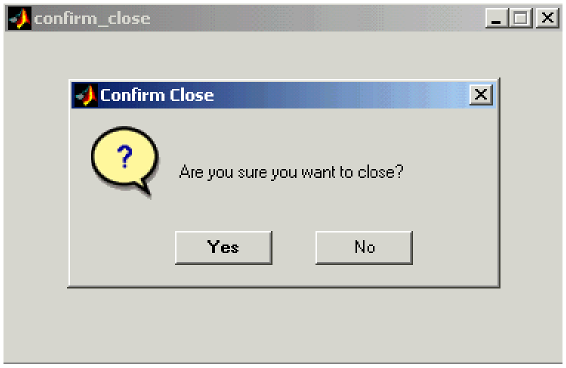
Select this template if you want your GUI to return a string or to be modal.

See “Example: Using the Modal Dialog to Confirm an Operation” on page 3-13 for an example of using this template with another GUI.

Example: Using the Modal Dialog to Confirm an Operation

This example illustrates how to use the modal dialog GUI together with another GUI that has a **Close** button. Clicking the **Close** button displays the modal dialog, which asks users to confirm that they really want to proceed with the close operation.

The following figure illustrates the dialog positioned over the GUI application, awaiting the user's response.



Are you sure you want to close?

The example is presented in the following sections:

- "View the Layout and GUI M-File" on page 3-13
- "Setting Up the Close Confirmation Dialog" on page 3-14
- "Setting Up the GUI with the Close Button" on page 3-15
- "Running the GUI with the Close Button" on page 3-16
- "How the GUI and Dialog Work" on page 3-17

View the Layout and GUI M-File

If you are reading this in the MATLAB Help Browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

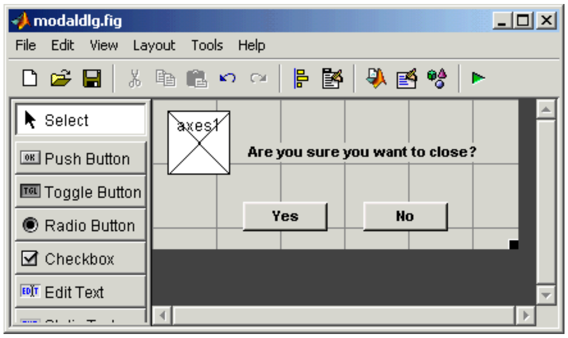
- Click here to display the GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

Setting Up the Close Confirmation Dialog

To set up the dialog, do the following:

1. Select **New** from the **File** menu in the GUIDE Layout Editor.
2. In the **GUIDE Quick Start** dialog, select the **Modal Question Dialog** template and click **OK**.
3. Right-click the static text, `Do you want to create a question dialog?`, in the Layout Editor and select **Property Inspector** from the pop-up menu.
4. Scroll down to `String` in the Property Inspector and change the `String` property to `Are you sure you want to close?`
5. Select **Save** from the **File** menu and type `modaldlg.fig` in the **File name** field.

The GUI should now appear as in the following figure.



Setting Up the GUI with the Close Button

To set up the second GUI with a **Close** button, do the following:

1 Select **New** from the **File** menu in the GUIDE Layout Editor.

2 In the **GUIDE Quick Start** dialog, select **Blank GUI (Default)** and click **OK**. This opens the blank GUI in a new Layout Editor window.

3 Drag a push button from the Component palette of the Layout Editor into the layout area.

4 Right-click the push button and select **Property Inspector** from the pop-up menu.

5 Change the `String` property to `Close`.

6 Change the `Tag` property to `close_pushbutton`.

7 Click the M-file editor icon on the toolbar of the Layout Editor.

8 Click the callback icon on the toolbar of the M-file editor and select `close_pushbutton_Callback` from the menu.

The following generated code for the close button callback should appear in the M-file editor:

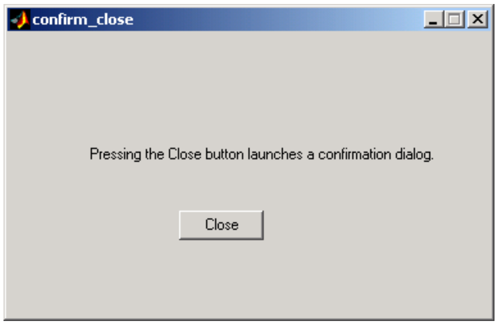
```
% --- Executes on button press in close_pushbutton.
function close_pushbutton_Callback(hObject, eventdata, handles)
% hObject    handle to close_pushbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

9 After these comments, add the following code:

```
% Get the current position of the GUI from the handles structure
% to pass to the modal dialog.
pos_size = get(handles.figure1,'Position');
% Call modaldlg with the argument 'Position'.
user_response = modaldlg('Title','Confirm Close');
switch user_response
case {'No'}
   % take no action
case 'Yes'
   % Prepare to close GUI application window
   %                  .
   %                  .
   %                  .
   delete(handles.figure1)
end
```

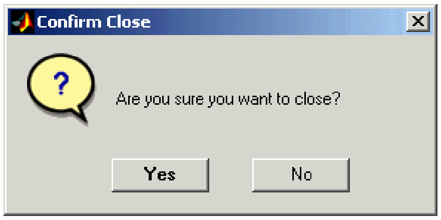
Running the GUI with the Close Button

Run the GUI with the **Close** button by clicking the **Run** button on the Layout Editor toolbar. The GUI appears as in the following figure:



Pressing the Close button launches a confirmation dialog.

When you click the **Close** button on the GUI, the modal dialog appears as shown in the following figure:



Clicking the **Yes** button closes both the close dialog and the GUI that calls it. Clicking the **No** button closes just the dialog.

How the GUI and Dialog Work

This section describes what occurs when you click the **Close** button on the GUI:

1 User clicks the **Close** button. Its callback then

- Gets the current position of the GUI from the `handles` structure with the command

```
pos_size = get(handles.figure1,'Position')
```

- Calls the modal dialog with the command

```
user_response = modaldlg('Title','Confirm Close');
```

  This is an example of calling a GUI with a property value pair. In this case, the figure property is `'Title'`, and its value is the string `'Confirm Close'`. Opening `modaldlg` with this syntax displays the text "Confirm Close" at the top of the dialog. See "Input and Output Arguments" on page 5-6 for more information about the options for calling a GUI.

2 The modal dialog opens with the `'Position'` obtained from the GUI that calls it.

3 The opening function in the modal dialog M-file:

- Makes the dialog modal.
- Executes the `uiwait` command, which causes the dialog to wait for the user to click the **Yes** button or the **No** button, or click the close box (`X`) on the window border.

4 When a user clicks one of the two push buttons, the callback for the push button

- Updates the output field in the `handles` structure
- Executes `uiresume` to return control to the opening function where `uiwait` is called.

5 The output function is called, which returns the string `Yes` or `No` as an output argument, and deletes the dialog with the command

```
delete(handles.figure1)
```

6 When the GUI with the **Close** button regains control, it receives the string `Yes` or `No`. If the answer is `'No'`, it does nothing. If the answer is `'Yes'`, the close button callback closes the GUI with the command

```
delete(handles.figure1)
```

For more information on programming GUIs, see “Programming GUIs” on page 5-1.

Example: Displaying an Image File

This example shows how to display an image file on a GUI. The example

- Displays an image file on an axes.
- Resizes the axes to fit the image.
- Resizes the GUI so that its width and height are 100 pixels larger than the image file.

To build the example:

1 Open a blank template in the Layout Editor.

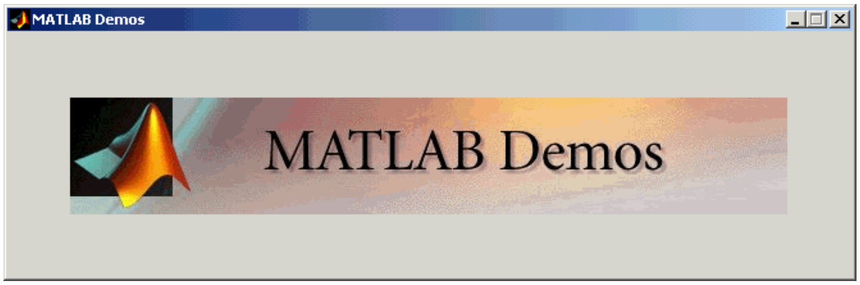
2 Drag an axes into the layout area.

3 Select **M-file** editor from the **View** menu.

4 Add the following code to the opening function:

```
set(hObject, 'Units', 'pixels');
handles.banner = imread([matlabroot filesep 'demos' filesep
'banner.jpg']); % Read the image file banner.jpg
info = imfinfo([matlabroot filesep 'demos' filesep
'banner.jpg']); % Determine the size of the image file
position = get(hObject, 'Position');
set(hObject, 'Position', [position(1:2) info.Width + 100
info.Height + 100]);
axes(handles.axes1);
image(handles.banner)
set(handles.axes1, ...
    'Visible', 'off', ...
    'Units', 'pixels', ...
    'Position', [50 50 info.Width info.Height]);
```

When you run the GUI, it appears as in the following figure.



The preceding code performs the following operations:

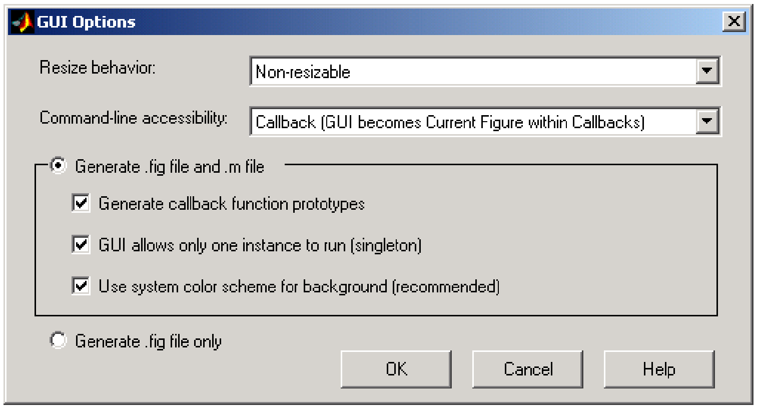
- Reads the image file `banner.jpg` using the command `imread`.
- Determines the size of the image file in pixels using the command `imfinfo`.
- Sets the width and height of the GUI to be 100 pixels greater than the corresponding dimensions of the image file, which are stored as `info.Width` and `info.Height`, respectively. The width and height of the GUI are the third and fourth entries of the vector `position`.
- Displays the image in the axes using the command `image(handles.banner)`.
- Makes the following changes to the axes properties:
  - Sets `'Visible'` to `'off'` so that the axes are invisible
  - Sets `'Units'` to `'pixels'` to match the units of the vector position
  - Sets `'Position'` to `[50, 50, info.Width info.Height]` to set the size of the axes equal to that of the image file, and center the image file on the GUI.

Selecting GUI Options

After opening a new GUI template in the Layout Editor, but before adding components to the layout, you should configure the GUI using the **GUI Options** dialog. Access the dialog by selecting **GUI Options...** from the Layout Editor **Tools** menu.

Configuring the GUI M-File

The **GUI Options** dialog enables you to select whether you want GUIDE to generate only a FIG-file for your layout or both a FIG-file and an M-file. You can also select a number of different behaviors for your GUI.



GUI Options

Resize behavior: Non-resizable

Command-line accessibility: Callback (GUI becomes Current Figure within Callbacks)

[✓ Generate .fig file and .m file]

[✓] Generate callback function prototypes

[✓] GUI allows only one instance to run (singleton)

[✓] Use system color scheme for background (recommended)

Generate .fig file only

The following sections describe the options in this dialog:

- "Resize Behavior" on page 3-22
- "Command-Line Accessibility" on page 3-23
- "Generate FIG-File and M-File" on page 3-25
- "Generate Callback Function Prototypes" on page 3-26
- "GUI Allows Only One Instance to Run (Singleton)" on page 3-27
- "Using the System Background Colors" on page 3-28
- "Generate FIG-File Only" on page 3-28

Resize Behavior

You can control whether users can resize the figure window containing your GUI and how MATLAB handles resizing. GUIDE provides three options:

- **Non-resizable** — Users cannot change the window size (default).
- **Proportional** — MATLAB automatically rescales the components in the GUI in proportion to the new figure window size.
- **User-specified** — Program the GUI to behave in a certain way when users resize the figure window.

The first two approaches simply set properties appropriately and require no other action. **User-specified** resizing requires you to write a callback routine that recalculates sizes and positions of the components based on the new figure size. The following sections discuss each approach.

Making Your GUI Nonresizable

Certain types of GUIs are typically nonresizable. Warning and simple question dialog boxes, particularly modal windows, are usually not resizable. After a simple interaction, users dismiss these GUIs so changing their size is not necessary.

Property Settings. GUIDE sets the following properties to create nonresizable GUIs:

- `Units` properties of the figure, axes, and uicontrols are set to `characters` (the Layout Editor default) so the GUI displays at the correct size on different computers.
- `Resize` figure property is set to `off`.
- `ResizeFcn` figure property is left unset.

Allowing Proportional GUI Resizing

Use this approach if you want to allow users to resize the GUI and are satisfied with a behavior that simply scales each component's size and relative position in proportion to the new figure size. Note that the font size of component labels does not resize and, if the size is reduced enough, these labels may become unreadable. This approach works well with simple GUI tools and dialog boxes that apply settings without closing.

Property Settings. GUIDE sets the following properties to create proportional resizing GUIs:

- `Units` properties of the axes and uicontrols are set to `normalized` so that these components resize and reposition as the figure window changes size.
- `Units` property of the figure is set to `characters` so the GUI displays at the correct size at run-time, based on any changes in font size.
- `Resize` figure property set to `on` (the default).
- `ResizeFcn` figure property is left unset.

User-Specified Resize Operation

You can create GUIs that accommodate resizing, while at the same time maintain the appearance and usability of your original design by programming the figure `ResizeFcn` callback routine. This callback routine allows you to recalculate the size and position of each component based on the new figure size.

This approach to handling figure resizing is used most typically in GUI-based applications that require user interaction on an ongoing basis. Such an application might contain axes for displaying data and various components whose position and size are critical to the successful use of the interface.

Property Settings. GUIDE sets the following properties to implement this style of GUI:

- `Units` properties of the figure, axes, and UIcontrols are set to `characters` so the GUI displays at the correct size at run-time.
- `Resize` figure property is set to `on` (the default).
- `ResizeFcn` figure property requires a callback routine to handle resizing.

See “The Address Book Resize Function” on page 6-43 for an example of a user-written resize function.

Command-Line Accessibility

You can restrict access to the GUI figure handle from the command line with the **Command-line accessibility** options. This prevents users from inadvertently changing the appearance of the GUI by entering commands, such as `plot`, that alter the current figure. With the default option, which is **Callback (GUI becomes Current Figure within Callbacks)**, the GUI can only become the MATLAB current figure, by the command `gcf`, while a callback is executing.

There may be occasions when you want the GUI figure handle to be accessible from the command line. For example, you might want the GUI to display plots created at the command line. In this case, you should select **On (GUI may become Current Figure from Command Line)**.

Access Options

There are four options for **Command line accessibility**:

- **Callback (GUI becomes Current Figure within Callbacks)**
- **Off (GUI never becomes Current Figure)**
- **On (GUI may become Current Figure from Command Line)**
- **Other (Use settings from Property Inspector)**

The following table summarizes how the four **Command line accessibility** options configure `HandleVisibility` and `IntegerHandle` in the Property Inspector (see Figure Properties That Control Access):

Option	Handle Visibility	Integer Handle
Callback	`Callback`	`off`
Off	`off`	`off`
On	`on`	`on`
Other	user specifies	user specifies

Figure Properties That Control Access

There are two figure properties that control command-line accessibility of the figure:

- `HandleVisibility` — Determines whether the figure's handle is visible to commands that attempt to access the current figure.
- `IntegerHandle` — Determines if a figure's handle is an integer or a floating point value.

HandleVisibility — Callback. Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles. You should use this option if your GUI contains axes.

HandleVisibility — Off. Setting the `HandleVisibility` property to `off` removes the handle of the figure from the list of root object children so it will not become the current figure (which is the target for graphics output). The handle remains valid, however, so a command that specifies the handle explicitly still works (such as `close(1)`). However, you cannot use commands that operate only on the current figure or axes. These commands include `xlabel`, `ylabel`, `zlabel`, `title`, `gca`, `gcf`, and `findobj`.

HandleVisibility — On. Handles are always visible when `HandleVisibility` is `on`.

IntegerHandle. Setting the `IntegerHandle` property to `off` causes MATLAB to assign nonreusable real-number handles (e.g., 67.0001221...) instead of integers. This greatly reduces the likelihood of someone accidently performing an operation on the figure.

Using findobj

When you set the **Command-line accessibility** to `off`, the handle of the GUI figure is hidden. This means you cannot use `findobj` to locate the handles of the uicontrols in the GUI. As an alternative, the GUI M-file creates an object `handle` structure that contains the `handles` of each uicontrol in the GUI and passes this structure to subfunctions.

Generate FIG-File and M-File

Select **Generate FIG-file and M-file** in the **GUI Options** dialog if you want GUIDE to create both the FIG-file and the GUI M-file (this is the default). Once you have selected this option, you can select any of the following items in the frame to configure the M-file:

- Generate callback function prototypes
- Application allows only one instance to run (singleton)
- Use system color scheme for background

Generate Callback Function Prototypes

When you select **Generate callback function prototypes** in the **GUI Options** dialog, GUIDE adds a subfunction to the GUI M-file for any component you add to the GUI (note that frame, axes, and static text components do not use their `Callback` property and therefore have no subfunction). You must then write the code for the callback in this subfunction.

GUIDE also adds a subfunction whenever you edit a callback routine from the Layout Editor's right-click context menu and when you add menus to the GUI using the Menu Editor.

Naming Callback Subfunctions

When you add a component to your GUI layout, GUIDE assigns a value to its `Tag` property, which is then used to generate the name of the callback.

For example, the first push button you add to the layout is tagged `pushbutton1`. When generating the M-file, GUIDE adds a callback subfunction called `pushbutton1_Callback`. If you define a `ButtonDownFcn` for the same push button, GUIDE names its subfunction `pushbutton1_ButtonDownFcn`.

Callback Function Syntax

The callback function syntax is of the form

```
function objectTag_Callback(hObject, eventdata, handles)
```

The arguments are listed in the following table.

`hObject`	The handle of the object whose callback is executing.
`eventdata`	Empty — reserved for future use.
`handles`	A structure containing the handles of all components in the GUI whose fieldnames are defined by the object's `Tag` property. Can also be used to pass data to other callback functions or the command line.

For example, if you create a layout having a push button whose `Tag` property is set to `pushbutton1`, then GUIDE generates the following subfunction header in the GUI M-file.

```
function pushbutton1_Callback(hObject, eventdata, handles)
```

Assigning the Callback Property String

When you first add a component to your GUI layout, its `Callback` property is set to the string `%automatic`. This string signals GUIDE to replace it with one that calls the appropriate callback subfunction in the GUI M-file when you save or run the GUI. For example, GUIDE sets the `Callback` property for `pushbutton1` uicontrol to

```
my_gui('pushbutton1_Callback',gcbo,[],guidata(gcbo))
```

where

- `my_gui` is the name of the GUI M-file.
- `pushbutton1_Callback` is the name of the callback routine subfunction defined in `my_gui`.
- `gcbo` is a command that returns the handle of the callback object (i.e., `pushbutton1`).
- `[]` is a place holder for the currently unused `eventdata` argument.
- `guidata(gcbo)` returns the `handles` structure.

See "Callback Function Syntax" on page 3-26 for more information on callback function arguments and "Renaming GUI Files and Tags" on page 3-44 for more information on how to change the names used by GUIDE.

GUI Allows Only One Instance to Run (Singleton)

This option allows you to select between two behaviors for the GUI figure:

- Allow MATLAB to display only one instance of the GUI at a time.
- Allow MATLAB to display multiple instances of the GUI.

If you allow only one instance, MATLAB reuses the existing GUI figure whenever the command to run the GUI is issued. If a GUI already exists, MATLAB brings it to the foreground rather than creating a new figure.

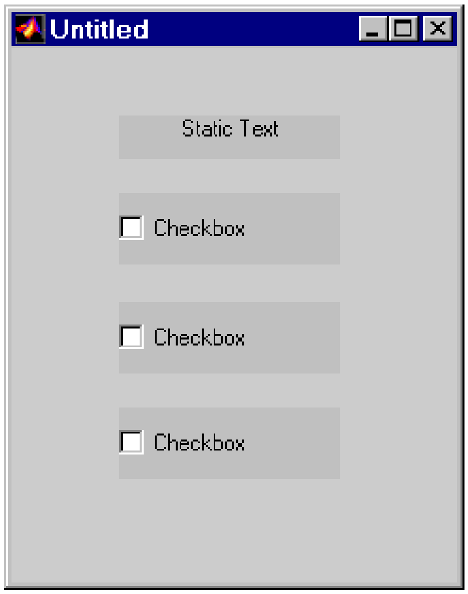
If you clear this option, MATLAB creates a new GUI figure whenever you issue the command to run the GUI.

Using the System Background Colors

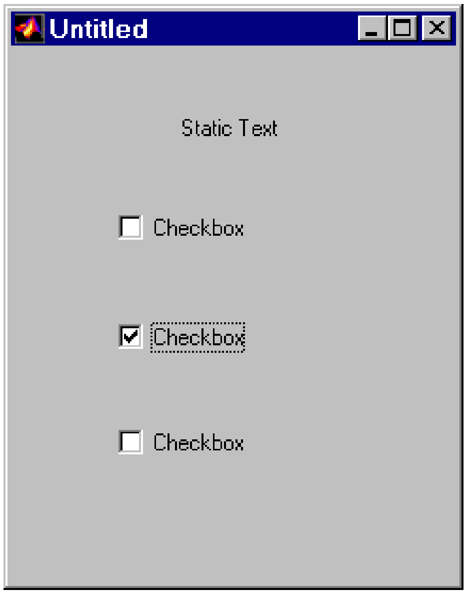
The color used for GUI components varies on different computer systems. This option enables you to make the figure background color the same as the default uicontrol background color, which is system dependent.

If you select **Use system color scheme for background** (the default), GUIDE changes the figure background color to match the color of the GUI components.

The following figures illustrate the results with and without system color matching.



Without system color matching



With system color matching

Generate FIG-File Only

Select **Generate FIG-file only** in the **GUI Options** dialog if you do not want GUIDE to generate the M-file. When you save the GUI from the Layout Editor, GUIDE creates a FIG-file, which you can redisplay using the `open` command.

When you select this option, you must set the `Callback` property of each component in your GUI to a string that MATLAB can evaluate and perform the desired action. This string can be an expression or the name of an M-file.

Select this option if you want to use a completely different programming style than that provided by the GUI M-file.

User Interface Controls

The Layout Editor component palette contains the user interface controls that you can use in your GUI. These components are MATLAB uicontrol objects and are programmable via their `Callback` properties. This section provides information on these components.

Push Buttons	Sliders
Toggle Buttons	Frames
Radio Buttons	List Boxes
Check Boxes	Pop-Up Menus
Edit Text	Axes
Static Text	Figures

Push Buttons

Push buttons generate an action when clicked (e.g., an **OK** button may close a dialog box and apply settings). When you click the mouse on a push button, it appears depressed; when you release the mouse, the button appears raised; and its callback executes.

Properties to Set

- `String` — Set this property to the character string you want displayed on the push button.
- `Tag` — GUIDE uses the `Tag` property to name the callback subfunction in the GUI M-file. Set `Tag` to a descriptive name (e.g., `close_button`) before activating the GUI.

Programming the Callback

When the user clicks the push button, its callback executes. Push buttons do not return a value or maintain a state.

Toggle Buttons

Toggle buttons generate an action and indicate a binary state (e.g., on or off). When you click on a toggle button, it appears depressed and remains depressed until you release the mouse button, at which point the callback executes. A subsequent mouse click returns the toggle button to the raised state and again executes its callback.

Programming the Callback

The callback routine needs to query the toggle button to determine what state it is in. MATLAB sets the `Value` property equal to the `Max` property when the toggle button is depressed (`Max` is 1 by default) and equal to the `Min` property when the toggle button is not depressed (`Min` is 0 by default).

From the GUI M-file

The following code illustrates how to program the callback in the GUI M-file.

```
function togglebutton1_Callback(hObject, eventdata, handles)
button_state = get(hObject,'Value');
if button_state == get(hObject,'Max')
    % toggle button is pressed
elseif button_state == get(hObject,'Min')
    % toggle button is not pressed
end
```

Adding an Image to a Push Button or Toggle Button

Assign the `CData` property an m-by-n-by-3 array of RGB values that define a truecolor image. For example, the array a defines 16-by-128 truecolor image using random values between 0 and 1 (generated by `rand`).

```
a(:,:,1) = rand(16,128);
a(:,:,2) = rand(16,128);
a(:,:,3) = rand(16,128);
set(hObject,'CData',a)
```

See `ind2rgb` for information on converting an `(X, MAP)` image to an `RGB` image.

Radio Buttons

Radio buttons are similar to check boxes, but are intended to be mutually exclusive within a group of related radio buttons (i.e., only one button is in a selected state at any given time). To activate a radio button, click the mouse button on the object. The display indicates the state of the button.

Implementing Mutually Exclusive Behavior

Radio buttons have two states — selected and not selected. You can query and set the state of a radio button through its `Value` property:

- `Value = Max`, button is selected (`1` by default)
- `Value = Min`, button is not selected (`0` by default)

To make radio buttons mutually exclusive within a group, the callback for each radio button must set the `Value` property to `0` on all other radio buttons in the group. MATLAB sets the `Value` property to `1` on the radio button clicked by the user.

The following subfunction, when added to the GUI M-file, can be called by each radio button callback. The argument is an array containing the handles of all other radio buttons in the group that must be cleared.

```
function mutual_exclude(off)
set(off,'Value',0)
```

Obtaining the Radio Button Handles. The handles of the radio buttons are available from the `handles` structure, which contains the handles of all components in the GUI.

The following code shows the call to `mutual_exclude` being made from the first radio button's callback in a group of four radio buttons:

```
function radiobutton1_Callback(hObject, eventdata, handles)
off =
[handles.radiobutton2,handles.radiobutton3,handles.radiobutton4]
;
mutual_exclude(off)
% Continue with callback
    .
    .
    .
```

After setting the radio buttons to the appropriate state, the callback can continue with its implementation-specific tasks.

Check Boxes

Check boxes generate an action when checked and indicate their state as checked or not checked. Check boxes are useful when providing the user with a number of independent choices that set a mode (e.g., display a toolbar or generate callback function prototypes).

The `Value` property indicates the state of the check box by taking on the value of the `Max` or `Min` property (1 and 0 respectively by default):

- `Value = Max`, box is checked.
- `Value = Min`, box is not checked.

You can determine the current state of a check box from within its callback by querying the state of its `Value` property, as illustrated in the following example:

```
function checkbox1_Callback(hObject, eventdata, handles)
if (get(hObject,'Value') == get(hObject,'Max'))
   % then checkbox is checked-take approriate action
else
   % checkbox is not checked-take approriate action
end
```

Edit Text

Edit text controls are fields that enable users to enter or modify text strings. Use edit text when you want text as input. The `String` property contains the text entered by the user.

To obtain the string typed by the user, get the `String` property in the callback.

```
function edittext1_Callback(hObject, eventdata, handles)
user_string = get(hObject,'string');
% proceed with callback...
```

Single or Multiple Line Selection

The values of the `Min` and `Max` properties determine whether users can select single or multiple lines in the edit text:

- If `Max    Min > 1`, users can select multiple lines.
- If `Max    Min <= 1`, users can select only a single line.

For example, setting Max to 2, with the default value of 0 for Min, enables users to select multiple lines

Obtaining Numeric Data from an Edit Test Component

MATLAB returns the value of the edit text String property as a character string. If you want users to enter numeric values, you must convert the characters to numbers. You can do this using the str2double command, which converts strings to doubles. If the user enters nonnumeric characters, str2double returns NaN.

You can use the following code in the edit text callback. It gets the value of the String property and converts it to a double. It then checks whether the converted value is NaN (isnan), indicating the user entered a nonnumeric character and displays an error dialog (errordlg).

```
function edittext1_Callback(hObject, eventdata, handles)
user_entry = str2double(get(hObject,'string'));
if isnan(user_entry)
      errordlg('You must enter a numeric value','Bad Input','modal')
end
% proceed with callback...
```

Triggering Callback Execution

On UNIX systems, clicking on the menu bar of the figure window causes the edit text callback to execute. However, on Microsoft Windows systems, if an editable text box has focus, clicking on the menu bar does not cause the editable text callback routine to execute. This behavior is consistent with the respective platform conventions. Clicking on other components or the background of the GUI executes the callback.

Static Text

Static text controls display lines of text. Static text is typically used to label other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively and there is no way to invoke the callback routine associated with it.

Sliders

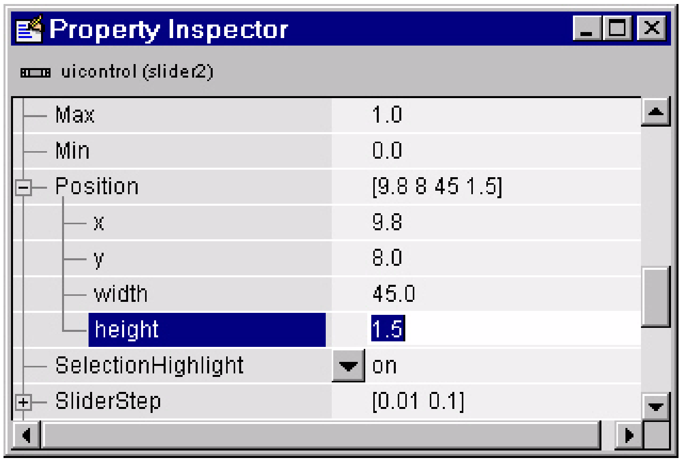
Sliders accept numeric input within a specific range by enabling the user to move a sliding bar. Users move the bar by pressing the mouse button and dragging the slide, by clicking in the trough, or by clicking an arrow. The location of the bar indicates a numeric value.

Slider Orientation

You can orient the slider either horizontally or vertically by setting the relative width and height of the `Position` property:

- Horizontal slider — Width is greater than height.
- Vertical slider — Height is greater than width.

For example, these settings create a horizontal slider.



uicontrol (slider2)

Max	1.0
Min	0.0
Position	[9.8 8 45 1.5]
x	9.8
y	8.0
width	45.0
height	1.5
SelectionHighlight	on
SliderStep	[0.01 0.1]

Current Value, Range, and Step Size

There are four properties that control the range and step size of the slider:

- `Value` contains the current value of the slider.
- `Max` defines the maximum slider value.
- `Min` defines the minimum slider value.
- `SliderStep` specifies the size of a slider step with respect to the range.

The Value property contains the numeric value of the slider. You can set this property to specify an initial condition and query it in the slider's callback to obtain the value set by the user. For example, your callback could contain the statement

```
slider_value = get(handles.slider1,'Value');
```

The Max and Min properties specify the slider's range (Max - Min).

The SliderStep property controls the amount the slider Value property changes when you click the mouse on the arrow button or on the slider trough. Specify SliderStep as a two-element vector. The default, [0.01 0.10], provides a 1 percent change for clicks on an arrow and a 10 percent change for clicks in the trough. The actual step size is a function of the slider step and the slider range.

Designing a Slider

Suppose you want to create a slider with the following behavior:

- Slider range = 5 to 8
- Arrow step size = 0.4
- Trough step size = 1
- Initial value = 6.5

From these values you need to determine and set the Max, Min, SliderStep, and Value properties. You can do this by adding the following code to the initialization section of the GUI M-file (after the creation of the handles structure):

```
slider_step(1) = 0.4/(8-5);
slider_step(2) = 1/(8-5);
set(handles.slider1,'sliderstep',slider_step,...
     'max',8,'min',5,'Value',6.5)
```

You can also assign the slider properties using the Property Inspector:

- `SliderStep, X .133`
- `SliderStep, Y .333`
- `Max 8`
- `Min 5`

- `Value 6.5`

Triggering Callback Execution

The slider callback is executed when the user releases the mouse button.

Frames

Frames are boxes that enclose regions of a figure window. Frames can make a user interface easier to understand by visually grouping related controls. Frames have no callback routines associated with them and only uicontrols can appear within frames (axes cannot).

Placing Components on Top of Frames

Frames are opaque. If you add a frame after adding components that you want to be positioned within the frame, you need to bring forward those components. Use the **Bring to Front** and **Send to Back** operations in the **Layout** menu for this purpose.

List Boxes

List boxes display a list of items and enable users to select one or more items.

The `String` property contains the list of strings displayed in the list box. The first item in the list has an index of 1. Using the Property Inspector, enter the list box items, one per line.

The `Value` property contains the index into the list of strings that corresponds to the selected item. If the user selects multiple items, then `Value` is a vector of indices.

By default, the first item in the list is highlighted when the list box is first displayed. If you do not want any item highlighted, then set the `Value` property to empty, `[]`. This works only when multiple selection is enabled.

The `ListboxTop` property defines which string in the list displays as the top most item when the list box is not large enough to display all list entries. `ListboxTop` is an index into the array of strings defined by the `String` property and must have a value between 1 and the number of strings. Noninteger values are fixed to the next lowest integer.

Single or Multiple Selection

The values of the `Min` and `Max` properties determine whether users can select single or multiple items in the list box:

- If `Max - Min > 1`, the user can select multiple items.
- If `Max - Min <= 1`, the user can only select a single item.

See "Single or Multiple Line Selection" on page 3-33.

Selection Type

List boxes differentiate between single and double clicks on an item and set the figure `SelectionType` property to `normal` or `open` accordingly. See "Triggering Callback Execution" on page 3-34 for information on how to program multiple selection.

Triggering Callback Execution

MATLAB evaluates the list box's callback after the mouse button is released or a keypress event (including arrow keys) that changes the `Value` property (i.e., any time the user clicks on an item, but not when clicking on the list box scrollbar). This means the callback is executed after the first click of a double-click on a single item or when the user is making multiple selections.

In these situations, you need to add another component, such as a **Done** button (push button) and program its callback routine to query the list box `Value` property (and possibly the figure `SelectionType` property) instead of creating a callback for the list box. If you are using the automatically generated M-file option, you need to either:

- Set the list box `Callback` property to the empty string (`''`) and remove the callback subfunction from the GUI M-file.
- Leave the callback subfunction stub in the GUI M-file empty so that no code executes when users click on list box items.

The first choice is best if you are sure you will not use the list box callback and you want to minimize the size and efficiency of the GUI M-file. However, if you think you may want to define a callback for the list box at some time, it is simpler to leave the callback stub in the M-file.

List Box Examples

See the following examples for more information on using list boxes:

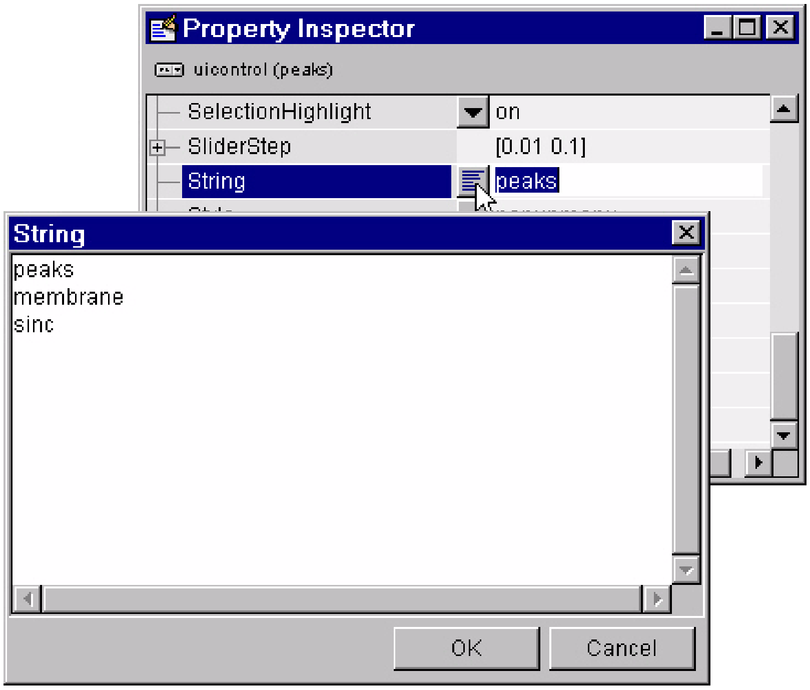
- "List Box Directory Reader" on page 6-9 — Shows how to creates a GUI that displays the contents of directories in a list box and enables users to open a variety of file types by double-clicking on the filename.
- "Accessing Workspace Variables from a List Box" on page 6-15 — Shows how to access variables in the MATLAB base workspace from a list box GUI.

Pop-Up Menus

Pop-up menus open to display a list of choices when users click the arrow.

Adding Items to the Popup Menu

The `String` property contains the list of string displayed in the pop-up menu. Use the Property Inspector to add items to the pop-up menu by typing one per line in the `String` edit box:



```
peaks
membrane
sinc
```

Determine Which Item Is Selected

The `Value` property contains the index of the selected item. For example, if the `String` contained the three items, `peaks`, `membrane`, and `sinc` and the `Value` property has a value of 2, then the item selected is `membrane`.

When not open, a pop-up menu displays the current choice, which is determined by the index contained in the `Value` property. The first item in the list has an index of 1.

Pop-up menus are useful when you want to provide users with a number of mutually exclusive choices, but do not want to take up the amount of space that a series of radio buttons requires.

Programming the Pop-Up Menu

You can program the popup menu callback to work by checking only the index of the item selected (contained in the `Value` property) or you can obtain the actual string contained in the selected item.

This callback checks the index of the selected item and uses a switch statement to take action based on the value. If the contents of the popup menu is fixed, then you can use this approach.

```
function popupmenu1_Callback(hObject, eventdata, handles)
val = get(hObject,'Value');
switch val
case 1
% The user selected the first item
case 2
% The user selected the second item
% etc.
```

This callback obtains the actual string selected in the pop-up menu. It uses the value to index into the list of strings. This approach may be useful if your program dynamically loads the contents of the pop-up menu based on user action and you need to obtain the selected string. Note that it is necessary to convert the value returned by the `String` property from a cell array to a string.

```
function popupmenu1_Callback(hObject, eventdata, handles)
val = get(hObject,'Value');
string_list = get(hObject,'String');
selected_string = string_list{val}; % convert from cell array to
string
```

```
% etc.
```

Enabling or Disabling Controls

You can control whether a control responds to mouse button clicks by setting the `Enable` property. Controls have three states:

- on — The control is operational.
- off — The control is disabled and its label (set by the `string` property) is grayed out.
- inactive — The control is disabled, but its label is not grayed out.

When a control is disabled, clicking on it with the left mouse button does not execute its callback routine. However, the left-click causes two other callback routines to execute:

1. First the figure `WindowButtonDownFcn` callback executes
2. Then the control's `ButtonDownFcn` callback executes

A right mouse button click on a disabled control posts a context menu, if one is defined for that control. See the `Enable` property description for more details.

Axes

Axes enable your GUI to display graphics (e.g., graphs and images). Like all graphics objects, axes have properties that you can set to control many aspects of its behavior and appearance. See "Axes Properties" in the MATLAB Graphics documentation for general information on axes objects.

Axes Callbacks

Axes are not uicontrol objects, but can be programmed to execute a callback when users click a mouse button in the axes. Use the axes `ButtonDownFcn` property to define the callback.

Plotting to Axes in GUIs

GUIs that contain axes should ensure the **Command-line accessibility** option in the **GUI Options** dialog is set to **Callback** (the default). This enables you to issue plotting commands from callbacks without explicitly specifying the target axes. See “Command-Line Accessibility” on page 3-23 for more information about how this option works.

Note If your GUI is opened as the result of another GUI’s callback, you might need to explicitly specify the target axes. See “GUIs with Multiple Axes” following.

GUIs with Multiple Axes

If a GUI has multiple axes, you should explicitly specify which axes you want to target when you issue plotting commands. You can do this using the `axes` command and the `handles` structure. For example,

```
axes(handles.axes1)
```

makes the axes whose `Tag` property is `axes1` the current axes, and therefore the target for plotting commands. You can switch the current axes whenever you want to target a different axes. See “GUI with Multiple Axes” on page 6-2 for an example that uses two axes.

Figure

Figures are the windows that contain the GUI you design with the Layout Editor. See the description of figure properties for information on what figure characteristics you can control.

Displaying Plots in a Separate Figure

To prevent a figure from becoming the target of plotting commands issued at the command line or by other GUIs, you can set the `HandleVisibility` and `IntegerHandle` properties to `off`. However, this means the figure is also hidden from plotting commands issued by your GUI.

To issue plotting commands from your GUI, you should create a figure and axes, saving the handles (you can store them in the `handles` structure). You then parent the axes to the figure and then parent the graphics objects created by the plotting command to the axes. The following steps summarize this approach:

- Save the handle of the figure when you create it.

- Create an axes, save its handle, and set its `Parent` property to the figure handle.
- Create the plot, and save the handles, and set their `Parent` properties to the handle of the axes.

The following code illustrates these steps:

```
fHandle = figure('HandleVisibility','off','IntegerHandle','off',...
      'Visible','off');
aHandle = axes('Parent',fHandle);
pHandles = plot(PlotData,'Parent',aHandle);
set(fHandle,'Visible','on')
```

Note that not all plotting commands accept property name/property value specifications as arguments. Consult the reference page for the specific command to see what arguments you can specify.

Renaming GUI Files and Tags

It is a good idea to use descriptive names for component `Tag` properties and callback subfunction names. GUIDE assigns a value to the `Tag` property of every component you insert in your layout (e.g., `pushbutton1`) and then uses this string to name the callback subfunction (e.g., `pushbutton1_Callback`). You should choose descriptive names for the `Tag` properties before activating or saving your GUI for the first time.

GUIDE automatically assigns a string to each component's `Tag` property and uses this string to

- Construct the name of the generated callback subfunctions (e.g., `tag_Callback`) when you run or save the GUI
- Set corresponding callback properties to point to the subfunctions
- Add a field to the `handles` structure containing the handle of the object (e.g., `handles.`*`tag`*)

Note Since GUIDE uses the `Tag` property to name functions and structure fields, the tag you select must be a valid MATLAB variable name. Use `isvarname` to determine if the string you want to use is valid.

Renaming GUI Files Using Save As

When you rename a GUI FIG-file, by selecting **Save As** from the Layout Editor **File** menu, GUIDE also renames the GUI M-file and resets the callback properties to properly execute the callbacks.

Changing Component Tag Properties

Once GUIDE generates the GUI M-file, if you subsequently change a component's `Tag` in the Property Inspector, GUIDE can correctly update the following items according to the new `Tag`, provided that all components have distinct tags:

- The component's callback functions in the M-file
- The component's callback properties, which you can view in the Property Inspector

- References in the M-file to the field of the `handles` structure containing the component's handle

---

Note If you change callback properties manually in the Property Inspector, as opposed to changing the `Tag`, GUIDE does not update the corresponding callbacks in the M-file, and might not be able to update the callback properties or references to the handles structure.

Updating Callback Properties and Functions After Changing Tags

You might have to update callback properties and functions manually after changing tags if you

- Assign the same tag to more than one component
- Select multiple items in the Layout Editor and change their tags

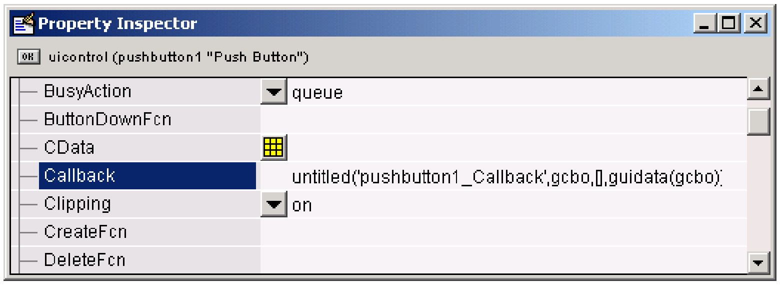
In either of these cases, you should make sure that

- Callback functions in the M-file are updated.
- Callback properties are named so that they point to the correct functions in the M-file.
- References in the M-file to the field of the `handles` structure containing the component's handle are updated.

The next section explains how to change the name of a callback property.

Changing the Name of Callback Properties

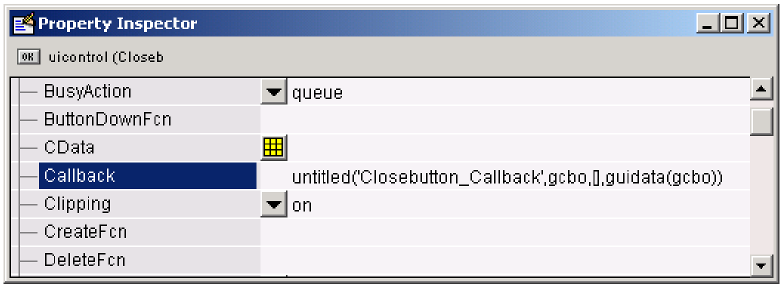
You can change the name of a callback property so that it points to the correct callback using the Property Inspector. For example, to change a push button's callback property, select the push button in the Layout Editor and then select **Property Inspector** in the **View** menu. Scroll down in the Property Inspector until you come to `Callback`, as shown in the following figure.



uicontrol (pushbutton1 "Push Button")

BusyAction	queue
ButtonDownFcn	
CData	
Callback	untitled('pushbutton1_Callback',gcbo,[],guidata(gcbo))
Clipping	on
CreateFcn	
DeleteFcn	

As shown, the callback property points to `pushbutton_Callback` in the M-file. If you need to change the callback property to `Closebutton_Callback`, simply replace the string `pushbutton_Callback` with `Closebutton_Callback` in the `Callback` field, as shown in the following figure.



uicontrol (Closeb

BusyAction	queue
ButtonDownFcn	
CData	
Callback	untitled('Closebutton_Callback',gcbo,[],guidata(gcbo))
Clipping	on
CreateFcn	
DeleteFcn	

GUIDE generates similar strings for the other callback properties.

Exporting a GUI to a Single M-File

You can export a GUI from GUIDE to a single M-file that does not require a FIG-file. This enables you to

- View the layout code for the GUI
- Run the GUI in MATLAB 6.1

To export your GUI, do the following steps:

1. Save the GUI in GUIDE, if you have not already done so.
2. Select **Export** from the **File** menu. If you changed the GUI since you last saved it, this opens a dialog informing you that exporting will save changes to your figure and M-file, and asking if you want to continue.
3. Click **OK** in the confirmation dialog.
4. Save the exported M-file in the **Save as** dialog. By default, GUIDE gives the exported M-file the name of the GUI M-file with `_export` appended.

Note If you save a large data set in the GUI figure or a uicontrol, GUIDE might also export a MAT-file containing the data in addition to exporting an M-file. The name of the MAT-file is the same as the exported M-file except for the extension `.mat`.

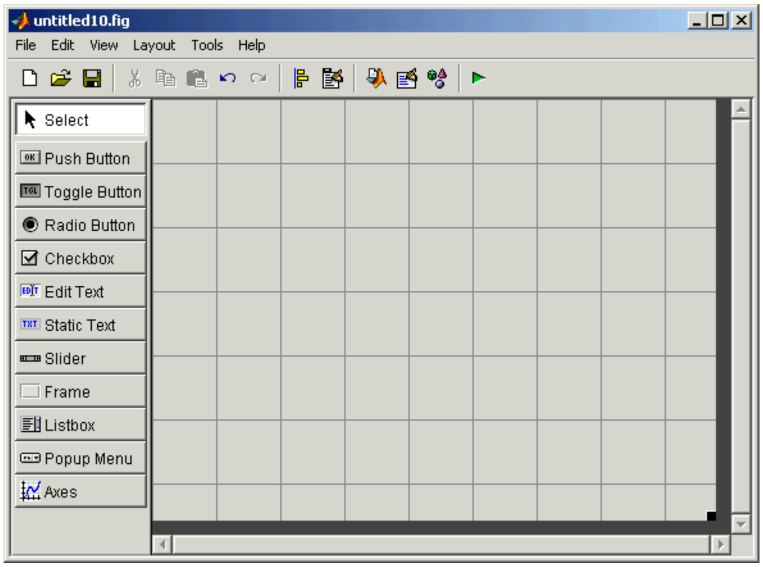
GUIDE Layout Tools

This section describes GUIDE's layout tools, which simplify the process of creating GUIs.

Laying Out GUIs — The Layout Editor

The Layout Editor enables you to select GUI components (uicontrol objects) from the *component palette*, at the left side of Layout Editor, and arrange them in the layout area, to the right. When you press the **Run** button, the functioning GUI appears outside the Layout Editor.

To start the Layout Editor, first open the GUIDE Quick Start dialog by entering `guide` at the MATLAB prompt. Click **OK** in the dialog to open a blank GUI template in the Layout Editor, as shown in the following picture.



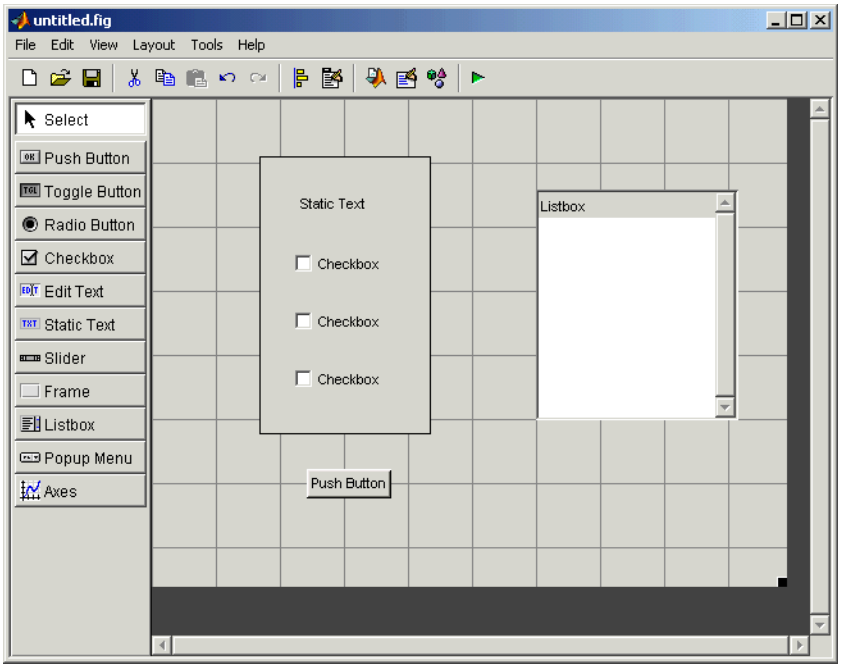
If you want to load an existing GUI for editing, type

```
guide filename.fig
```

or use **Open...** from the **File** menu on the Layout Editor.

Placing Objects In the Layout Area

Select the type of component you want to place in your GUI by clicking on it in the component palette. The cursor changes to a cross, which you can then use to select the position of the upper-left corner of the control, or you can set the size of the control by clicking in the layout area and then dragging the cursor to the lower-right corner before releasing the mouse button.



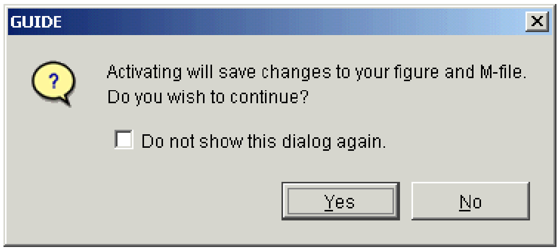
Note that components in the Layout Editor are not active. The next section describes how to generate a functioning GUI.

Running the GUI

To run the GUI you design in the Layout Editor, select **Run** in the **Tools** menu or click the green **Run** button on the toolbar.

When you run a GUI, the following occurs:

- GUIDE first prompts you to save both the M-file and FIG-file with the dialog shown in the following figure.

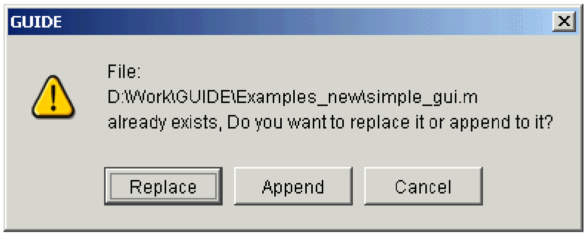


GUIDE

Activating will save changes to your figure and M-file.
Do you wish to continue?

☐ Do not show this dialog again.

- If you click **Yes** and you have not saved the GUI previously, GUIDE opens a **Save As** dialog box so you can select a name for the M-file GUIDE is going to generate.
- Clicking **Save** in the **Save As** dialog box, GUIDE saves the companion FIG-file with the same name as the M-file, but with a `.fig` extension.
- If an M-file with the same name exists, GUIDE prompts you to replace or append to the existing code in the M-file.



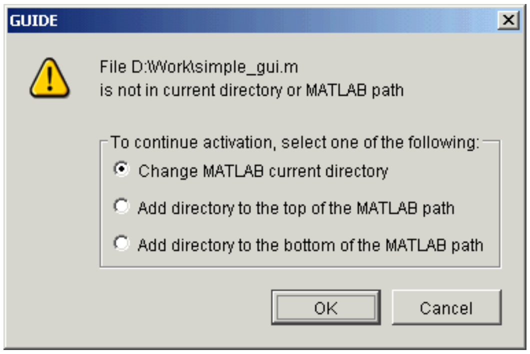
GUIDE

File:
D:\Work\GUIDE\Examples_new\simple_gui.m
already exists, Do you want to replace it or append to it?

Replace — writes over the existing file.

Append — inserts new callbacks for components added since the last save and make changes to the code based on change made from the Application Options dialog.

- If the directory in which you saved the GUI is not on the MATLAB path, GUIDE opens a dialog box with three options, as shown in the following figure.



File D:\Work\simple_gui.m
is not in current directory or MATLAB path

To continue activation, select one of the following:
(● Change MATLAB current directory) ○ Add directory to the top of the MATLAB path ○ Add directory to the bottom of the MATLAB path

Change MATLAB current directory — changes the MATLAB current directory to the directory where you saved the GUI.

Add directory to the top of the MATLAB path — adds the directory where you saved the GUI to the top of the MATLAB path

Add directory to the bottom of the MATLAB path — adds the directory where you saved the GUI to the bottom of the MATLAB path

- MATLAB executes the M-file to display the GUI. The options specified in the Application Options dialog are functional in the GUI.

Note GUIDE automatically saves both the M-file and the FIG-file when you run the GUI.

Saving the Layout

Once you have created the GUI layout, you can save it as a FIG-file (a binary file that saves the contents of a figure) using the **Save** or **Save As** item from the **File** menu. GUIDE generates the M-file automatically when you save or run the figure.

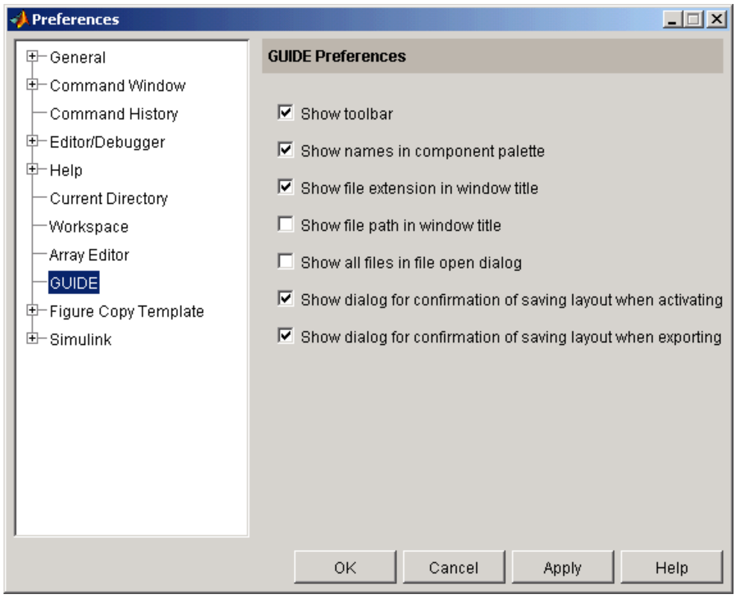
Displaying Your GUI

You can display the GUI figure using the `openfig`, `open`, or `hgload` command. These commands load FIG-files into the MATLAB workspace.

Generally, however, you launch your GUI by executing the M-file that is generated by GUIDE. This M-file contains the commands to load the GUI and provides a framework for the component callbacks. See “Configuring the GUI M-file” for more information.

Layout Editor Preferences

You can set preferences for the Layout Editor by selecting **Preferences** from the **File** menu.



GUIDE Preferences

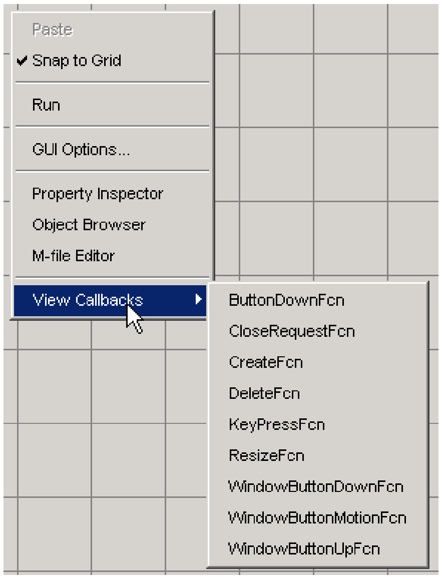
- [✓] Show toolbar
- [✓] Show names in component palette
- [✓] Show file extension in window title
- [ ] Show file path in window title
- [ ] Show all files in file open dialog
- [✓] Show dialog for confirmation of saving layout when activating
- [✓] Show dialog for confirmation of saving layout when exporting

Layout Editor Context Menus

When working in the Layout Editor, you can select an object with the left mouse button and then click the right button to display a context menu. In addition to containing items found on the Layout Editor window menu, this context menu enables you to add a subfunction to your GUI M-file for any of the additional object properties that define callback routines.

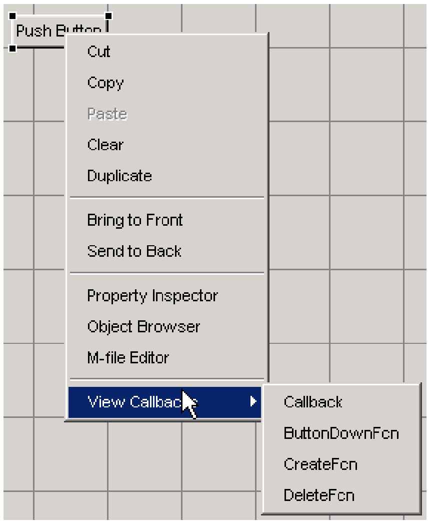
Figure Context Menus

The following picture shows the context menu associated with figure objects. Note that all the properties that define callback routines for figures are listed in the submenu.



GUI Component Context Menus

The following picture shows the context menu associated with uicontrol objects. All the properties that define callback routines for this object are listed in the lower panel. Note that while axes do not have `CallBack` properties, you can program the `ButtonDownFcn` property callback to execute whenever the user clicks the mouse over the axes.


Push Button
Cut
Copy
Paste
Clear
Duplicate
Bring to Front
Send to Back
Property Inspector
Object Browser
M-file Editor
View Callbacks
Callback
ButtonDownFcn
CreateFcn
DeleteFcn

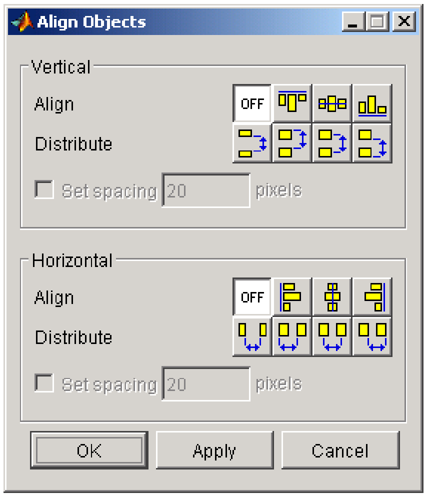
Aligning Components in the Layout Editor

You can select and drag any component or group of components within the layout area. In addition, the Layout Editor provides a number of features that facilitate more precise alignment of objects with respect to one another:

- Alignment Tool — align and distribute groups of components.
- Grid and Rulers — align components on a grid with optional snap to grid.
- Guide Lines — vertical and horizontal snap-to guides at arbitrary locations.
- Bring to Front, Send to Back, Bring Forward, Send Backward — control the front to back arrangement of components.

Aligning Groups of Components — The Alignment Tool

The Alignment Tool enables you to position objects with respect to each other and to adjust the spacing between selected objects. The specified alignment operations apply to all components that are selected when you press the **Apply** button.



The alignment tool provides two types of alignment operations:

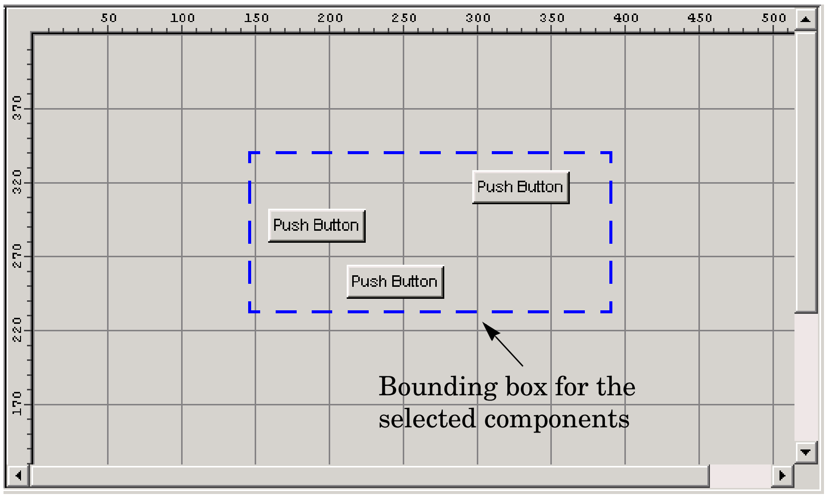
- **Align** — align all selected components to a single reference line.

- **Distribute** — space all selected components uniformly with respect to each other.

Both types of alignment can be applied in the vertical and horizontal directions. Note that, in many cases, it is better to apply alignments independently to the vertical or to the horizontal using two separate steps.

Align Options

There are both vertical and horizontal align options. Each option aligns selected components to a reference line, which is determined by the bounding box that encloses the selected objects. For example, the following picture of the layout area shows the bounding box (indicated by the dashed line) formed by three selected push buttons.



All of the align options (vertical top, center, bottom and horizontal left, center, right) place the selected components with respect to corresponding edge (or center) of this bounding box.

Distribute Options

Distributing components adds equal space between all components in the selected group. The distribute options operate in two different modes:

- Equally space selected components within the bounding box (default)

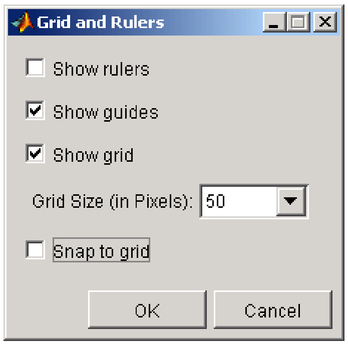
- Space selected components to a specified value in pixels (check **Set spacing** and specify a pixel value)

Both modes enable you to specify how the spacing is measured, as indicated by the button labels on the alignment tool. These options include spacing measured with respect to the following edges:

- Vertical — inner, top, center, and bottom
- Horizontal — inner, left, center, and right

Grids and Rulers

The layout area displays a grid and rulers to facilitate component layout. Grid lines are spaced at 50-pixel intervals by default and you can select from a number of other values ranging from 10 to 200 pixels. You can optionally enable *snap-to-grid*, which causes any object that is moved or resized to within 9 pixels of a grid line to jump to that line. Snap-to-grid works with or without a visible grid.


Grid and Rulers

☐ Show rulers

☑ Show guides

☑ Show grid

Grid Size (in Pixels): 50

☐ Snap to grid

Use the Grid and Rulers dialog (accessed by selecting **Grid and Rulers** from the **Layout** menu) to:

- Control visibility of rulers, grid, and guide lines
- Set the grid spacing
- Enable or disable snap-to-grid

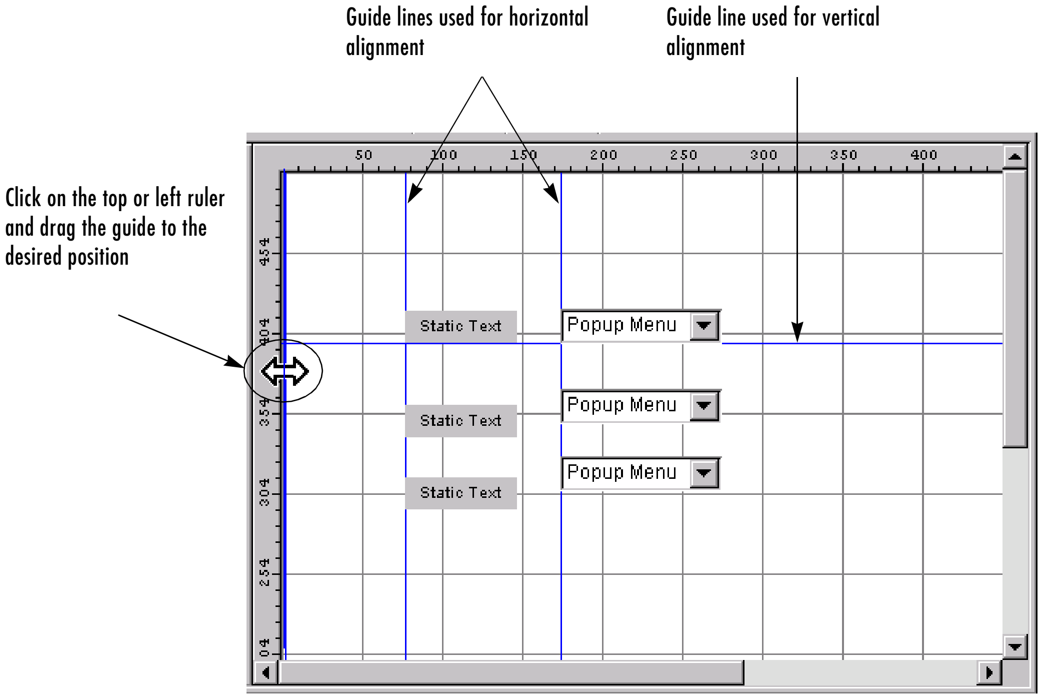
Aligning Components to Guide Lines

The Layout Editor has both vertical and horizontal snap-to guide lines. Components snap to the line when you move or resize them to within nine pixels of the line.

Guide lines are useful when you want to establish a reference for component alignment at an arbitrary location in the Layout Editor.

Creating Guide Lines

To create a guide line, click on the top or left ruler and drag the line into the layout area.


Guide lines used for horizontal alignment
Guide line used for vertical alignment
Click on the top or left ruler and drag the guide to the desired position
Static Text
Popup Menu
Static Text
Popup Menu
Static Text
Popup Menu

Front to Back Positioning

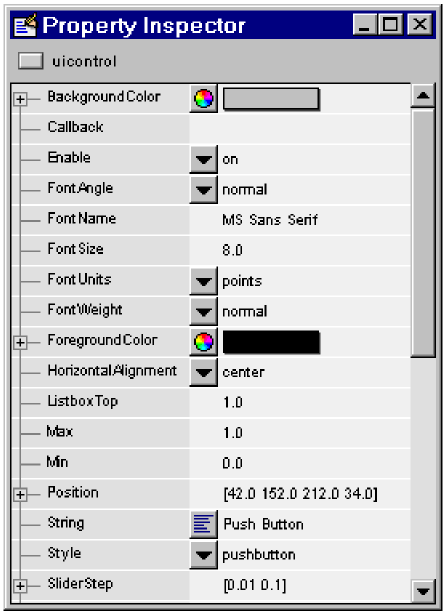
The Layout Editor provides four operations that enable you to control the front to back positioning of objects that overlap:

- Bring to Front — move the selected object(s) in front of nonselected objects (available from the right-click context menu or the **Ctrl+F** shortcut).
- Send to Back — move the selected object(s) behind nonselected objects (available from the right-click context menu or the **Ctrl+B** shortcut).
- Bring Forward — move the selected object(s) forward by one level (i.e., in front of the object directly forward of it, but not in front of all objects that overlay it).
- Send Backward — move the selected object(s) back by one level (i.e., behind of the object directly in back of it, but not behind of all objects that are behind it).

Access these operations from the **Layout** menu.

Setting Component Properties — The Property Inspector

The Property Inspector enables you to set the properties of the components in your layout. It provides a list of all settable properties and displays the current value. Each property in the list is associated with an editing device that is appropriate for the values accepted by the particular property. For example, a color picker to change the `BackgroundColor`, a pop-up menu to set `FontAngle`, and a text field to specify the `Callback` string.



BackgroundColor	⬜
Callback	
Enable	on
FontAngle	normal
FontName	MS Sans Serif
FontSize	8.0
FontUnits	points
FontWeight	normal
ForegroundColor	⬛
HorizontalAlignment	center
ListboxTop	1.0
Max	1.0
Min	0.0
Position	[42.0 152.0 212.0 34.0]
String	Push Button
Style	pushbutton
SliderStep	[0.01 0.1]

See the description of uicontrol properties for information on what values you can assign to each property and what each property does.

Displaying the Property Inspector

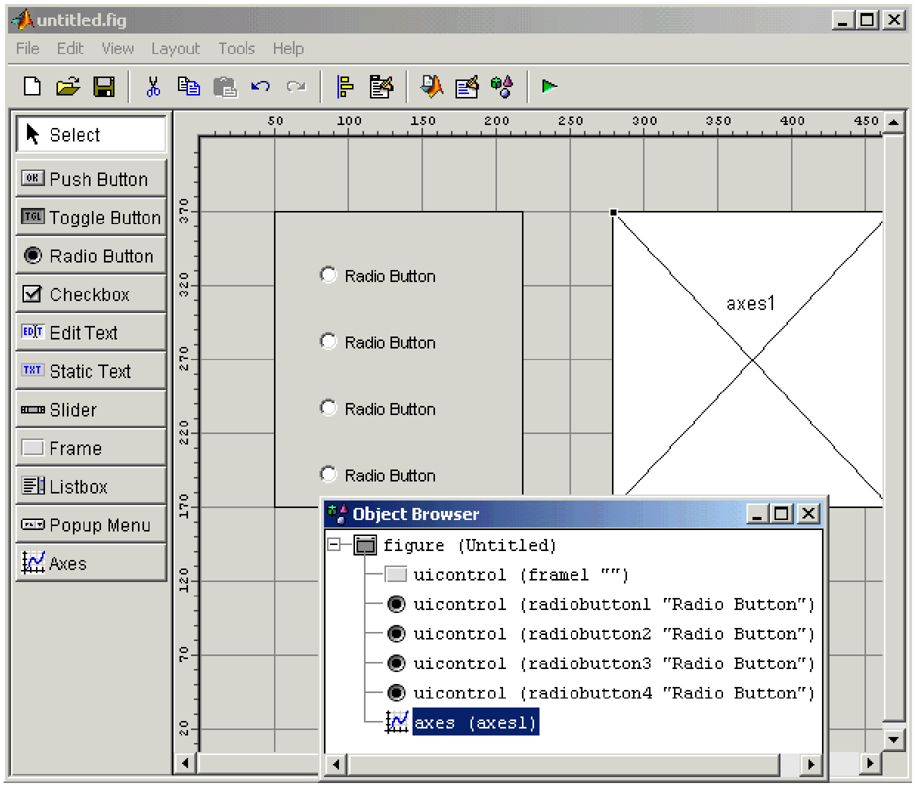
You can display the Property Inspector by:

- Double-clicking on a component in the Layout Editor.
- Selecting **Property Inspector** in the **View** menu.

- Right-clicking on a component and selecting **Inspect Properties** from the context menu.

Viewing the Object Hierarchy — The Object Browser

The Object Browser displays a hierarchical list of the objects in the figure. The following illustration shows the figure object and its child objects. The first uicontrol created was the frame. Next the radio buttons were added. Finally the axes was positioned next to the frame.



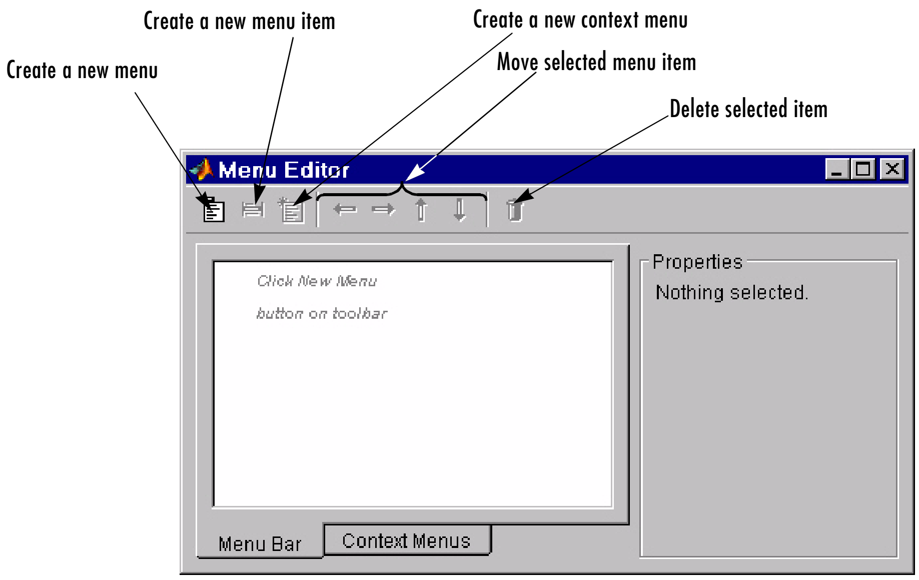
```
figure (Untitled)
  uicontrol (frame1 "")
  uicontrol (radiobutton1 "Radio Button")
  uicontrol (radiobutton2 "Radio Button")
  uicontrol (radiobutton3 "Radio Button")
  uicontrol (radiobutton4 "Radio Button")
  axes (axes1)
```

Creating Menus — The Menu Editor

MATLAB enables you to create two kinds of menus:

- Menu bar objects — menus displayed on the figure menubar
- Context menus — menus that pop up when users right-click on graphics objects

You create both types of menus using the Menu Editor, which you can access from the **Menu Editor** item on the **Tool** menu and from the Layout Editor toolbar.



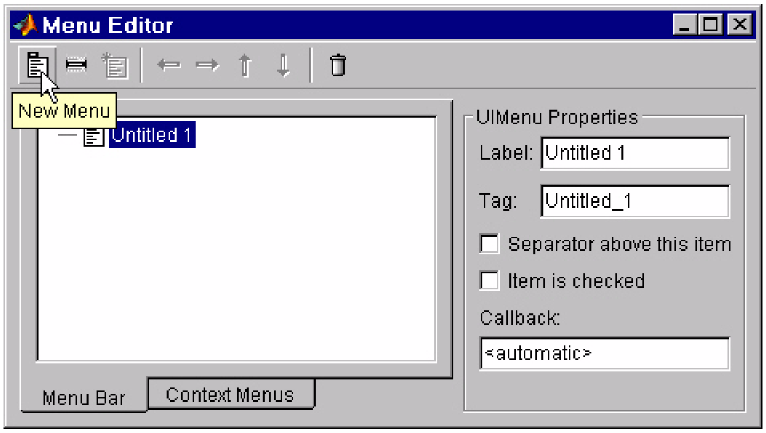
These menus are implemented with `uimenu` and `uicontextmenu` objects.

Defining Menus for the Menu Bar

When you create a menu, MATLAB adds it to the figure menubar. You can then create menu items for that menu. Each item can also have submenu items, and these items can have submenus, and so on.

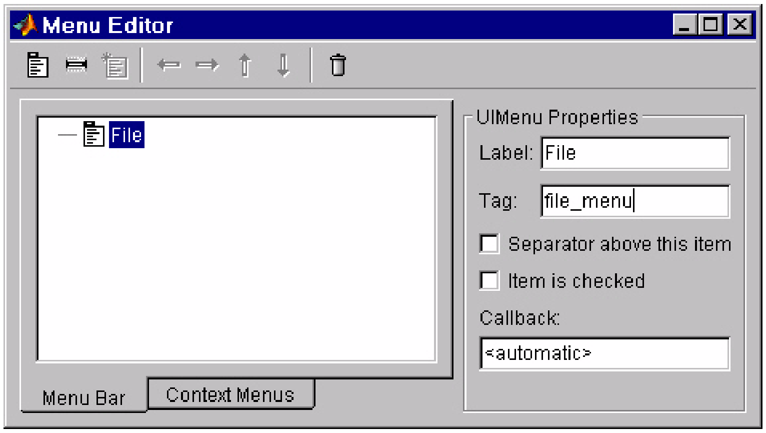
Creating a Menu

The first step is to use the New Menu tool to create a menu.



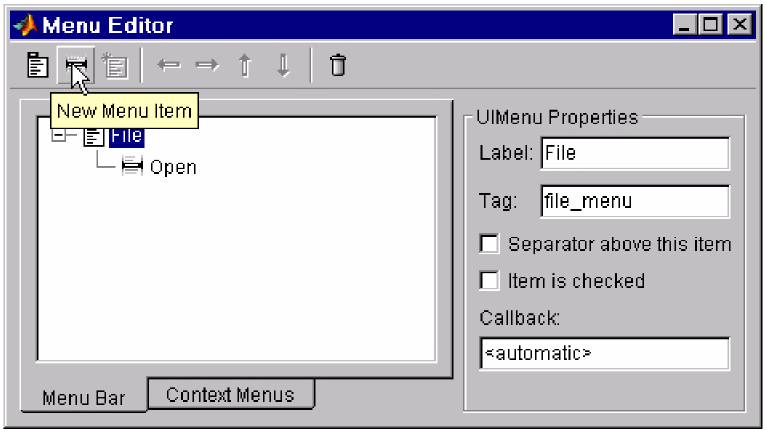
Specifying Menu Properties

When you click on the menu, text fields appear that allow you to set the `Label`, `Tag`, `Separator`, and `Checked` menu properties as well as specifying the `Callback` string.



Adding Items to the Menu

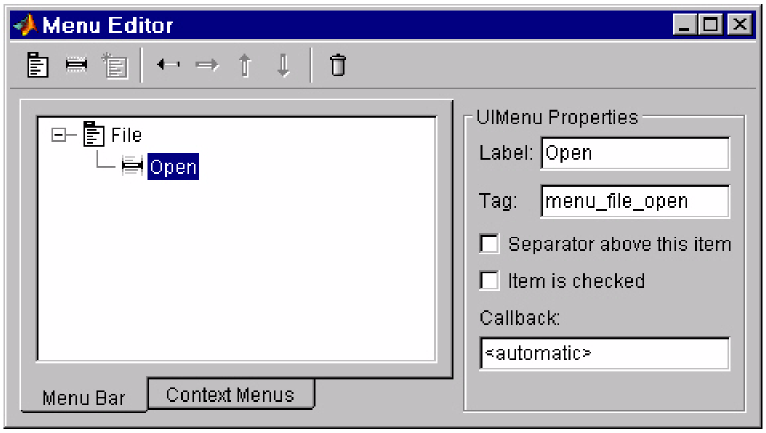
Use the New Menu Item tool to define the menu items that are displayed under the top-level menu.



UIMenu Properties
Label: File
Tag: file_menu
☐ Separator above this item
☐ Item is checked
Callback:
<automatic>

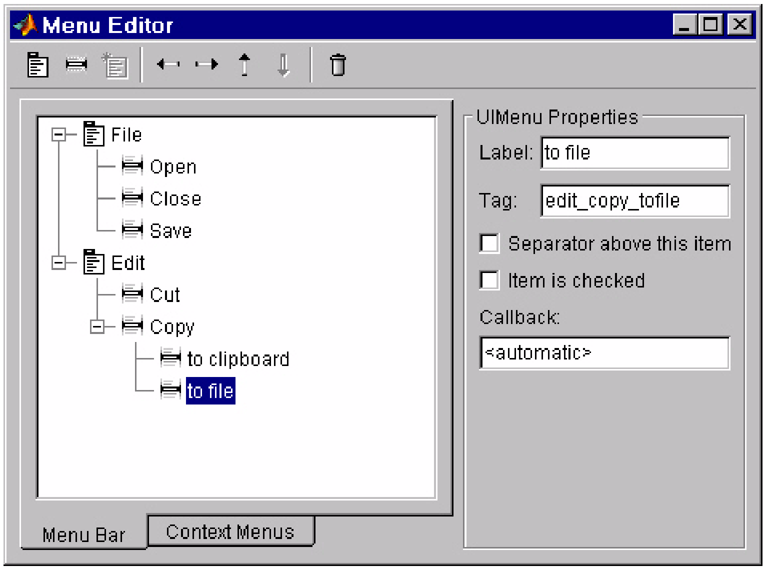
New Menu Item adds a submenu to the selected item. For example, if you add a **Print** item to the **File** menu in the illustration above, select **File** before clicking on New Menu Item.

Fill in the **Label** and **Tag** text fields for the new submenu.



UIMenu Properties
Label: Open
Tag: menu_file_open
☐ Separator above this item
☐ Item is checked
Callback:
<automatic>

Create additional levels in the same way. For example, the following picture show an Edit menu having a Copy submenu, which itself has two submenus.



UIMenu Properties

Label: to file

Tag: edit_copy_tofile

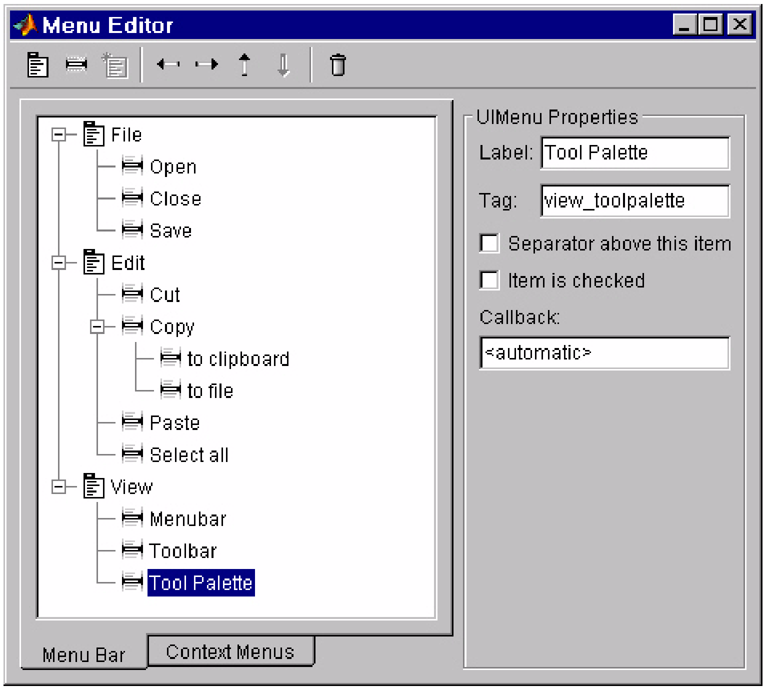
☐ Separator above this item

☐ Item is checked

Callback: <automatic>

Laying Out Three Menus

The following Menu Editor illustration shows three menus defined for the figure menubar.



Label: Tool Palette

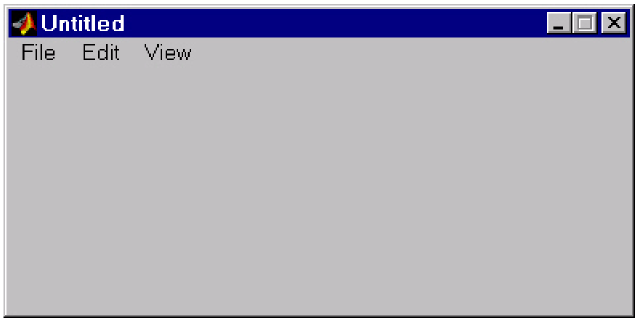
Tag: view_toolpalette

☐ Separator above this item

☐ Item is checked

Callback: <automatic>

When you run the GUI, the menus appear in the menubar.



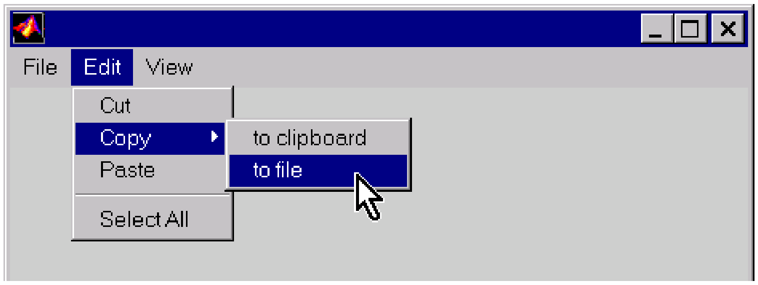
Menu Callbacks

By default, the **Callback** text field in the Menu Editor is set to the string `%automatic`. This causes GUIDE to add the empty callback subfunction to the GUI M-file when you save or run the GUI. If you change this string, GUIDE does not add a subfunction for that menu item.

Functions Generated in the GUI M-file

While the Menu Editor generates an empty callback subfunction for every menu and submenu, you may not need to program the callbacks for top-level menus. This is because clicking on a top-level menu automatically displays the submenus.

Consider the example from the previous section, as illustrated in the following picture:



When a user selects the **to file** item under the **Edit** menu **Copy** item, only the **to file** callback is required to perform the action.

Suppose, however, that only certain objects can be copied to a file. You can use the **Copy** menu's callback to enable or disable the **to file** item, depending on the type of object selected.

Syntax of the Callback Subfunction

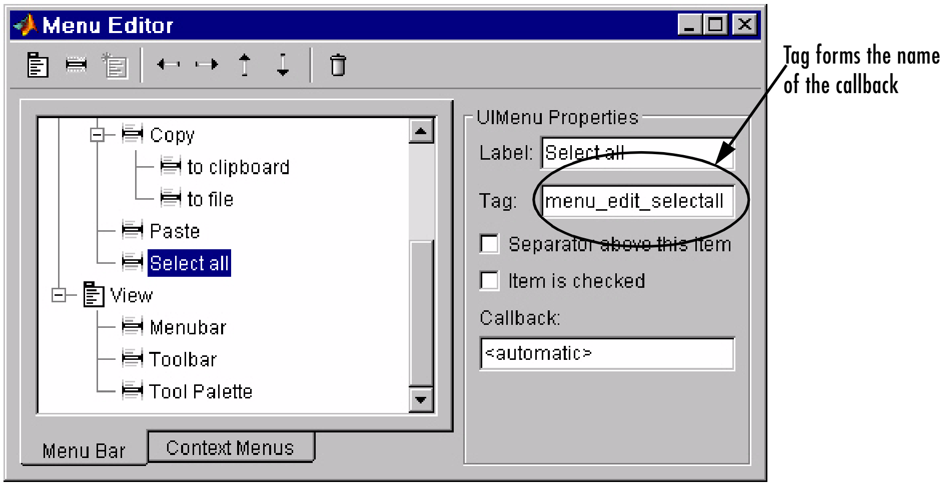
The GUI M-file contains all callbacks for the GUI, including the menu callbacks. All generated callbacks use the same syntax.

For example, using the **Select All** menu item from the previous example gives the following callback string:

```
MyGui('menu_edit_selectall_Callback',gcbo,[],guidata(gcbo))
```

where:

- `MyGui` — is the name of the GUI M-file that launches the figure containing the menus.
- `menu_edit_selectall_Callback` — is the name of the subfunction callback for the **Select All** menu item (derived from the `Tag` specified in the Menu Editor).
- `gcbo` — is the handle of the **Select All** uimenu item.
- `[]` — is an empty matrix used as a place holder for future use.
- `guidata(gcbo)` — gets the `handles` structure from the figure's application data



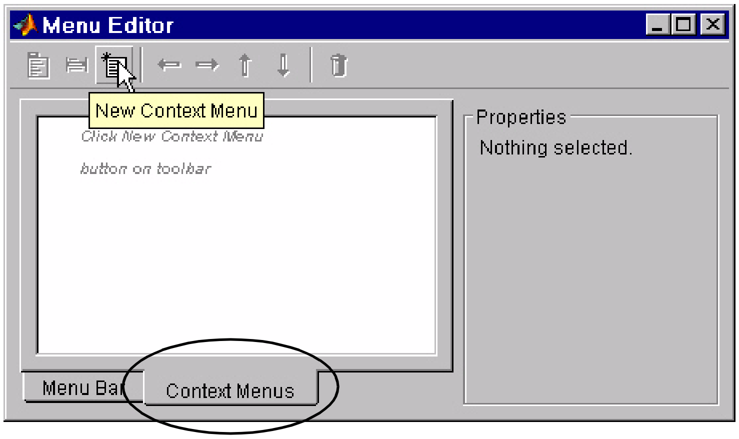
UIMenu Properties
Label: Select all
Tag: menu_edit_selectall
☐ Separator above this item
☐ Item is checked
Callback:
<automatic>

Defining Context Menus

Context menus are displayed when users right-click on the object for which the menu is defined. The Menu Editor enables you to define context menus and associate them with objects in the layout.

Creating the Parent Menu

All items in a context menu are children of a menu that is not displayed on the figure menubar. To define the parent menu, select New Context Menu from the Menu Editor's toolbar.

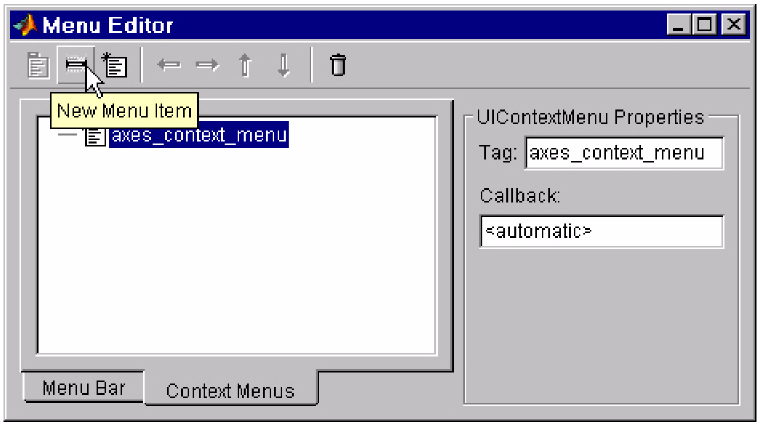


Note You must select the Menu Editor's **Context Menus** tab before you begin to define a context menu.

Select the menu and specify the `Tag` to identify the context menu (`axes_context_menu` in this example).

Adding Items to the Context Menu

Create the items that will appear in the context menu using New Menu Item on the Menu Editor's toolbar.



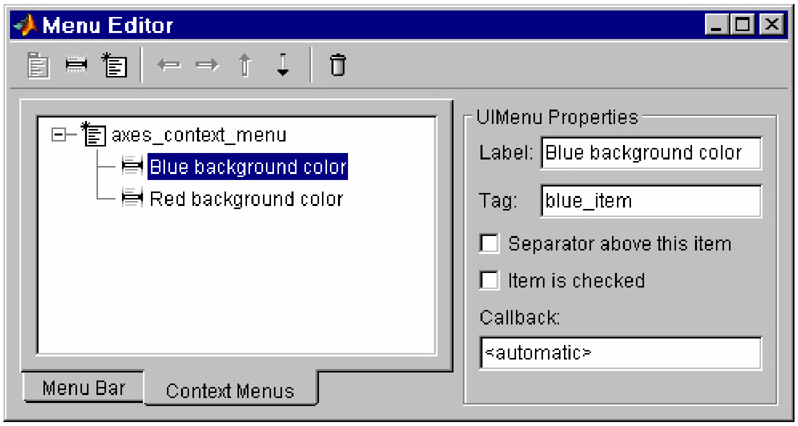
UIContextMenu Properties

Tag: axes_context_menu

Callback:

<automatic>

When you select the menu item, the Menu Editor displays text fields for you to enter the menu `Label` and `Tag` properties.



UIMenu Properties

Label: Blue background color

Tag: blue_item

☐ Separator above this item

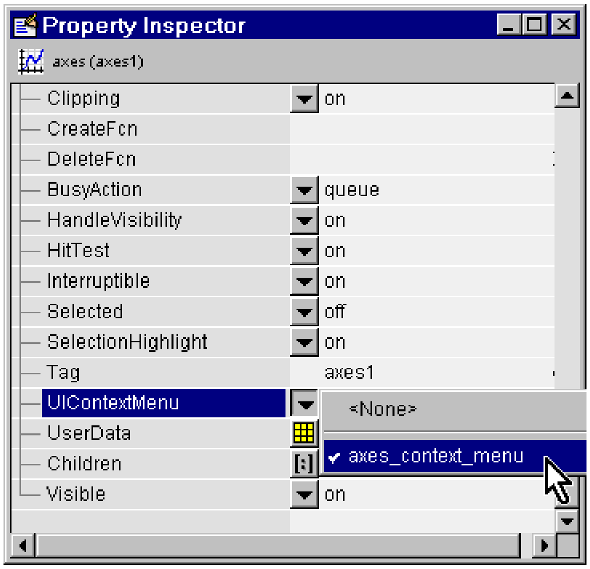
☐ Item is checked

Callback:

<automatic>

Associating the Context Menu with an Object

Select the object in the Layout Editor for which you are defining the context menu. Use the Property Inspector to set this object's `UIContextMenu` property to the desired context menu.

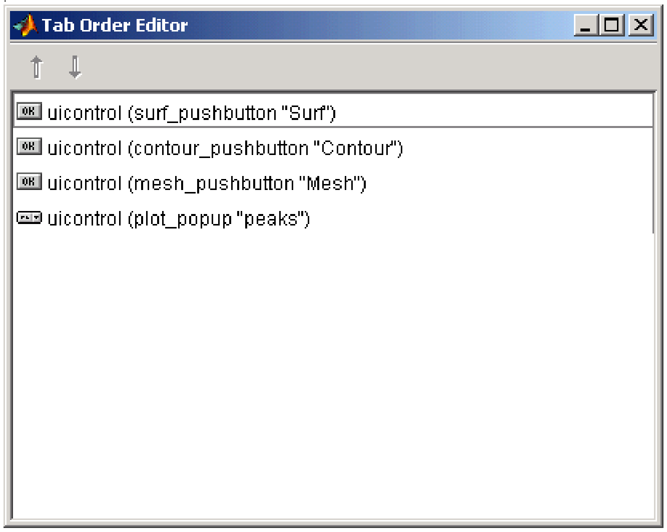


Clipping	on
CreateFcn	
DeleteFcn	
BusyAction	queue
HandleVisibility	on
HitTest	on
Interruptible	on
Selected	off
SelectionHighlight	on
Tag	axes1
UIContextMenu	<None> ✓ axes_context_menu
UserData	
Children	
Visible	on

Add a callback routine subfunction to the GUI M-file for each item in the context menu. This callback executes when users select the particular context menu item. See “The Menu Callback” for information on defining the syntax.

Setting the Tab Order — The Tab Order Editor

A GUI's tab order is the order in which components of the GUI are selected when a user presses the **Tab** key on the keyboard. When you create a GUI, GUIDE sets the tab order to be the order in which you add components in the Layout Editor. This is often not the best order for the user: if you add a component at the top of the GUI after adding one below it, pressing the **Tab** key selects the lower component before the upper one, contrary to the user's expectations. To change the tab order of GUI components, select **Tab Order Editor** in the Tools menu of the Layout Editor. For example, if you open the Tab Order Editor with the GUI described in "Example: Creating a GUI" on page 2-1, in the Layout Editor, the Tab Order Editor appears as in the following figure:



uicontrol (surf_pushbutton "Surf")
uicontrol (contour_pushbutton "Contour")
uicontrol (mesh_pushbutton "Mesh")
uicontrol (plot_popup "peaks")

The Tab Order Editor displays the components of the GUI in their current tab order. To change the tab order, select a component and press the **up** or **down** arrow to move the component up or down in the list. For example, if you want the pop-up menu, whose `Tag` is `plot_popup`, to be first in the tab order, select the pop-up menu and press the **up** arrow three times to move it to the top.

Programming GUIs

This chapter covers the following topics related to programming the M-file within the framework that GUIDE generates.

Understanding the GUI M-File

The GUI M-file generated by GUIDE controls the GUI and determines how it responds to a user's actions, such as pressing a push button or selecting a menu item. The M-file contains all the code needed to run the GUI, including the callbacks for the GUI components. While GUIDE generates the framework for this M-file, you must program the callbacks to perform the functions you want them to.

This section explains the overall structure of the M-file. The section covers the following topics:

- "Sharing Data with the Handles Structure" on page 5-2
- "Functions and Callbacks in the M-File" on page 5-3
- "Opening Function" on page 5-4
- "Output Function" on page 5-5
- "Callbacks" on page 5-6
- "Input and Output Arguments" on page 5-6

Sharing Data with the Handles Structure

When you run a GUI, the M-file creates a `handles` structure that contains all the data for GUI objects, such as controls, menus, and axes. The `handles` structure is passed as an input to each callback. You can use the `handles` structure to

- Share data between callbacks
- Access GUI data

Sharing Data

To store data that is contained in a variable `X`, set a field of the `handles` structure equal to `X` and then save the `handles` structure with the `guidata` function:

```
handles.current_data = X;
guidata(hObject,handles)
```

You can retrieve the data in any subsequent callback with the command

```
X = handles.current_data;
```

For an example of sharing data between callbacks, see “Example: Passing Data Between Callbacks” on page 5-8.

For more information about handles, see “Managing GUI Data with the Handles Structure” on page 5-8.

Accessing GUI Data

You can access any of the data for the GUI components from the `handles` structure. For example, suppose your GUI has a pop-up menu, whose `Tag` is `my_menu`, containing three items, whose `String` properties are `chocolate`, `strawberry`, and `vanilla`. You want another component in the GUI — a push button, for example — to execute a command on the currently selected menu item. In the callback for the push button, you can insert the command

```
all_choices = get(handles.my_menu, 'String')
current_choice = all_choices{get(handles.my_menu, 'Value')};
```

The command sets the value of `current_choice` to `chocolate`, `strawberry`, or `vanilla`, depending on which item is currently selected in the menu.

You can also access the data for the entire GUI from the handles structure. If the figure’s `Tag` is `figure1`, then

```
handles.figure1
```

contains the figure’s handle. For example, you can make the GUI close itself with the command

```
delete(handles.figure1)
```

For an example of closing a GUI with this command, look at the callback for the **Close** push button for the template described in “GUI with Axes and Menu” on page 3-9.

Functions and Callbacks in the M-File

You can add code to the following parts of the GUI M-file:

- Opening function — executes before the GUI becomes visible to the user.
- Output function — outputs data to the command line, if necessary.
- Callbacks — execute each time the user activates the corresponding component of the GUI.

Common Input Arguments

All functions in the M-file have the following input arguments corresponding to the handles structure:

- `hObject` — the handle to the figure or `Callback` object
- `handles` — structure with handles and user data (see `guidata`)

The `handles` structure is saved at the end of each function with the command

```
guidata(hObject, handles);
```

Additional arguments for the opening and output functions are described in the following sections.

Opening Function

The opening function contains code that is executed just before the GUI is made visible to the user. You can access all the uicontrols for the GUI in the opening function, because all objects in the GUI are created before the opening function is called. You can add code to the opening function to perform tasks that need to be done before the user has access to the GUI — for example, creating data, plots or images, or making the GUI blocking with the `uiwait` command.

For a GUI whose file name is `my_gui`, the definition line for the opening function is

```
function my_gui_OpeningFcn(hObject, eventdata, handles, varargin)
```

Besides the arguments `hObject` and `handles` (see "Common Input Arguments" preceding), the opening function has the following input arguments:

- `eventdata` reserved for a future version of MATLAB
- `varargin` command line arguments to untitled (see `varargin`)

All command line arguments are passed to the opening function via `varargin`. If you call the GUI with a property name/property value pair as arguments, the the GUI opens with the property set to the specified value. For example, `my_gui('Position', [71.8 44.9 74.8 19.7])` opens the GUI at the specified position, since `Position` is a valid figure property.

If the input argument is not a valid figure property, you must add code to the opening function to make use of the argument. For an example, look at the opening function for the modal question dialog template. The added code enables you to open the modal dialog with the syntax

```
my_gui('String','Do you want to exit?')
```

which displays the text `'Do you want to exit'` on the GUI. In this case, it is necessary to add code to the opening function because `'String'` is not a valid figure property.

Output Function

The output function returns output arguments to the command line. This is particularly useful if you want to return a variable to another GUI. For an example, see "Example: Using the Modal Dialog to Confirm an Operation" on page 3-13.

GUIDE generates the following lines of code in the output function:

```
% --- Outputs from this function are returned to the command line.
function varargout = my_gui_OutputFcn(hObject, eventdata,
handles)
% Get default command line output from handles structure
varargout{1} = handles.output;
```

If the GUI is not blocking — in other words, if the M-file does not contain the command `uiwait` — the output is simply the handle to the GUI, which is assigned to `handles.output` in the opening function.

To make the GUI return a different output — for example, if you want it to return the result of a user response, such as pressing a push button — do the following:

- Add the command `uiwait;` to the opening function to make the M-file halt execution until a user activates a component in the GUI.
- For each component of the GUI where you expect a user response, make the callback update the value of `handles.output`, and execute `uiresume.`

For example, if the GUI contains a push button whose `String` property is `'Yes'`, add the following code to its callback to make the GUI return `'Yes'` when the user presses the push button:

```
handles.output = 'Yes';
guidata(hObject, handles);
uiresume;
```

See the section "Managing GUI Data with the Handles Structure" on page 5-8 for more information about passing data with the `handles` structure.

When the GUI is called with the command

```
OUT = my_gui
```

and a user presses the **Yes** push button, the GUI returns the output `OUT = 'Yes'` to the command line.

The output `varargout`, which is a cell array, can contain any number of output arguments. By default, GUIDE creates just one output argument, `handles.output`. To create another output argument, add the command

```
varargout{2} = handles.second_output;
```

to the output function. You can set the value of `handles.second_output` in any callback and then save it with the `guidata` command.

If you want, you can choose more descriptive names than `output` or `second_output` for the fields of the handles structure corresponding to the output arguments.

Callbacks

When a user activates a component of the GUI, the GUI executes the corresponding callback. The name of the callback is determined by the component's `Tag` property. For example, a push button with the `Tag` `print_button` executes the callback

```
function print_button_Callback(hObject, eventdata, handles)
```

Input and Output Arguments

The following examples illustrate various ways to call a GUI named `my_gui` with different arguments. All arguments are passed to the opening function in the GUI M-file.

- Calling `my_gui` with no arguments opens `my_gui`.
- Calling `H = my_gui` returns the handle to `my_gui`.

- Calling `my_gui('Property', Value,...)`, where `'Property'` is a valid figure property, opens `my_gui` using the given property-value pair. You can call the GUI with more than one property-value pair.

  For example, `my_gui('Position', [71.8 44.9 74.8 19.7])` opens the GUI at the specified position, since `Position` is a valid figure property. See the reference page for `figure` for a list of figure properties.
- Calling `my_gui('My_function', hObject, eventdata, handles,...)` calls the subfunction `My_function` in the GUI M-file with the given input arguments.
- Calling `my_gui('Key_word', Value,...)`, where `'Key_word'` is any string that is *not* a valid figure property or the name of a subfunction, creates a new `my_gui`, and passes the pair `'Key_word'`, `Value` to the opening function in the GUI M-file via `varargin`.

  You can use this calling syntax to pass arguments that are not figure properties to the GUI. For example, `my_gui('Temperature', 98.6)` opens the GUI and passes the vector `['Temperature', 98.6]` to the opening function.

See “An Address Book Reader” on page 6-31 for an example that uses this syntax. The example creates a GUI called `address_book`. Calling the GUI with the syntax `address_book('book', my_contacts)` opens the GUI with the MAT-file `my_contacts`, which contains a list of names and addresses you want to display. Note that you can use any string that is not a valid figure property or the name of a callback in place of the string `'book'`. See the reference page for `figure` for a list of figure properties.

Note If you call a GUI with multiple property value pairs, all the pairs that correspond to valid figure properties must precede all the pairs that do not correspond to valid figure properties in the sequence of arguments.

Managing GUI Data with the Handles Structure

GUIDE provides a mechanism, called the *handles structure*, for storing and retrieving shared data using the same structure that contains the GUI component handles. The `handles` structure, which contains the handles of all the components in the GUI, is passed to each callback in the GUI M-file. Therefore, this structure is useful for saving any shared data. The following example illustrates this technique.

If you are not familiar with MATLAB structures, see Structures for more information.

This section covers the following topics:

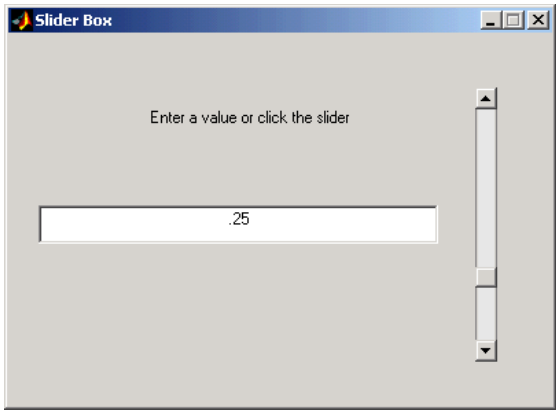
- “Example: Passing Data Between Callbacks” on page 5-8
- “Application Data” on page 5-11

Example: Passing Data Between Callbacks

This example demonstrates how to use the `handles` structure to pass data between callbacks. The example passes data between a slider and an editable text box in the following way:

- When a user moves the slider, the text box displays the slider’s current value.
- When a user types a value into the text box, the slider updates to this value.
- If a user enters a value in the text box that is out of range for the slider — that is, a value that is not between 0 an 1 — the application returns a message indicating how many times he has entered an erroneous value.

The following figure shows the GUI with a static text field above the edit text box.



Defining the Data Fields in the Opening Function

The GUI records the number of times a user enters an erroneous value in the text box and stores this number in a field of the `handles` structure. You can define this field, called `number_errors`, in the opening function as follows:

```
handles.number_errors = 0;
```

Type this command before the following line, which GUIDE automatically inserts into the opening function.

```
guidata(hObject, handles); % Save the updated structure
```

The `guidata` command saves the `handles` structure so that it can be retrieved in the callbacks.

Note To save any changes that you make to the handles structure, you must add the command `guidata(hObject, handles)` following the code that implements the changes.

Setting the Edit Text Value from the Slider Callback

The following command in the slider callback updates the value displayed in the edit text when a user moves the slider and releases the mouse button.

```
set(handles.edit1,'String',...
    num2str(get(handles.slider1,'Value')));
```

The code combines three commands:

- The `get` command obtains the current value of the slider.
- The `num2str` command converts the value to a string.
- The `set` command resets the `String` property of the edit text to the updated value.

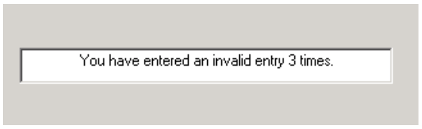
Setting the Slider Value from the Edit Text Callback

The edit text callback sets the slider's value to the number the user types in, after checking to see if it is a single numeric value between 0 and 1. If the value is out of range, then the error count is incremented and the edit text displays a message telling the user how many times they have entered an invalid number.

```
val = str2double(get(handles.edit1,'String'));
% Determine whether val is a number between 0 and 1
if isnumeric(val) & length(val)==1 & ...
   val >= get(handles.slider1,'Min') & ...
   val <= get(handles.slider1,'Max')
   set(handles.slider1,'Value',val);
else
% Increment the error count, and display it
   handles.number_errors = handles.number_errors+1;
   guidata(hObject,handles); % store the changes
   set(handles.edit1,'String',...
   ['You have entered an invalid entry ',...
num2str(handles.number_errors),' times.']);
end
```

If the user types a number between 0 and 1 in the edit box and then clicks outside the edit box, the callback sets `handles.slider1` to the new value and the slider moves to the corresponding position.

If the entry is invalid — for example, `2.5` — the GUI increments the value of `handles.number_errors` and displays a message like the following:


You have entered an invalid entry 3 times.

Application Data

Application data provides a way for applications to save and retrieve data stored with the GUI. This technique enables you to create what is essentially a user-defined property for an object. You can use this property to store data.

The GUI M-file uses application data to store the `handles` structure.

When using the GUIDE-generated M-file, it is simpler to use `guidata` than to access application data directly.

Functions for Accessing Application Data

The following functions provide access to application data.

Functions for Accessing Application-Defined Data

Function	Purpose
`setappdata`	Specify application data
`getappdata`	Retrieve named application data
`isappdata`	True if the named application data exists
`rmappdata`	Remove the named application data

Designing for Cross-Platform Compatibility

You can use specific property settings to create a GUI that behaves more consistently when run on different platforms:

- Use the default font (uicontrol `FontName` property).
- Use the default background color (uicontrol `BackgroundColor` property).
- Use figure character units (figure `Units` property).

Using the Default System Font

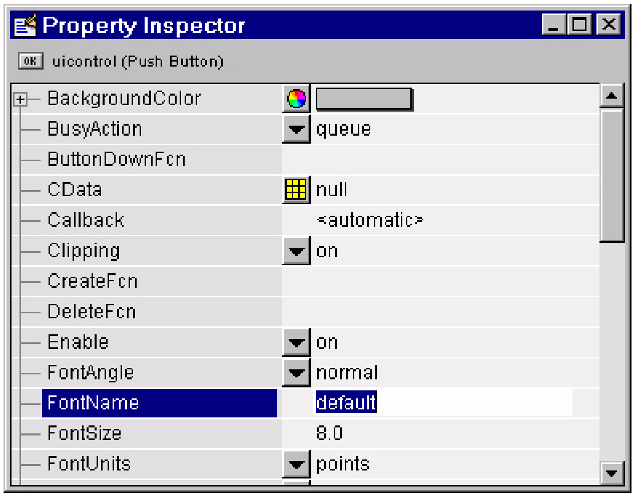
By default, uicontrols use the default font for the platform on which they are running. For example, when displaying your GUI on PCs, uicontrols uses MS San Serif. When your GUI runs on a different platform, it uses that computer's default font. This provides a consistent look with respect to your GUI and other application GUIs.

If you have set the `FontName` property to a named font and want to return to the default value, you can set the property to the string `default`. This ensures that MATLAB uses the system default at runtime.

From within the GUI M-file, use the set command. For example, if there is a push button in your GUI and its handle is stored in the `pushbutton1` field of the `handles` structure, then the statement,

```
set(handles.pushbutton1,'FontName','default')
```

sets the `FontName` property to use the system default. You can also use the Property Inspector to set this property:



BackgroundColor	
BusyAction	queue
ButtonDownFcn	
CData	null
Callback	<automatic>
Clipping	on
CreateFcn	
DeleteFcn	
Enable	on
FontAngle	normal
FontName	default
FontSize	8.0
FontUnits	points

Specifying a Fixed-Width Font

If you want to use a fixed-width font for a uicontrol, set its `FontName` property to the string `fixedwidth`. This special identifier ensures that your GUI uses the standard fixed-width font for the target platform.

You can find the name of the fixed-width font that is used on a given platform by querying the root `FixedWidthFontName` property.

```
get(0,'FixedWidthFontName')
```

Using a Specific Font Name

You can specify an actual font name (such as Times or Courier) for the `FontName` property. However, doing so may cause your GUI to look poorly when run on a different computer. If the target computer does not have the specified font, it will substitute another font that may not look good in your GUI or may not be the standard font used for GUIs on that system. Also, different versions of the same named font may have different size requirements for a given set of characters.

Using Standard Background Color

By default, uicontrols use the standard background color for the platform on which it is running (e.g., the standard shade of gray on the PC differs from that on UNIX). When your GUI is deployed on a different platform, it uses that computer's standard color. This provides a consistent look with respect to your GUI and other application GUIs.

If you change the `BackgroundColor` to another value, MATLAB always uses the specified color.

Cross-Platform Compatible Figure Units

Cross-platform compatible GUIs should look correct on computers having different screen sizes and resolutions. Since the size of a pixel can vary on different computer displays, using the default figure `Units` of `pixels` does not produce a GUI that looks the same on all platforms.

For this reason, GUIDE sets the figure `Units` property to `characters`.

System-Dependent Units

Figure character units are defined by characters from the default system font; one character unit equals the width of the letter x in the system font. The height of one character is the distance between the baselines of two lines of text (note that character units are not square).

GUIDE sets the figure `Units` property to `characters` so your GUIs automatically adjust the size and relative spacing of components as the GUI displays on different computers. For example, if the size of the text label on a component becomes larger because of different system font metrics, then the component size and the relative spacing between components increases proportionally.

Types of Callbacks

The primary mechanism for implementing a GUI is programming the callback of the uicontrol objects used to build the interface. However, in addition to the uicontrol `Callback` property, there are other properties that enable you to define callbacks.

Callback Properties for All Graphics Objects

All graphics objects have three properties that enable you to define callback routines:

- `ButtonDownFcn` — MATLAB executes this callback when users click the left mouse button and the cursor is over the object or within a five-pixel border around the object. See "Which Callback Executes" for information specific to uicontrols
- `CreateFcn` — MATLAB executes this callback when creating the object.
- `DeleteFcn` — MATLAB executes this callback just before deleting the object.

Callback Properties for Figures

Figures have additional properties that execute callback routines with the appropriate user action. Only the `CloseRequestFcn` has a callback defined by default:

- `CloseRequestFcn` — MATLAB executes the specified callback when a request is made to close the figure (by a `close` command, by the window manager menu, or by quitting MATLAB).
- `KeyPressFcn` — MATLAB executes the specified callback when users press a key and the cursor is within the figure window.
- `ResizeFcn` — MATLAB executes the specified callback routine when users resize the figure window.
- `WindowButtonDownFcn` — MATLAB executes the specified callback when users click the mouse button and the cursor is within the figure, but not over an enabled uicontrol.
- `WindowButtonMotionFcn` — MATLAB executes the specified callback when users move the mouse button within the figure window.

- WindowButtonUpFcn — MATLAB executes the specified callback when users release the mouse button, after having pressed the mouse button within the figure.

Which Callback Executes

Clicking on an enabled uicontrol prevents any ButtonDownFcn and WindowButtonDownFcn callbacks from executing. If you click on an inactive uicontrol, figure, or other graphics objects having callbacks defined, MATLAB first executes the WindownButtonDownFcn of the figure (if defined) and then the ButtonDownFcn of the object targeted by the mouse click.

Adding A Callback

To add a callback subfunction to the GUI M-file, click the right mouse button while the object is selected to display the Layout Editor context menu. Select the desired callback from the context menu and GUIDE adds the subfunction stub to the GUI M-file.

Interrupting Executing Callbacks

By default, MATLAB allows an executing callback to be interrupted by subsequently invoked callbacks. For example, suppose you have created a dialog box that displays a progress indicator while loading data. This dialog could have a “Cancel” button that stops the loading operation. The “Cancel” button’s callback routine would interrupt the currently executing callback routine.

There are cases where you may not want user actions to interrupt an executing callback. For example, a data analysis tool may need to perform lengthy calculations before updating a graph. An impatient user may inadvertently click the mouse on other components and thereby interrupt the calculations while in progress. This could change the MATLAB state before returning to the original callback.

The following sections provide more information on this topic:

- “Controlling Interruptibility”
- “The Event Queue”
- “Event Processing During Callback Execution”

Controlling Interruptibility

All graphics objects have an `Interruptible` property that determines whether their callbacks can be interrupted. The default value is `on`, which means that callbacks can be interrupted. However, MATLAB checks the event queue only when it encounters certain commands — `drawnow`, `figure`, `getframe`, `pause`, and, `waitfor`. Otherwise, the callback continues to completion.

The Event Queue

MATLAB commands that perform calculations or assign values to properties execute as they are encountered in the callback. However, commands or actions that affect the state of the figure window generate events that are placed in a queue. Events are caused by any command that causes the figure to be redrawn or any user action, such as a button click or cursor movement, for which there is a callback routine defined.

MATLAB processes the event queue only when the callback finishes execution or when the callback contains the following commands:

- `drawnow`
- `figure`
- `getframe`
- `pause`
- `waitfor`

When MATLAB encounters one of these commands in a callback, it suspends execution and processes the events in the event queue. The way MATLAB handles an event depends on the event type and the setting of the callback object's `Interruptible` property:

- Events that would cause another callback to execute (e.g., clicking a push button or figure window mouse button actions) can execute the callback only if the current callback object's `Interruptible` property is `on`.
- Events that cause the figure window to redraw execute the redraw regardless of the value of the current callback object's `Interruptible` property.

Note that callbacks defined for an object's `DeleteFcn` or `CreateFcn` or a figure's `CloseRequestFcn` or `ResizeFcn` interrupt an executing callback regardless of the value of the object's `Interruptible` property.

What Happens to Events That Are Blocked — BusyAction Property

All objects have a `BusyAction` property that determines what happens to their events when processed during noninterruptible callback routine execution.

`BusyAction` has two possible values:

- `queue` — Keep the event in the event queue and process it after the noninterruptible callback finishes.
- `cancel` — Discard the event and remove it from the event queue.

Event Processing During Callback Execution

The following sequence describes how MATLAB processes events while a callback executes:

1 If MATLAB encounters a `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` command in the callback routine, MATLAB suspends execution and begins processing the event queue.

2 If the event at the top of the queue calls for a figure window redraw, MATLAB performs the redraw and proceeds to the next event in the queue.

3 If the event at the top of the queue would cause a callback to execute, MATLAB determines whether the object whose callback is suspended is interruptible.

4 If the callback is interruptible, MATLAB executes the callback associated with the interrupting event. If that callback contains a `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` command, MATLAB repeats these steps for the remaining events in the queue.

5 If the callback is not interruptible, MATLAB checks the `BusyAction` property of the object that generated the event.

   a If `BusyAction` is `queue`, MATLAB leaves the event in the event queue.

   b If `BusyAction` is `cancel`, MATLAB discards the event.

6 When all events have been processed (either left in the queue, discarded, or handled as a redraw), MATLAB resumes execution of the interrupted callback routine.

This process continues until the callback completes execution. When MATLAB returns the prompt to the command window, all events have been processed.

Controlling Figure Window Behavior

When designing a GUI you need to consider how you want the figure window to behave. The appropriate behavior for a particular GUI depends on intended use. Consider the following examples:

- A GUI that implements tools for annotating graphs is usually designed to be available while the user performs other MATLAB tasks. Perhaps this tool operates on only one figure at a time so you need a new instance of this tool for each graph.
- A dialog that requires an answer to a question may need to block MATLAB execution until the user answers the question. However, the user may need to look at other MATLAB windows to obtain information needed to answer the question. The dialog is blocking.
- A dialog that warns users that the specified operation will delete files so you want to force the user to respond to the warning before performing any other action. In this case, the dialog is both blocking and modal.

The following techniques are useful for handling these GUI design issues:

- Allow single or multiple instances of the GUI at any one time.
- Use modal figure windows that allow users to interact only with the GUI.

Using Modal Figure Windows

Modal windows trap all keyboard and mouse events that occur in any visible MATLAB window. This means a modal GUI figure can process the user interactions with any of its components, but does not allow the user to access any other MATLAB window (including the command window). In addition, a modal window remains stacked on top of other MATLAB windows until it is deleted, at which time focus returns to the window that last had focus. See the figure `WindowStyle` property for more details.

Use modal figures when you want to force users to respond to your GUI before allowing them to take other actions in MATLAB.

Making a Figure Modal

Set the figure's `WindowStyle` property to `modal` to make the window modal. You can use the Property Inspector to change this property or add a statement in the initialization section of the GUI M-file in the opening function with the `set` command.

```
set(hObject,'WindowStyle','modal')
```

Dismissing a Modal Figure

A GUI using a modal figure must take one of the following actions in a callback routine to release control:

- Delete the figure.
  ```
  delete(handles.figure1)
  ```
- Make the figure invisible.
  ```
  set(handles.figure1,'Visible','off')
  ```
- Change the figure's `WindowStyle` property to `normal`.
  ```
  set(handles.figure1,'WindowStyle','normal')
  ```

The user can also type **Ctrl+C** in a modal figure to convert it to a normal window.

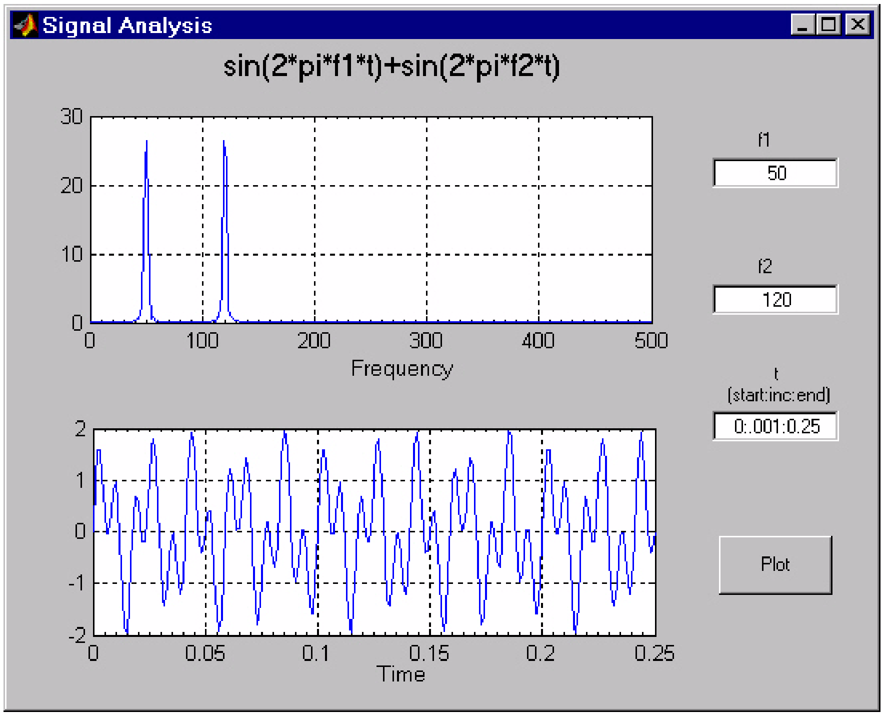
The next section, "Application Examples" on page 6-1, provides more examples of programming GUIs.

Application Examples

This chapter contains a series of examples that illustrate techniques that are useful for creating GUIs. Each example provides a link to the actual GUI in the GUIDE Layout Editor and a link to the GUI M-file displayed in the MATLAB editor. If you are reading this in the MATLAB Help browser, you can use these links to view the GUI and the M-file.

GUI with Multiple Axes

This example creates a GUI that contains two axes for plotting data. For simplicity, this example obtains data by evaluating an expression using parameters entered by the user.



f1: 50

f2: 120

t (start:inc:end): 0:.001:0.25

Techniques Used in the Example

GUI-building techniques illustrated in this example include

- Controlling which axes is the target for plotting commands.
- Using edit text controls to read numeric input and MATLAB expressions.

View the Layout and GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the MATLAB Editor.

Design of the GUI

This GUI requires three input values:

- Frequency one (`f1`)
- Frequency two (`f1`)
- A time vector (`t`)

When the user clicks the **Plot** button, the GUI puts these values into a MATLAB expression that is the sum of two sine function:

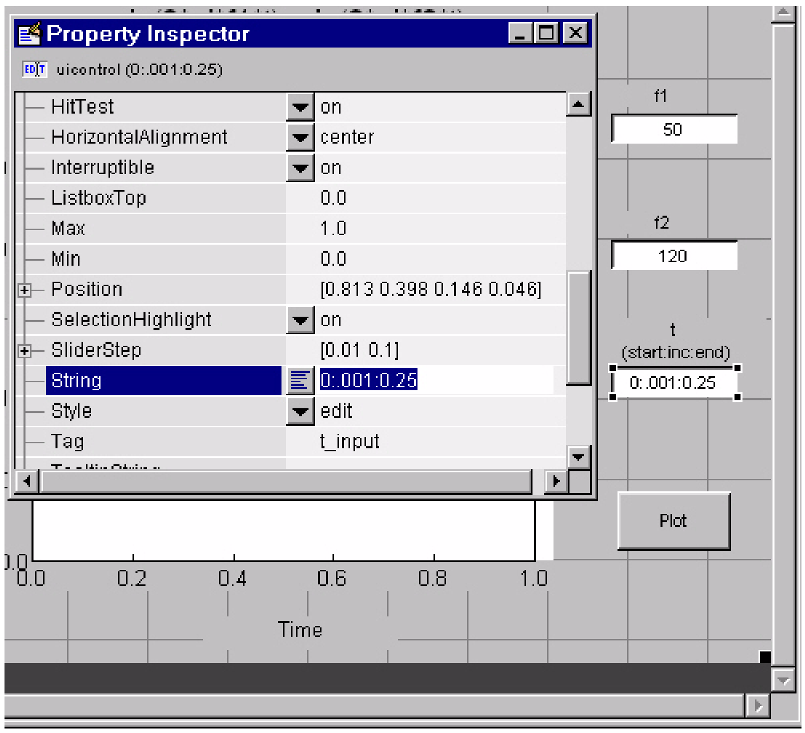
```
x = sin(2*pi*f1*t) + sin(2*pi*f2*t)
```

The GUI then calculates the FFT of `x` and creates two plots — one frequency domain and one time domain.

Specifying Default Values for the Inputs

The GUI uses default values for the three inputs. This enables users to click on the **Plot** button and see a result as soon as the GUI is run. It also helps to indicate what values the user might enter.

To create the default values, set the `String` property of the edit text. The following figure shows the value set for the time vector.

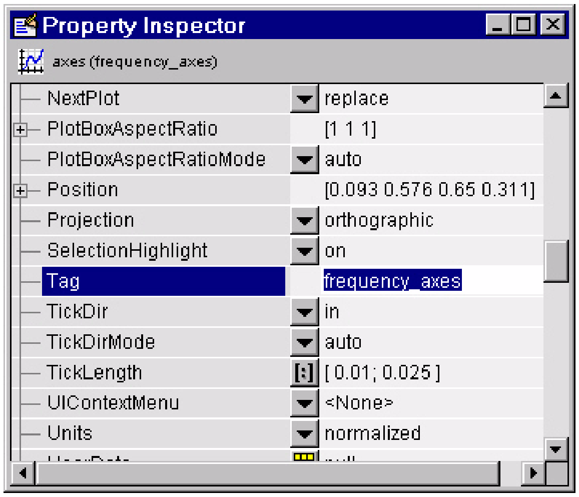


HitTest	on
HorizontalAlignment	center
Interruptible	on
ListboxTop	0.0
Max	1.0
Min	0.0
Position	[0.813 0.398 0.146 0.046]
SelectionHighlight	on
SliderStep	[0.01 0.1]
String	0:.001:0.25
Style	edit
Tag	t_input

Identifying the Axes

Since there are two axes in this GUI, you must be able to specify which one you want to target when you issue the plotting commands. To do this, use the `handles` structure, which contains the handles of all components in the GUI.

The field name in the `handles` structure that contains the handle of any given component is derived from the component's `Tag` property. To make code more readable (and to make it easier to remember) this examples sets the `Tag` to descriptive names.



axes (frequency_axes)

NextPlot	replace
PlotBoxAspectRatio	[1 1 1]
PlotBoxAspectRatioMode	auto
Position	[0.093 0.576 0.65 0.311]
Projection	orthographic
SelectionHighlight	on
Tag	frequency_axes
TickDir	in
TickDirMode	auto
TickLength	[0.01; 0.025]
UIContextMenu	<None>
Units	normalized

For example, the `Tag` of the axes used to display the FFT is set to `frequency_axes`. Therefore, within a callback, you access its handle with

```
handles.frequency_axes
```

Likewise, the `Tag` of the time axes is set to `time_axes`.

See “Managing GUI Data with the Handles Structure” on page 5-8 for more information on the `handles` structure. See “Plot Push Button Callback” on page 6-6 for the details of how to use the handle to specify the target axes.

GUI Option Settings

There are two GUI option settings that are particularly important for this GUI:

- Resize behavior: **Proportional**
- Command-line accessibility: **Callback**

Proportional Resize Behavior. Selecting **Proportional** as the resize behavior enables users to change the GUI to better view the plots. The components change size in proportion to the GUI figure size. This generally produces good results except when extremes of dimensions are used.

Callback Accessibility of Object Handles. When GUIs include axes, handles should be visible from within callbacks. This enables you to use plotting commands like you would on the command line. Note that **Callback** is the default setting for command-line accessibility.

See “Selecting GUI Options” on page 3-21 for more information.

Plot Push Button Callback

This GUI uses only the **Plot** button callback; the edit text callbacks are not needed and have been deleted from the GUI M-file. When a user clicks the **Plot** button, the callback performs three basic tasks — it gets user input from the edit text components, calculates data, and creates the two plots.

Getting User Input

The three edit text boxes provide a way for the user to enter values for the two frequencies and the time vector. The first task for the callback is to read these values. This involves:

- Reading the current values in the three edit text boxes using the `handles` structure to access the edit text handles.
- Converting the two frequency values (`f1` and `f2`) from string to doubles using `str2double`.
- Evaluating the time string using `eval` to produce a vector `t`, which the callback used to evaluate the mathematical expression.

The following code shows how the callback obtains the input.

```
% Get user input from GUI
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));
```

Calculating Data

Once the input data has been converted to numeric form and assigned to local variables, the next step is to calculate the data needed for the plots. See the `fft` function for an explanation of how this is done.

Targeting Specific Axes

The final task for the callback is to actually generate the plots. This involves

- Making the appropriate axes current using the `axes` command and the handle of the axes. For example,

  ```
  axes(handles.frequency_axes)
  ```

- Issuing the `plot` command.
- Setting any properties that are automatically reset by the `plot` command.

The last step is necessary because many plotting commands (including `plot`) clear the axes before creating the graph. This means you cannot use the Property Inspector to set the `XMinorTick` and grid properties that are used in this example, since they are reset when the callback executes `plot`.

When looking at the following code listing, note how the `handles` structure is used to access the handle of the axes when needed.

Plot Button Code Listing

```
function plot_button_Callback(hObject, eventdata, handles)
% hObject    handle to plot_button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get user input from GUI
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));

% Calculate data
x = sin(2*pi*f1*t) + sin(2*pi*f2*t);
y = fft(x,512);
m = y.*conj(y)/512;
f = 1000*(0:256)/512;;

% Create frequency plot
axes(handles.frequency_axes) % Select the proper axes
plot(f,m(1:257))
set(handles.frequency_axes,'XMinorTick','on')
grid on

% Create time plot
axes(handles.time_axes) % Select the proper axes
```

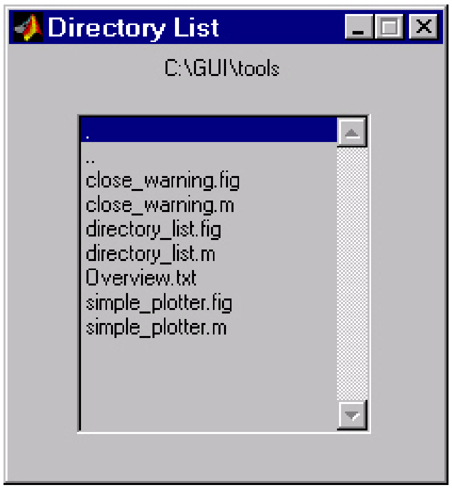
```
plot(t,x)
set(handles.time_axes,'XMinorTick','on')
grid on
```

List Box Directory Reader

This example uses a list box to display the files in a directory. When the user double clicks on a list item, one of the following happens:

- If the item is a file, the GUI opens the file appropriately for the file type.
- If the item is a directory, the GUI reads the contents of that directory into the list box.
- If the item is a single dot (.), the GUI updates the display of the current directory.
- If the item is two dots (..), the GUI changes to the directory up one level and populates the list box with the contents of that directory.

The following figure illustrates the GUI.



View the Layout and GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

Implementing the GUI

The following sections describe the implementation:

- "Specifying the Directory to List" — shows how to pass a directory path as input argument when the GUI is run.
- "Loading the List Box" — describes the subfunction that loads the contents of the directory into the list box. This subfunction also saves information about the contents of a directory in the `handles` structure.
- "The List Box Callback" — explains how the list box is programmed to respond to user double clicks on items in the list box.

Specifying the Directory to List

You can specify the directory to list when the GUI is first opened by passing the string `'create'` and a string containing the full path to the directory as arguments. The syntax for doing this is `list_box('create','dir_path')`. If you do not specify a directory (i.e., if you call the GUI M-file with no input arguments), the GUI then uses the MATLAB current directory.

The default behavior of the GUI M-file that GUIDE generates is to run the GUI when there are no input arguments or to call a subfunction when the first input argument is a character string. This example changes this behavior so that you can call the M-file with

- No input arguments — run the GUI using the MATLAB current directory.
- First input argument is `'create'` and second input argument is a string that specifies a valid path to a directory — run the GUI, displaying the specified directory.

- First input argument is not a directory, but is a character string and there is more than one argument — execute the subfunction identified by the argument (execute callback).

The following code listing show the setup section of the GUI M-file, which does one the following:

- Sets the list box directory to the current directory, if no directory is specified.
- Changes the current directory, if a directory is specified.

```
function lbox2_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to untitled (see VARARGIN)

% Choose default command line output for lbox2
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

if nargin == 3,
    initial_dir = pwd;
elseif nargin == 4 & exist(varargin{1},'dir')
    initial_dir = varargin{1};
else
    errordlg('Input argument must be a valid directory','Input
Argument Error!')
    return
end
% Populate the listbox
load_listbox(initial_dir,handles)
```

Loading the List Box

This example creates a subfunction to load items into the list box. This subfunction accepts the path to a directory and the `handles` structure as input arguments. It performs these steps:

- Change to the specified directory so the GUI can navigate up and down the tree as required.
- Use the `dir` command to get a list of files in the specified directory and to determine which name is a directory and which is a file. `dir` returns a structure (`dir_struct`) with two fields, `name` and `isdir`, which contain this information.
- Sort the file and directory names (`sortrows`) and save the sorted names and other information in the `handles` structure so this information can be passed to other functions.

  The `name` structure field is passed to `sortrows` as a cell array, which is transposed to get one file name per row. The `isdir` field and the sorted index values, `sorted_index`, are saved as vectors in the `handles` structure.
- Call `guidata` to save the `handles` structure.
- Set the list box `String` property to display the file and directory names and set the `Value` property to `1`. This is necessary to ensure `Value` never exceeds the number of items in `String`, since MATLAB updates the `Value` property only when a selection occurs and not when the contents of `String` changes.
- Displays the current directory in the text box by setting its `String` property to the output of the `pwd` command.

The `load_listbox` function is called by the opening function of the GUI M-file as well as by the list box callback.

```
function load_listbox(dir_path, handles)
cd (dir_path)
dir_struct = dir(dir_path);
[sorted_names,sorted_index] = sortrows({dir_struct.name}');
handles.file_names = sorted_names;
handles.is_dir = [dir_struct.isdir];
handles.sorted_index = [sorted_index];
guidata(handles.figure1,handles)
set(handles.listbox1,'String',handles.file_names,...
   'Value',1)
set(handles.text1,'String',pwd)
```

The List Box Callback

The list box callback handles only one case: a double-click on an item. Double clicking is the standard way to open a file from a list box. If the selected item is a file, it is passed to the `open` command; if it is a directory, the GUI changes to that directory and lists its contents.

Defining How to Open File Types

The callback makes use of the fact that the `open` command can handle a number of different file types. However, the callback treats FIG-files differently. Instead of opening the FIG-file, it passes it to the `guide` command for editing.

Determining Which Item the User Selected

Since a single click on an item also invokes the list box callback, it is necessary to query the figure `SelectionType` property to determine when the user has performed a double click. A double-click on an item sets the `SelectionType` property to `open`.

All the items in the list box are referenced by an index from `1` to `n`, where `1` refers to the first item and `n` is the index of the `n`th item. MATLAB saves this index in the list box `Value` property.

The callback uses this index to get the name of the selected item from the list of items contained in the `String` property.

Determining if the Selected Item is a File or Directory

The `load_listbox` function uses the `dir` command to obtain a list of values that indicate whether an item is a file or directory. These values (1 for directory, 0 for file) are saved in the `handles` structure. The list box callback queries these values to determine if current selection is a file or directory and takes the following action:

- If the selection is a directory — change to the directory (`cd`) and call `load_listbox` again to populate the list box with the contents of the new directory.
- If the selection is a file — get the file extension (`fileparts`) to determine if it is a FIG-file, which is opened with `guide`. All other file types are passed to `open`.

The `open` statement is called within a `try/catch` block to capture errors in an error dialog (`errordlg`), instead of returning to the command line.

```
function listbox1_Callback(hObject, eventdata, handles)
if strcmp(get(handles.figure1,'SelectionType'),'open') % If double click
```

```
    index_selected = get(handles.listbox1,'Value');
    file_list = get(handles.listbox1,'String');
    filename = file_list{index_selected}; % Item selected in list box
    if  handles.is_dir(handles.sorted_index(index_selected)) % If directory
        cd (filename)
        load_listbox(pwd,handles) % Load list box with new directory
    else
        [path,name,ext,ver] = fileparts(filename);
        switch ext
        case '.fig'
            guide (filename) % Open FIG-file with guide command
        otherwise
            try
                     open(filename) % Use open for other file types
            catch
                     errordlg(lasterr,'File Type Error','modal')
            end
        end
    end
end
```

Opening Unknown File Types

You can extend the file types that the `open` command recognizes to include any file having a three-character extension. You do this by creating an M-file with the name `openxyz`, where `xyz` is the extension. Note that the list box callback does not take this approach for FIG-files since `openfig.m` is required by the GUI M-file. See `open` for more information.

Accessing Workspace Variables from a List Box

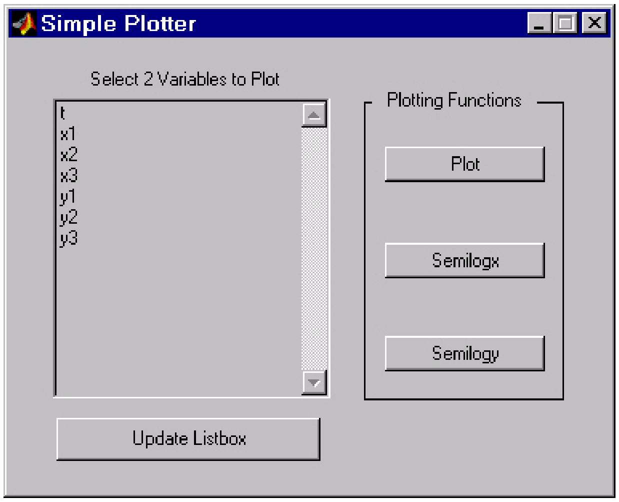
This GUI uses a list box to display workspace variables, which the user can then plot.

Techniques Used in This Example

This example demonstrates how to:

- Populate the list box with the variable names that exist in the base workspace.
- Display the list box with no items initially selected.
- Enable multiple item selection in the list box.
- Update the list items when the user press a button.
- Evaluate the plotting commands in the base workspace.

The following figure illustrates the layout.



Note that the list box callback is not used in this program because the plotting actions are initiated by push buttons. In this situation you must do one of the following:

- Leave the empty list box callback in the GUI M-file.

- Delete the string assigned to the list box `Callback` property.

View the Layout and GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

Reading Workspace Variables

When the GUI initializes, it needs to query the workspace variables and set the list box `String` property to display these variable names. Adding the following subfunction to the GUI M-file accomplishes this using `evalin` to execute the `who` command in the base workspace. The `who` command returns a cell array of strings, which are used to populate the list box.

```
function update_listbox(handles)
vars = evalin('base','who');
set(handles.listbox1,'String',vars)
```

The function's input argument is the `handles` structure generated by the GUI M-file. This structure contains the handle of the list box, as well as the handles all other components in the GUI.

The callback for the **Update Listbox** push button also calls `update_listbox`.

Reading the Selections from the List Box

This GUI requires the user to select two variables from the workspace and then choose one of three plot commands to create the graph: `plot`, `semilogx`, or `semilogy`.

Enabling Multiple Selection

To enable multiple selection in a list box, you must set the `Min` and `Max` properties so that `Max    Min > 1`. This requires you to change the default `Min` and `Max` values of `0` and `1` to meet these conditions. Use the Property Inspector to set these properties on the list box.

How Users Select Multiple Items

List box multiple selection follows the standard for most systems:

- **Control**-click left mouse button — noncontiguous multi-item selection
- **Shift**-click left mouse button — contiguous multi-item selection

Users must use one of these techniques to select the two variables required to create the plot.

Returning Variable Names for the Plotting Functions

The `get_var_names` subroutine returns the two variable names that are selected when the user clicks on one of the three plotting buttons. The function

- Gets the list of all items in the list box from the `String` property.
- Gets the indices of the selected items from the `Value` property.
- Returns two string variables, if there are two items selected. Otherwise `get_var_names` displays an error dialog explaining that the user must select two variables.

Here is the code for `get_var_names`:

```
function [var1,var2] = get_var_names(handles)
list_entries = get(handles.listbox1,'String');
index_selected = get(handles.listbox1,'Value');
if length(index_selected) ~= 2
   errordlg('You must select two variables',...
          'Incorrect Selection','modal')
else
```

```
    var1 = list_entries{index_selected(1)};
    var2 = list_entries{index_selected(2)};
end
```

Callbacks for the Plotting Buttons

The callbacks for the plotting buttons call `get_var_names` to get the names of the variables to plot and then call `evalin` to execute the plot commands in the base workspace.

For example, here is the callback for the `plot` function:

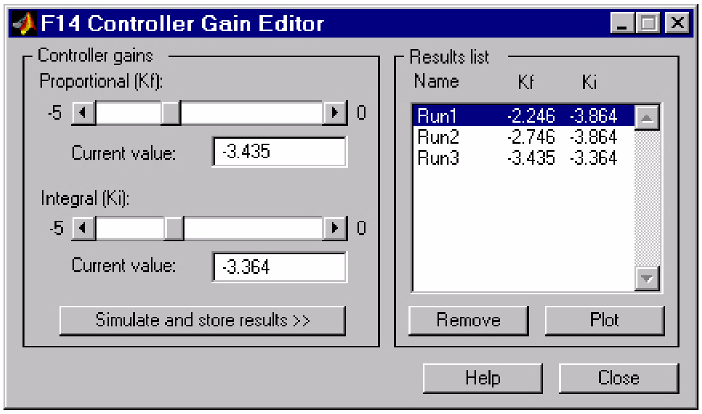
```
function plot_button_Callback(hObject, eventdata, handles)
[x,y] = get_var_names(handles);
evalin('base',['plot(' x ',' y ')'])
```

The command to evaluate is created by concatenating the strings and variables that result in the command:

```
plot(x,y)
```

A GUI to Set Simulink Model Parameters

This example illustrates how to create a GUI that sets the parameters of a Simulink model. In addition, the GUI can run the simulation and plot the results. The following picture shows the GUI after running three simulations with different values for controller gains.



Proportional (Kf) current value: -3.435

Integral (Ki) current value: -3.364

Name	Kf	Ki
Run1	-2.246	-3.864
Run2	-2.746	-3.864
Run3	-3.435	-3.364

Techniques Used in This Example

This example illustrates a number of GUI building techniques:

- Opening and setting parameters on a Simulink model from a GUI.
- Implementing sliders that operate in conjunction with text boxes, which display the current value as well as accepting user input.
- Enabling and disabling controls, depending on the state of the GUI.
- Managing a variety of shared data using the `handles` structure.
- Directing graphics output to figures with hidden handles.
- Adding a help button that displays `.html` files in the MATLAB Help browser.

View the Layout and GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

How to Use the GUI (Text of GUI Help)

You can use the F14 Controller Gain Editor to analyze how changing the gains used in the Proportional-Integral Controller affect the aircraft's angle of attack and the amount of G force the pilot feels.

Note that the Simulink diagram `f14.mdl` must be open to run this GUI. If you close the F14 Simulink model, the GUI reopens it whenever it requires the model to execute.

Changing the Controller Gains

You can change gains in two blocks:

- The Proportional gain (Kf) in the Gain block
- The Integral gain (Ki) in the Transfer Function block

You can change either of the gains in one of the two ways:

- Move the slider associated with that gain.
- Type a new value into the **Current value** edit field associated with that gain.

The block’s values are updated as soon as you enter the new value in the GUI.

Running the Simulation

Once you have set the gain values, you can run the simulation by clicking the **Simulate and store results** button. The simulation time and output vectors are stored in the **Results list**.

Plotting the Results

You can generate a plot of one or more simulation results by selecting the row of results (`Run1`, `Run2`, etc.) in the **Results list** that you want to plot and clicking the **Plot** button. If you select multiple rows, the graph contains a plot of each result.

The graph is displayed in a figure, which is cleared each time you click the **Plot** button. The figure's handle is hidden so that only the GUI can display graphs in this window.

Removing Results

To remove a result from the **Results list**, select the row or rows you want to remove and click the **Remove** button.

Running the GUI

The GUI is nonblocking and nonmodal since it is designed to be used as an analysis tool.

GUI Options Settings

This GUI uses the following GUI option settings:

- Resize behavior: **Non-resizable**
- Command-line accessibility: **Off**
- M-file options selected:
  - Generate callback function prototypes
  - GUI allows only one instance to run

Opening the Simulink Block Diagrams

This example is designed to work with the F14 Simulink model. Since the GUI sets parameters and runs the simulation, the F14 model must be open when the GUI is displayed. When the GUI M-file runs the GUI, it executes the `model_open` subfunction. The purpose of the subfunction is to

- Determine if the model is open (`find_system`).
- Open the block diagram for the model and the subsystem where the parameters are being set, if not open already (`open_system`).
- Change the size of the controller Gain block so it can display the gain value (`set_param`).
- Bring the GUI forward so it is displayed on top of the Simulink diagrams (`figure`).
- Set the block parameters to match the current settings in the GUI.

Here is the code for the `model_open` subfunction.

```
function model_open(handles)
if  isempty(find_system('Name','f14')),
    open_system('f14'); open_system('f14/Controller')
    set_param('f14/Controller/Gain','Position',[275 14 340 56])
    figure(handles.F14ControllerEditor)
    set_param('f14/Controller Gain','Gain',...
        get(handles.KfCurrentValue,'String'))
    set_param('f14/Controller/Proportional plus integral compensator',...
        'Numerator',...
        get(handles.KiCurrentValue,'String'))
end
```

Programming the Slider and Edit Text Components

This GUI employs a useful combination of components in its design. Each slider is coupled to an edit text component so that:

- The edit text displays the current value of the slider.
- The user can enter a value into the edit text box and cause the slider to update to that value.
- Both components update the appropriate model parameters when activated by the user.

Slider Callback

The GUI uses two sliders to specify block gains since these components enable the selection of continuous values within a specified range. When a user changes the slider value, the callback executes the following steps:

- Calls `model_open` to ensure that the Simulink model is open so that simulation parameters can be set.
- Gets the new slider value.
- Sets the value of the **Current value** edit text component to match the slider.
- Sets the appropriate block parameter to the new value (`set_param`).

Here is the callback for the **Proportional (Kf)** slider.

```
function KfValueSlider_Callback(hObject, eventdata, handles)
% Ensure model is open
model_open(handles)
% Get the new value for the Kf Gain from the slider
NewVal = get(hObject, 'Value');
% Set the value of the KfCurrentValue to the new value set by
slider
set(handles.KfCurrentValue,'String',NewVal)
% Set the Gain parameter of the Kf Gain Block to the new value
set_param('f14/Controller/Gain','Gain',num2str(NewVal))
```

Note that, while a slider returns a number and the edit text requires a string, uicontrols automatically convert the values to the correct type.

The callback for the **Integral (Ki)** slider follows a similar approach.

Current Value Edit Text Callback

The edit text box enables users to type in a value for the respective parameter. When the user clicks on another component in the GUI after typing into the text box, the edit text callback executes the following steps:

- Calls `model_open` to ensure that the Simulink model is open so that it can set simulation parameters.
- Converts the string returned by the edit box `String` property to a double (`str2double`).
- Checks whether the value entered by the user is within the range of the slider:

  If the value is out of range, the edit text `String` property is set to the value of the slider (rejecting the number typed in by the user).

  If the value is in range, the slider `Value` property is updated to the new value.

- Sets the appropriate block parameter to the new value (set_param).

Here is the callback for the Kf **Current value** text box.

```
function KfCurrentValue_Callback(hObject, eventdata, handles)
% Ensure model is open
model_open(handles)
% Get the new value for the Kf Gain
NewStrVal = get(hObject, 'String');
NewVal = str2double(NewStrVal);
% Check that the entered value falls within the allowable range
if  isempty(NewVal) | (NewVal< -5) | (NewVal>0),
    % Revert to last value, as indicated by KfValueSlider
    OldVal = get(handles.KfValueSlider,'Value');
    set(hObject, 'String',OldVal)
else, % Use new Kf value
    % Set the value of the KfValueSlider to the new value
    set(handles.KfValueSlider,'Value',NewVal)
    % Set the Gain parameter of the Kf Gain Block to the new value
    set_param('f14/Controller/Gain','Gain',NewStrVal)
end
```

The callback for the Ki **Current value** follows a similar approach.

Running the Simulation from the GUI

The GUI **Simulate and store results** button callback runs the model simulation and stores the results in the handles structure. Storing data in the handles structure simplifies the process of passing data to other subfunction since this structure can be passed as an argument.

When a user clicks on the **Simulate and store results** button, the callback executes the following steps:

- Calls sim, which runs the simulation and returns the data that is used for plotting.
- Creates a structure to save the results of the simulation, the current values of the simulation parameters set by the GUI, and the run name and number.
- Stores the structure in the handles structure.
- Updates the list box String to list the most recent run.

Here is the **Simulate and store results** button callback.

```
function SimulateButton_Callback(hObject, eventdata, handles)
[timeVector,stateVector,outputVector] = sim('f14');
% Retrieve old results data structure
if isfield(handles,'ResultsData') &
~isempty(handles.ResultsData)
   ResultsData = handles.ResultsData;
   % Determine the maximum run number currently used.
   maxNum = ResultsData(length(ResultsData)).RunNumber;
   ResultNum = maxNum+1;
else
   % Set up the results data structure
   ResultsData = struct('RunName',[],'RunNumber',[],...

'KiValue',[],'KfValue',[],'timeVector',[],'outputVector',[]);
   ResultNum = 1;
end
if isequal(ResultNum,1),
   % Enable the Plot and Remove buttons
   set([handles.RemoveButton,handles.PlotButton],'Enable','on')
end
% Get Ki and Kf values to store with the data and put in the
results list.
Ki = get(handles.KiValueSlider,'Value');
Kf = get(handles.KfValueSlider,'Value');
ResultsData(ResultNum).RunName = ['Run',num2str(ResultNum)];
ResultsData(ResultNum).RunNumber = ResultNum;
ResultsData(ResultNum).KiValue = Ki;
ResultsData(ResultNum).KfValue = Kf;
ResultsData(ResultNum).timeVector = timeVector;
ResultsData(ResultNum).outputVector = outputVector;
% Build the new results list string for the listbox
ResultsStr = get(handles.ResultsList,'String');
if isequal(ResultNum,1)
   ResultsStr = {['Run1',num2str(Kf),' ',num2str(Ki)]};
else
   ResultsStr = [ResultsStr;...
   {['Run',num2str(ResultNum),' ',num2str(Kf),'
',num2str(Ki)]}];
```

```
end
set(handles.ResultsList,'String',ResultsStr);
% Store the new ResultsData
handles.ResultsData = ResultsData;
guidata(hObject, handles)
```

Removing Results from the List Box

The GUI **Remove** button callback deletes any selected item from the **Results list** list box. It also deletes the corresponding run data from the `handles` structure. When a user clicks on the **Remove** button, the callback executes the following steps:

- Determines which list box items are selected when a user clicks on the **Remove** button and removes these items from the list box `String` property by setting each item to the empty matrix `[]`.
- Removes the deleted data from the `handles` structure.
- Displays the string `<empty>` and disables the **Remove** and **Plot** buttons (using the `Enable` property), if all the items in the list box are removed.
- Save the changes to the `handles` structure (`guidata`).

Here is the **Remove** button callback.

```
function RemoveButton_Callback(hObject, eventdata, handles)
currentVal = get(handles.ResultsList,'Value');
resultsStr = get(handles.ResultsList,'String');
numResults = size(resultsStr,1);
% Remove the data and list entry for the selected value
resultsStr(currentVal) =[];
handles.ResultsData(currentVal)=[];
% If there are no other entries, disable the Remove and Plot button
% and change the list sting to <empty>
if isequal(numResults,length(currentVal)),
   resultsStr = {'<empty>'};
   currentVal = 1;

set([handles.RemoveButton,handles.PlotButton],'Enable','off')
end
% Ensure that list box Value is valid, then reset Value and String
currentVal = min(currentVal,size(resultsStr,1));
```

```
set(handles.ResultsList,'Value',currentVal,'String',resultsStr)
% Store the new ResultsData
guidata(hObject, handles)
```

Plotting the Results Data

The GUI **Plot** button callback creates a plot of the run data and adds a legend. The data to plot is passed to the callback in the `handles` structure, which also contains the gain settings used when the simulation ran. When a user clicks on the **Plot** button, the callback executes the following steps:

- Collects the data for each run selected in the **Results list**, including two variables (time vector and output vector) and a color for each result run to plot.
- Generates a string for the legend from the stored data.
- Creates the figure and axes for plotting and saves the handles for use by the **Close** button callback.
- Plots the data, adds a legend, and makes the figure visible.

Plotting Into the Hidden Figure

The figure that contains the plot is created invisible and then made visible after adding the plot and legend. To prevent this figure from becoming the target for plotting commands issued at the command line or by other GUIs, its `HandleVisibility` and `IntegerHandle` properties are set to `off`. However, this means the figure is also hidden from the `plot` and `legend` commands.

Use the following steps to plot into a hidden figure:

- Save the handle of the figure when you create it.
- Create an axes, set its `Parent` property to the figure handle, and save the axes handle.
- Create the plot (which is one or more line objects), save these line handles, and set their `Parent` properties to the handle of the axes.
- Make the figure visible.

Plot Button Callback Listing

Here is the **Plot** button callback.

```
function PlotButton_Callback(hObject, eventdata, handles)
```

```
currentVal = get(handles.ResultsList,'Value');
% Get data to plot and generate command string with color
% specified
legendStr = cell(length(currentVal),1);
plotColor = {'b','g','r','c','m','y','k'};
for ctVal = 1:length(currentVal);
   PlotData{(ctVal*3)-2} =
handles.ResultsData(currentVal(ctVal)).timeVector;
   PlotData{(ctVal*3)-1} =
handles.ResultsData(currentVal(ctVal)).outputVector;
   numColor = ctVal - 7*( floor((ctVal-1)/7) );
   PlotData{ctVal*3} = plotColor{numColor};
   legendStr{ctVal} =
[handles.ResultsData(currentVal(ctVal)).RunName,...
       '; Kf=', ...

num2str(handles.ResultsData(currentVal(ctVal)).KfValue),...
       ';  Ki=', ...

num2str(handles.ResultsData(currentVal(ctVal)).KiValue)];
end
% If necessary, create the plot figure and store in handles
% structure
if ~isfield(handles,'PlotFigure') |
~ishandle(handles.PlotFigure),
   handles.PlotFigure = figure('Name','F14 Simulation
Output',...
       'Visible','off','NumberTitle','off',...
       'HandleVisibility','off','IntegerHandle','off');
   handles.PlotAxes = axes('Parent',handles.PlotFigure);
   guidata(hObject, handles)
end
% Plot data
pHandles = plot(PlotData{:},'Parent',handles.PlotAxes);
% Add a legend, and bring figure to the front
legend(pHandles(1:2:end),legendStr{:})
% Make the figure visible and bring it forward
figure(handles.PlotFigure)
```

The GUI Help Button

The GUI **Help** button callback displays an HTML file in the MATLAB Help browser. It uses two commands:

- The `which` command returns the full path to the file when it is on the MATLAB path
- The `web` command displays the file in the Help browser.

This is the **Help** button callback.

```
function HelpButton_Callback(hObject, eventdata, handles)
HelpPath = which('f14ex_help.html');
web(HelpPath);
```

You can also display the help document in a Web browser or load an external URL. See the Web documentation for a description of these options.

Closing the GUI

The GUI **Close** button callback closes the plot figure, if one exists and then closes the GUI. The handle of the plot figure and the GUI figure are available from the `handles` structure. The callback executes two steps:

- Checks to see if there is a `PlotFigure` field in the `handles` structure and if it contains a valid figure handle (the user could have closed the figure manually).
- Closes the GUI figure

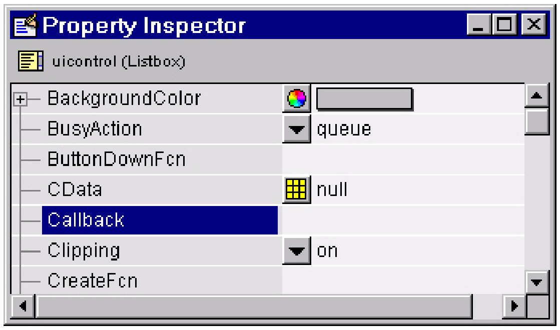
This is the **Close** button callback.

```
function CloseButton_Callback(hObject, eventdata, handles)
% Close the GUI and any plot window that is open
if isfield(handles,'PlotFigure') & ishandle(handles.PlotFigure),
   close(handles.PlotFigure);
end
close(handles.F14ControllerEditor);
```

The List Box Callback and Create Function

This GUI does not use the list box callback since the actions performed on list box items are carried out by push buttons (**Simulate and store results**, **Remove**, and **Plot**). However, GUIDE automatically inserts a callback stub when you add the list box and automatically sets the `Callback` property to execute this subfunction whenever the callback is triggered (which happens when users select an item in the list box).

In this case, there is no need for the list box callback to execute, so you should delete it from the GUI M-file. It is important to remember to also delete the `Callback` property string so MATLAB does not attempt to execute the callback. You can do this using the property inspector:



```
BackgroundColor
BusyAction        queue
ButtonDownFcn
CData             null
Callback
Clipping          on
CreateFcn
```

See the description of list boxes for more information on how to trigger the list box callback.

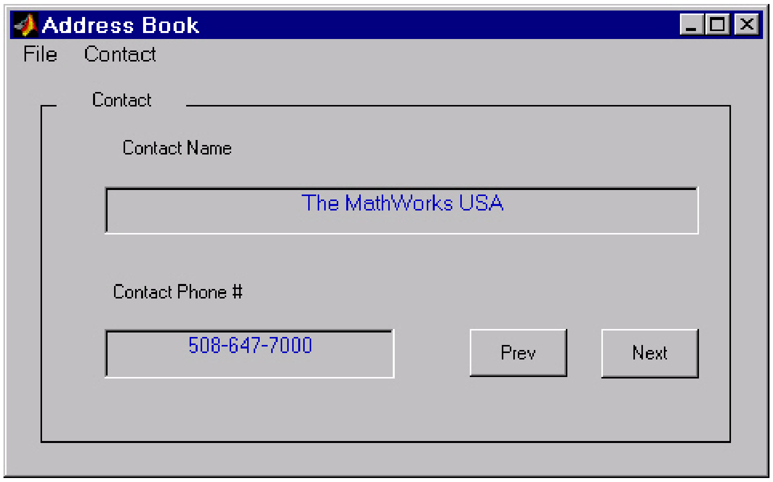
Setting the Background to White

The list box create function enables you to determine the background color of the list box. The following code shows the create function for the list box that is tagged `ResultsList`.

```
function ResultsList_CreateFcn(hObject, eventdata, handles)
% Hint: listbox controls usually have a white background, change
%       'usewhitebg' to 0 to use default.  See ISPC and COMPUTER.
usewhitebg = 1;
if usewhitebg
    set(hObject,'BackgroundColor','white');
else
set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
```

An Address Book Reader

This example shows how to implement a GUI that displays names and phone numbers, which it reads from a MAT-file.



Techniques Used in This Example

This example demonstrates the following GUI programming techniques:

- Uses open and save dialogs to provide a means for users to locate and open the address book MAT-files and to save revised or new address book MAT-files.
- Defines callbacks written for GUI menus.
- Uses the GUI's `handles` structure to save and recall shared data.
- Uses a GUI figure resize function.

Managing Shared Data

One of the key techniques illustrated in this example is how to keep track of information and make it available to the various subfunctions. This information includes

- The name of the current MAT-file

- The names and phone numbers stored in the MAT-file
- An index pointer that indicates the current name and phone number, which must be updated as the user pages through the address book
- The figure position and size
- The handles of all GUI components

The descriptions of the subfunctions that follow illustrate how to save and retrieve information from the `handles` structure. See "Sharing Data with the Handles Structure" on page 5-2 for background information on this structure.

View the Layout and GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the MATLAB Editor.

Running the GUI

The GUI is nonblocking and nonmodal since it is designed to be displayed while you perform other MATLAB tasks.

GUI Option Settings

This GUI uses the following GUI option settings:

- Resize behavior: **User-specified**
- Command-line accessibility: **Off**

- GUI M-file options selected:

  Generate callback function prototypes

  Application allows only one instance to run

Calling the GUI

You can call the GUI M-file with no arguments, in which case the GUI uses the default address book MAT-file, or you can specify an alternate MAT-file from which the GUI reads information. In this example, the user calls the GUI with a pair of arguments, `address_book('book', 'my_list.mat')`. The first argument, `'book'`, is a key word that the M-file looks for in the opening function. If the M-file finds the key word, it knows tho use the second argument as the MAT-file for the address book. Calling the GUI with this syntax is analogous to calling it with a valid property-value pair, such as `('color', 'red')`. However, since `'book'` is not a valid figure property, in this example the opening function in the M-file includes code to recognize the pair `('book', 'my_list.mat')`.

Note that it is not necessary to use the key word `'book'`. You could program the M-file to accept just the MAT-file as an argument, using the syntax `address_book('my_list.mat')`. The advantage of calling the GUI with the pair `('book', 'my_list.mat')` is that you can program the GUI to accept other user arguments, as well as valid figure properties, using the property-value pair syntax. The GUI can then identify which property the user wants to specify from the property name.

The following code shows how to program the opening function to look for the key word `'book'`, and if it finds the key word, to use the MAT-file specified by the second argument as the list of contacts.

```
function address_book_OpeningFcn(hObject, eventdata, handles, varargin)
% Choose default command line output for address_book
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
% User added code follows
if nargin < 4
    % Load the default address book
    Check_And_Load([],handles);
    % If first element in varargin is 'book' and the second element is a
    % MATLAB file, then load that file
```

```
elseif (length(varargin) == 2 & strcmpi(varargin{1},'book') & (2 ==
exist(varargin{2},'file')))
    Check_And_Load(varargin{2},handles);
else
    errordlg('File Not Found','File Load Error')
    set(handles.Contact_Name,'String','')
    set(handles.Contact_Phone,'String','')
end
```

Loading an Address Book Into the Reader

There are two ways in which an address book (i.e., a MAT-file) is loaded into the GUI:

- When running the GUI, you can specify a MAT-file as an argument. If you do not specify an argument, the GUI loads the default address book (`addrbook.mat`).
- The user can select **Open** under the **File** menu to browse for other MAT-files.

Validating the MAT-file

To be a valid address book, the MAT-file must contain a structure called `Addresses` that has two fields called `Name` and `Phone`. The `Check_And_Load` subfunction validates and loads the data with the following steps:

- Loads (`load`) the specified file or the default if none is specified.
- Determines if the MAT-file is a valid address book.
- Displays the data if it is valid. If it is not valid, displays an error dialog (`errordlg`).
- Returns `1` for valid MAT-files and `0` if invalid (used by the **Open** menu callback)
- Saves the following items in the `handles` structure:
  - The name of the MAT-file
  - The `Addresses` structure
  - An index pointer indicating which name and phone number are currently displayed

Check_And_Load Code Listing

This is the `Check_And_Load` function.

```
function pass = Check_And_Load(file,handles)
% Initialize the variable "pass" to determine if this is a valid
% file.
pass = 0;
% If called without any file then set file to the default file
% name.
% Otherwise if the file exists then load it.
if isempty(file)
   file = 'addrbook.mat';
   handles.LastFile = file;
   guidata(handles.Address_Book,handles)
end
if exist(file) == 2
   data = load(file);
end
% Validate the MAT-file
% The file is valid if the variable is called "Addresses" and it
% has fields called "Name" and "Phone"
flds = fieldnames(data);
if (length(flds) == 1) & (strcmp(flds{1},'Addresses'))
   fields = fieldnames(data.Addresses);
   if (length(fields) == 2) &(strcmp(fields{1},'Name')) &
(strcmp(fields{2},'Phone'))
        pass = 1;
   end
end
% If the file is valid, display it
if pass
   % Add Addresses to the handles structure
   handles.Addresses = data.Addresses;
   guidata(handles.Address_Book,handles)
   % Display the first entry
   set(handles.Contact_Name,'String',data.Addresses(1).Name)
   set(handles.Contact_Phone,'String',data.Addresses(1).Phone)
   % Set the index pointer to 1 and save handles
   handles.Index = 1;
   guidata(handles.Address_Book,handles)
```

```
else
   errordlg('Not a valid Address Book','Address Book Error')
end
```

The Open Menu Callback

The address book GUI contains a **File** menu that has an **Open** submenu for loading address book MAT-files. When selected, **Open** displays a dialog (`uigetfile`) that enables the user to browser for files. The dialog displays only MAT-files, but users can change the filter to display all files.

The dialog returns both the filename and the path to the file, which is then passed to `fullfile` to ensure the path is properly constructed for any platform. `Check_And_Load` validates and load the new address book.

Open_Callback Code Listing

```
function Open_Callback(hObject, eventdata, handles)
[filename, pathname] = uigetfile( ...
   {'*.mat', 'All MAT-Files (*.mat)'; ...
       '*.*','All Files (*.*)'}, ...
   'Select Address Book');
% If "Cancel" is selected then return
if isequal([filename,pathname],[0,0])
   return
% Otherwise construct the fullfilename and Check and load the file
else
   File = fullfile(pathname,filename);
   % if the MAT-file is not valid, do not save the name
   if Check_And_Load(File,handles)
       handles.LastFIle = File;
       guidata(hObject, handles)
   end
end
```

See the "Creating Menus — The Menu Editor" on page 4-17 section for information on creating the menu.

The Contact Name Callback

The **Contact Name** text box displays the name of the address book entry. If you type in a new name and press enter, the callback performs these steps:

- If the name exists in the current address book, the corresponding phone number is displayed.
- If the name does not exist, a question dialog (questdlg) asks you if you want to create a new entry or cancel and return to the name previously displayed.
- If you create a new entry, you must save the MAT-file with the **File -> Save** menu.

Storing and Retrieving Data

This callback makes use of the handles structure to access the contents of the address book and to maintain an index pointer (handles.Index) that enables the callback to determine what name was displayed before it was changed by the user. The index pointer indicates what name is currently displayed. The address book and index pointer fields are added by the Check_And_Load function when the GUI is run.

If the user adds a new entry, the callback adds the new name to the address book and updates the index pointer to reflect the new value displayed. The updated address book and index pointer are again saved (guidata) in the handles structure.

Contact Name Callback

```
function Contact_Name_Callback(hObject, eventdata, handles)
% Get the strings in the Contact Name and Phone text box
Current_Name = get(handles.Contact_Name,'string');
Current_Phone = get(handles.Contact_Phone,'string');
% If empty then return
if isempty(Current_Name)
   return
end
% Get the current list of addresses from the handles structure
Addresses = handles.Addresses;
% Go through the list of contacts
% Determine if the current name matches an existing name
for i = 1:length(Addresses)
   if strcmp(Addresses(i).Name,Current_Name)
       set(handles.Contact_Name,'string',Addresses(i).Name)
       set(handles.Contact_Phone,'string',Addresses(i).Phone)
       handles.Index = i;
       guidata(hObject, handles)
```

```
        return
    end
end
% If it's a new name, ask to create a new entry
Answer=questdlg('Do you want to create a new entry?', ...
    'Create New Entry', ...
    'Yes','Cancel','Yes');
switch Answer
case 'Yes'
    Addresses(end+1).Name = Current_Name; % Grow array by 1
    Addresses(end).Phone = Current_Phone;
    index = length(Addresses);
    handles.Addresses = Addresses;
    handles.Index = index;
    guidata(hObject, handles)
    return
case 'Cancel'
    % Revert back to the original number

set(handles.Contact_Name,'String',Addresses(handles.Index).Name)

set(handles.Contact_Phone,'String',Addresses(handles.Index).Phon
e)
    return
end
```

The Contact Phone Number Callback

The **Contact Phone #** text box displays the phone number of the entry listed in the **Contact Name** text box. If you type in a new number click one of the push buttons, the callback opens a question dialog that asks you if you want to change the existing number or cancel your change.

Like the **Contact Name** text box, this callback uses the index pointer (`handles.Index`) to update the new number in the address book and to revert to the previously displayed number if the user selects **Cancel** from the question dialog. Both the current address book and the index pointer are saved in the `handles` structure so that this data is available to other callbacks.

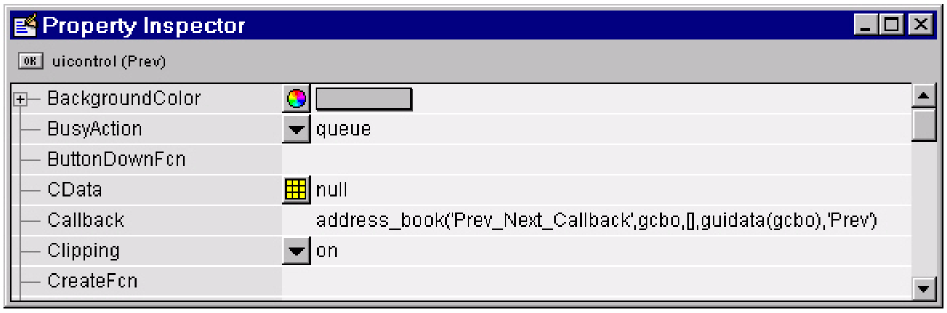
If you create a new entry, you must save the MAT-file with the **File –> Save** menu.

Code Listing

```
function Contact_Phone_Callback(hObject, eventdata, handles)
Current_Phone = get(handles.Contact_Phone,'string');
% If either one is empty then return
if isempty(Current_Phone)
    return
end
% Get the current list of addresses from the handles structure
Addresses = handles.Addresses;
Answer=questdlg('Do you want to change the phone number?', ...
    'Change Phone Number', ...
    'Yes','Cancel','Yes');
switch Answer
case 'Yes'
    % If no name match was found create a new contact
    Addresses(handles.Index).Phone = Current_Phone;
    handles.Addresses = Addresses;
    guidata(hObject, handles)
    return
case 'Cancel'
    % Revert back to the original number
    set(handles.Contact_Phone,'String',Addresses(handles.Index).Phone)
    return
end
```

Paging Through the Address Book — Prev/Next

The **Prev** and **Next** buttons page back and forth through the entries in the address book. Both push buttons use the same callback, `Prev_Next_Callback`. You must set the `Callback` property of both push buttons to call this subfunction, as the following illustration of the **Prev** push button `Callback` property setting shows.



```
BackgroundColor
BusyAction        queue
ButtonDownFcn
CData             null
Callback          address_book('Prev_Next_Callback',gcbo,[],guidata(gcbo),'Prev')
Clipping          on
CreateFcn
```

Determining Which Button Is Clicked

The callback defines an additional argument, `str`, that indicates which button, **Prev** or **Next**, was clicked. For the **Prev** button `Callback` property (illustrated above), the `Callback` string includes `'Prev'` as the last argument. The **Next** button `Callback` string includes `'Next'` as the last argument. The value of `str` is used in `case` statements to implement each button's functionality (see the code listing below).

Paging Forward or Backward

`Prev_Next_Callback` gets the current index pointer and the addresses from the `handles` structure and, depending on which button the user presses, the index pointer is decremented or incremented and the corresponding address and phone number are displayed. The final step stores the new value for the index pointer in the `handles` structure and saves the updated structure using `guidata`.

Code Listing

```
function Prev_Next_Callback(hObject, eventdata,handles,str)
% Get the index pointer and the addresses
index = handles.Index;
Addresses = handles.Addresses;
% Depending on whether Prev or Next was clicked change the display
switch str
case 'Prev'
    % Decrease the index by one
    i = index - 1;
    % If the index is less then one then set it equal to the index of the
```

```
        % last element in the Addresses array
        if i < 1
            i = length(Addresses);
        end
    case 'Next'
        % Increase the index by one
        i = index + 1;
        % If the index is greater than the size of the array then point
        % to the first item in the Addresses array
        if i > length(Addresses)
            i = 1;
        end
    end
    % Get the appropriate data for the index in selected
    Current_Name = Addresses(i).Name;
    Current_Phone = Addresses(i).Phone;
    set(handles.Contact_Name,'string',Current_Name)
    set(handles.Contact_Phone,'string',Current_Phone)
    % Update the index pointer to reflect the new index
    handles.Index = i;
    guidata(hObject, handles)
```

Saving Changes to the Address Book from the Menu

When you make changes to an address book, you need to save the current MAT-file, or save it as a new MAT-file. The **File** submenus **Save** and **Save As** enable you to do this. These menus, created with the Menu Editor, use the same callback, `Save_Callback`.

The callback uses the menu `Tag` property to identify whether **Save** or **Save As** is the callback object (i.e., the object whose handle is passed in as the first argument to the callback function). You specify the menu's `Tag` property with the Menu Editor.

Saving the Addresses Structure

The `handles` structure contains the `Addresses` structure, which you must save (`handles.Addresses`) as well as the name of the currently loaded MAT-file (`handles.LastFile`). When the user makes changes to the name or number, the `Contact_Name_Callback` or the `Contact_Phone_Callback` updates `handles.Addresses`.

Saving the MAT-File

If the user selects **Save**, the save command is called to save the current MAT-file with the new names and phone numbers.

If the user selects **Save As**, a dialog is displayed (uiputfile) that enables the user to select the name of an existing MAT-file or specify a new file. The dialog returns the selected filename and path. The final steps include

- Using fullfile to create a platform-independent pathname.
- Calling save to save the new data in the MAT-file.
- Updating the handles structure to contain the new MAT-file name.
- Calling guidata to save the handles structure.

Save_Callback Code Listing

```
function Save_Callback(hObject, eventdata, handles)
% Get the Tag of the menu selected
Tag = get(hObject, 'Tag');
% Get the address array
Addresses = handles.Addresses;
% Based on the item selected, take the appropriate action
switch Tag
case 'Save'
   % Save to the default addrbook file
   File = handles.LastFile;
   save(File,'Addresses')
case 'Save_As'
   % Allow the user to select the file name to save to
   [filename, pathname] = uiputfile( ...
       {'*.mat';'*.*'}, ...
       'Save as');
   % If 'Cancel' was selected then return
   if isequal([filename,pathname],[0,0])
       return
   else
       % Construct the full path and save
       File = fullfile(pathname,filename);
       save(File,'Addresses')
       handles.LastFile = File;
       guidata(hObject, handles)
```

```
    end
end
```

The Create New Menu

The **Create New** menu simply clears the **Contact Name** and **Contact Phone #** text fields to facilitate adding a new name and number. After making the new entries, the user must then save the address book with the **Save** or **Save As** menus. This callback sets the text `String` properties to empty strings:

```
function New_Callback(hObject, eventdata, handles)
set(handles.Contact_Name,'String','')
set(handles.Contact_Phone,'String','')
```

The Address Book Resize Function

The address book defines its own resize function. To use this resize function, you must set the **Application Options** dialog **Resize behavior** to `User-specified`, which in turn sets the figure's `ResizeFcn` property to:

```
address_book('ResizeFcn',gcbo,[],guidata(gcbo))
```

Whenever the user resizes the figure, MATLAB calls the `ResizeFcn` subfunction in the address book M-file (`address_book.m`)

Behavior of the Resize Function

The `resize` function allows users to make the figure wider, to accommodate long names and numbers, but does not allow the figure to be made narrower than its original width. Also, users cannot change the height. These restrictions do not limit the usefulness of the GUI and simplify the `resize` function, which must maintain the proper proportions between the figure size and the components in the GUI.

When the user resizes the figure and releases the mouse, the resize function executes. At that point, the resized figure's dimensions are saved. The following sections describe how the `resize` function handles the various possibilities.

Changing the Width

If the new width is greater than the original width, set the figure to the new width.

The size of the **Contact Name** text box changes in proportion to the new figure width. This is accomplished by:

- Changing the `Units` of the text box to `normalized`.
- Resetting the width of the text box to be 78.9% of the figure's width.
- Returning the `Units` to `characters`.

If the new width is less than the original width, use the original width.

Changing the Height

If the user attempts to change the height, use the original height. However, because the resize function is triggered when the user releases the mouse button after changing the size, the resize function cannot always determine the original position of the GUI on screen. Therefore, the resize function applies a compensation to the vertical position (second element in the figure `Position` vector) as follows:

vertical position when mouse released + height when mouse released minus the original height

When the figure is resized from the bottom, it stays in the same position. When resized from the top, the figure moves to the location where the mouse button is released.

Ensuring the Resized Figure Is On Screen

The `resize` function calls `movegui` to ensure that the resized figure is on screen regardless of where the user releases the mouse.

When the GUI is first run, it is displayed at the size and location specified by the figure `Position` property. You can set this property with the Property Inspector when you create the GUI.

Code Listing

```
function ResizeFcn(hObject, eventdata, handles)
% Get the figure size and position
Figure_Size = get(hObject, 'Position');
% Set the figure's original size in character units
Original_Size = [ 0 0 94 19.230769230769234];
% If the resized figure is smaller than the
% original figure size then compensate
if (Figure_Size(3)<Original_Size(3)) | (Figure_Size(4) ~= Original_Size(4))
```

```
      if Figure_Size(3) < Original_Size(3)
          % If the width is too small then reset to origianl width
          set(hObject, 'Position',...
    [Figure_Size(1) Figure_Size(2) Original_Size(3) Original_Size(4)])
          Figure_Size = get(hObject, 'Position');
      end
      if Figure_Size(4) ~= Original_Size(4)
          % Do not allow the height to change
          set(hObject, 'Position',...
    [Figure_Size(1), Figure_Size(2)+Figure_Size(4)-Original_Size(4),...
    Figure_Size(3), Original_Size(4)])
      end
end
% Adjust the size of the Contact Name text box
% Set the units of the Contact Name field to 'Normalized'
set(handles.Contact_Name,'units','normalized')
% Get its Position
C_N_pos = get(handles.Contact_Name,'Position');
% Reset it so that it's width remains normalized relative to figure
set(handles.Contact_Name,'Position',...
    [C_N_pos(1) C_N_pos(2)  0.789 C_N_pos(4)])
% Return the units to  Characters
set(handles.Contact_Name,'units','characters')
% Reposition GUI on screen
movegui(hObject, 'onscreen')
```

Index