MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

The MATH WORKS Inc.

Creating Graphical User Interfaces

Version 1

How to Contact The MathWorks:

☎	508-647-7000	Phone
	508-647-7001	Fax
✉	The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail
	`http://www.mathworks.com`	Web
	`ftp.mathworks.com`	Anonymous FTP server
	`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
	`subscribe@mathworks.com`	Subscribing user registration
	`service@mathworks.com`	Order status, license renewals, passcodes
	`info@mathworks.com`	Sales, pricing, and general information

Creating Graphical User Interfaces

Printing History: November 2000 New for MATLAB 6.0 (Release12) (Online only)

Contents

1 Creating GUIs with MATLAB

2 GUI Building Tools

3 Programming GUIs

4 Application Techniques

1

Creating GUIs with MATLAB

Introduction

A graphical user interface (GUI) is a user interface built with graphical objects, such as buttons, text fields, sliders, and menus. In general, these objects already have meanings to most computer users. For example, when you move a slider, a value changes; when you press an "OK" button, your settings are applied and a dialog box is dismissed. Of course, to leverage this built-in familiarity, you must be consistent in how you use the various GUI-building components.

Applications that provide GUIs are generally easier to learn and use since the person using the application does not need to know what commands are available or how they work. The action that results from a particular user action can be made clear by the design of the interface.

The sections that follow describe how to create GUIs with MATLAB. This includes laying out the components, programming them to do specific things in response to user actions, and saving and launching the GUI; in other words, the mechanics of creating GUIs. This documentation does not attempt to cover the "art" of good user interface design, which is an entire field unto itself. Topics covered in this section include:

- Creating GUIs with GUIDE – an overview of the GUI creation process in MATLAB.
- Editing Version 5 GUIs with Version 6 GUIDE – suggestions on how to proceed if you want to edit your pre-version 6 GUI with GUIDE.
- Selecting GUIDE Application Options – discussion of the various options you can select when begining your GUI impelementation.

Creating GUIs with GUIDE

MATLAB implements GUIs as figure windows containing various styles of uicontrol objects. You must program each object to perform the intended action when activated by the user of the GUI. In addition, you must be able to save and launch your GUI. All of these tasks are simplified by GUIDE, MATLAB's Graphical User Interface Development Environment.

GUI Development Environment

The process of implementing a GUI involves two basic tasks:

- Laying out the GUI components
- Programming the GUI components

GUIDE is primarily 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 also provides a framework for the implementation of the *callbacks* – the functions that execute when users activate a component in the GUI.

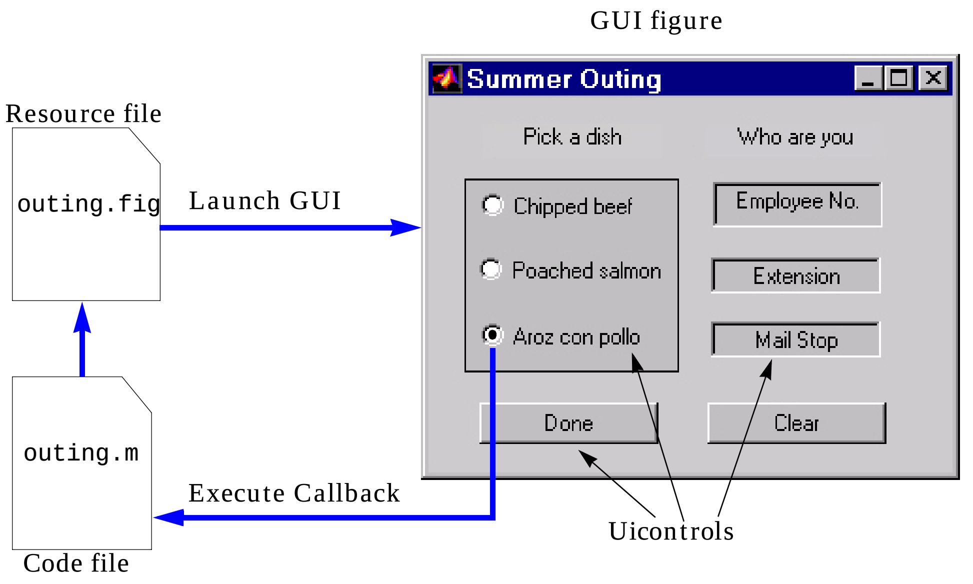
The Implementation of a GUI

While it is possible to write an M-file that contains all the commands to lay out a GUI, it is easier to use GUIDE to lay out the components interactively and to generate two files that save and launch the GUI:

- A FIG-file – 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.
- An M-file – contains the functions that launch and control the GUI and the callbacks, which are defined as subfunctions. This M-file is referred to as the *application M-file* in this documentation.

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

The following diagram illustrates the parts of a GUI implementation.



Features of the GUIDE-Generated Application M-File

GUIDE simplifies the creation of GUI applications by automatically generating an M-file framework directly from your layout. You can then use this framework to code your application M-file. This approach provides a number of advantages:

- The M-file contains code to implement a number of useful features (see Configuring Application Options for information on these features).
- The M-file adopts an effective approach to managing object handles and executing callback routines (see Creating and Storing the Object Handle Structure for more information).
- The M-files provides a way to manage global data. (see Managing GUI Data for more information).
- The automatically inserted subfunction prototypes for callback routines ensure compatibility with future releases. For more information, see Generating Callback Function Prototypes for information on syntax and arguments.

You can elect to have GUIDE generate only the FIG-file and write the application M-file yourself. There are no uicontrol creation commands in the application M-file; the layout information is contained in the FIG-file generated by the Layout Editor.

Beginning the Implementation Process

To begin implementing your GUI, proceed to the following sections:

- Selecting GUIDE Application Options – to set both FIG-file and M-file options.
- Using the Layout Editor – to begin laying out the GUI.
- Understanding the Application M-File – to understand programming techniques used in the application M-file.
- Application Techniques – to see a collection of examples that illustrate techniques that are useful for implementing GUIs.

Editing Version 5 GUIs with Version 6 GUIDE

MATLAB Version 5 GUIDE saved GUI layouts as MAT-file/M-file pairs. In Version 6, GUIDE saves GUI layouts as FIG-files. You can set GUIDE application options so that GUIDE also generates an M-file to program the GUI callbacks.

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.

```
h = 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(h)
```

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 are no longer used. To display the revised GUI, use the `open` or `hgload` command to load the newly created FIG-file, or you can also run the application M-file.

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 application M-file generated by Version 6 GUIDE can provide a model of how to restructure your code.

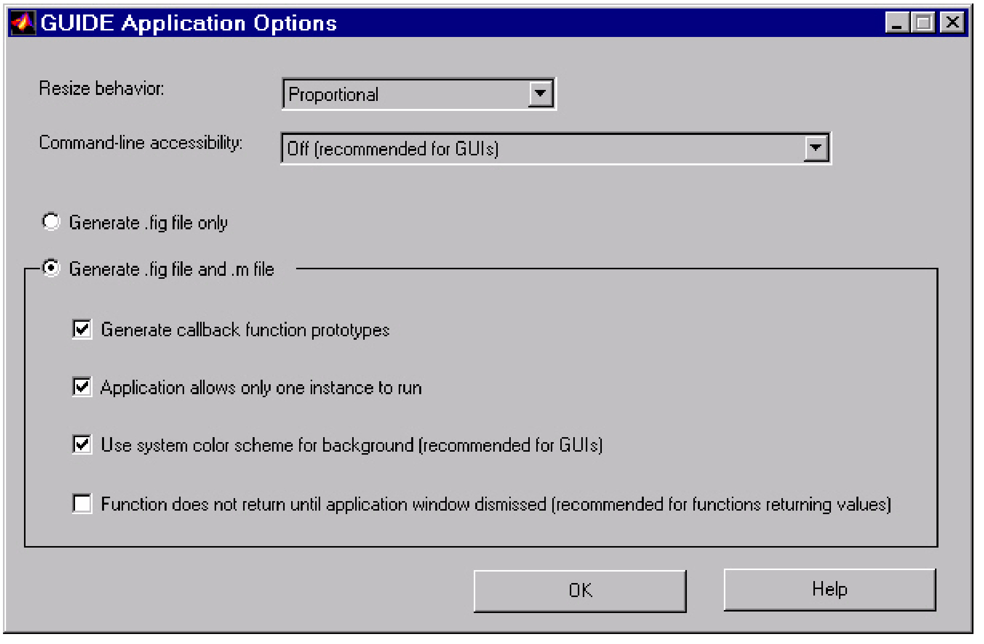
Note By default, GUIDE generates an application M-file having the same name as the FIG-file saved with the Layout Editor. When you activate a GUI from the Layout Editor, GUIDE attempts to execute this M-file to launch the GUI.

Selecting GUIDE Application Options

Issuing the `guide` command displays an empty Layout Editor with an untitled figure. Before adding components to the layout, you should configure the GUI using the GUIDE Application Options dialog. Access the dialog by selecting **Application Options** from the Layout Editor **Tools** menu.

Configuring the Application M-file

The GUIDE Application 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 its companion application M-file. You can also select a number of different behaviors for your GUI.



The following section describes the options in this dialog.

- Resize behavior
- Command-line accessibility

- Generate .fig file only
- Generate .fig file and .m file
- Generate callback function prototypes
- Application allows only one instance to run
- Use system color scheme for background
- Function does not return until application window dismissed

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** – let MATLAB automatically rescale 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 require only appropriate settings for certain properties. User-specified resizing requires you to write a callback routine that recalculate 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, these GUIs are dismissed so changing their size is not worthwhile.

Property Settings

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

- `Units` properties of the figure, axes, and uicontrols should be set to `characters` (the Layout Editor default) so the GUI displays at the correct size at runtime.
- `Resize` figure property set to `off`.
- `ResizeFcn` figure property does not require a callback routine.

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 within the figure window. 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. Users may want to resize these window to better fit them on the screen with other windows, but the precise layout to the GUI is not critical to its function.

Property Settings

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

- `Units` properties of the axes and uicontrols should be set to `normalized` so the these components resize and reposition as the figure window changes size.
- `Units` property of the figure should be set to `characters` so the GUI displays at the correct size at runtime.
- `Resize` figure property set to `on` (the default).
- `ResizeFcn` figure property does not require a callback routine.

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 essentially recalculates 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 should generally be set to `characters` so the GUI displays at the correct size at runtime.
- `Resize` figure property set to `on` (the default).
- `ResizeFcn` figure property requires a callback routine to handle resizing.

See The Address Book Resize Function for an example of a user-written resize function.

Command-Line Accessibility

When MATLAB creates a graph, the figure and axes are included in the list of children of their respective parents and their handles are available through commands such as `findobj`, `set`, and `get`. If you issue another plotting command, the output is directed to the current figure and axes.

GUIs are also created in figure windows. Generally, you do not want GUI figures to be available as targets for graphics output, since issuing a plotting command could direct the output to the GUI figure, resulting in the graph appearing in the middle of the GUI.

In contrast, if you create a GUI that contains an axes, such as a plotting tool, users need access to the figure. In this case, you should enable command-line access.

Access Options

The GUIDE Application Options dialog provides three options to control user access:

- **Off** – prevent command-line access to the GUI figure (default).
- **On** – enable command-line access to the GUI figure.
- **User-specified** – the GUI uses the values you set for the figure `HandleVisibility` and `IntegerHandle` properties.

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 located the handles of the uicontrols in the GUI. As an alternative, the application M-file creates a object handle structure that contains the handles of each uicontrol in the GUI and passes this structure to subfunctions.

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.

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)`).

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.

Electing to Generate Only the FIG-File

Select **Generate .fig file only** in the GUIDE Application Options dialog if you do not want GUIDE to generate an application M-file. When you save the GUI from the Layout Editor, GUIDE creates a FIG-file, which you can redisplay using the `open` or `hgload` 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 follow a completely different programming paradigm than that generated by the application M-file.

Generating the FIG-File and the M-File

Select **Generate .fig file and .m file** in the GUIDE Application Options dialog if you want GUIDE to create both the FIG-file and the application 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
- Use system color scheme for background
- Function does not return until application window dismissed

Generating Callback Function Prototypes

When you select **Generate callback function prototypes** in the GUIDE Application Options dialog, GUIDE adds a subfunction to the application M-file for any component you add to the GUI (note that frame and static text components do not use their `Callback` property). 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 right-click context menu.

Callback Function Syntax and Naming

The callback function syntax is of the form

```
function objectTag_Callback(h,eventdata,handles,varargin)
```

The arguments are listed in the following table.

Callback Function Arguments

`h`	The handle of the object whose callback is executing.
`eventdata`	Empty, reserved for future use.

Callback Function Arguments

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 main program.
varargin	A variable-length list of arguments that you want to be passed to the callback function.

For example, if you create a layout having a push button that has a `Tag` property set to `pushbutton1`, then GUIDE generates this subfunction in the application M-file.

```
function pushbutton1_Callback(h,eventdata,handles,varargin)
```

GUIDE then sets the `Callback` property of this push button to

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

where:

- `mygui` – is the name of the FIG-file saved for this GUI.
- `pushbutton1_Callback` – is the name of the callback subfunction.
- `gcbo` – is a command that returns the handle of the push button.
- `[]` – is an empty matrix used as a place holder for the `eventdata` argument.
- `guidata(gcbo)` – gets the `handles` structure from the figure's application data.

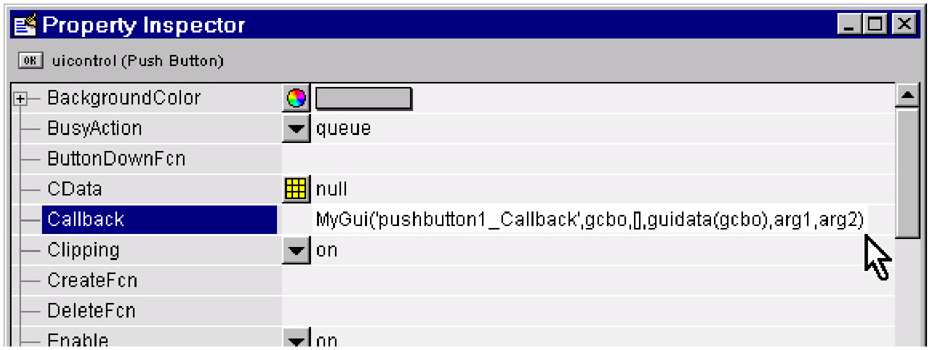
If you want to pass additional arguments to the push button's callback routine, edit the `Callback` property in the Property Inspector to add a comma-separated list of arguments (which are then handled by the `varargin` argument in the subfunction).

For example, it you want to add two arguments to the callback of a push button in the application M-file named `MyGui.m`, you need to edit the syntax in two places:

- Edit the function line of the callback subfunction to add the argument.

```
function varargout = pushbutton1_Callback(h,eventdata,handles,arg1,arg2)
```

- Edit the push button's callback to add the arguments.



Application Allows Only One Instance to Run

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 launch the GUI is issued. If a GUI already exists, MATLAB brings it to the foreground rather than creating a new figure.

If you uncheck this option, MATLAB creates a new GUI figure whenever the command to launch the GUI is issued.

Code in the Application M-File

GUIDE implements this feature by generating code in the application M-file that uses the `openfig` command. The `reuse` or `new` string specifies one instance or multiple instances of the GUI figure.

```
fig = openfig(mfilename,'reuse');
```

or

```
fig = openfig(mfilename,'new');
```

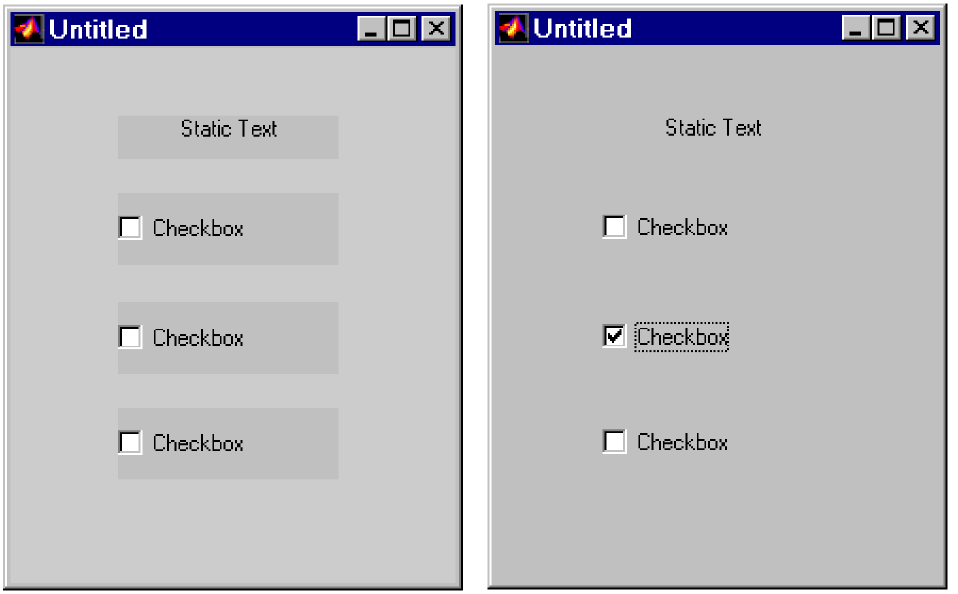
Note Ensure that you have only one occurrence of `openfig` in your application M-file, including in commented lines.

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 (right) and without (left) system color matching.



Code in the Application M-File

GUIDE implements this feature by generating code in the application M-file that sets the figure background color to the default uicontrol background color, which is system dependent.

```
% Use system color scheme for figure:
set(fig,'Color',get(0,'DefaultUicontrolBackgroundColor'));
```

Note Ensure that you have only one occurrence of this statement in your application M-file, including in commented lines.

Waiting for User Input

The GUIDE application option,

Function does not return until application window dismissed

generates an application M-file that is designed to wait for user input. It does this by calling `uiwait`, which blocks further execution of the M-file.

While execution waits, MATLAB processes the event queue. This means that any user-interactions with the GUI (such as clicking a push button) can invoke callback routines, but the execution stream always returns to the application M-file until one of two events occurs:

- The GUI figure is deleted.
- A callback for an object in the GUI figure executes a `uiresume` command.

This feature provides a way to block the MATLAB command line until the user responds to the dialog box, but at the same time, allows callback routines to execute. When used in conjunction with a modal dialog, you can restrict user interaction to the dialog.

Code in the Application M-File

GUIDE implements this feature by generating code in the application M-file that uses the `uiwait` command

```
% Wait for callbacks to run and window to be dismissed:
uiwait(fig);
```

where `fig` is the handle of the GUI figure.

Note Ensure that you have only one occurrence of `uiwait` in your application M-file, including in commented lines.

Renaming Application Files and Tags

It is often desirable 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`).

It is generally a good practice to select the tags and filenames before activating or saving your GUI for the first time.

Using Save As

When you select **Save As** from the Layout Editor **File** menu, GUIDE also renames the application M-file and resets the `Callback` properties to properly execute the callbacks.

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.

Getting Everything Right

If you make changes after GUIDE has generated the M-file and FIG-file, you must ensure that your code incorporates these changes. This section describes:

- Changing component tag properties
- Changing the name of callback subfunctions
- Changing the name of the M-file and FIG-file

Changing Component Tag Properties

Guide automatically assigns a string to the uicontrol `Tag` property and uses this string to:

- Construct the name of the generated callback subfunctions (e.g., *tag*`_Callback`)
- Add a field to the `handles` structure containing the handle of the object (e.g., `handles.`*tag*).

If you change the Tag after GUIDE generates the callback subfunction, GUIDE does not generate a new subfunction. However, since the handles structure is created at run-time, GUIDE uses the new Tag to name the field that contains the objects handle.

Problems Caused by Changing Tags

Changing the Tag can cause program errors when you have referenced an objects handle. For example, the following statement,

```
file_list = get(handles.listbox1,'String');
```

gets the value of the String property from the list box whose Tag is listbox1. If you had changed the list box's Tag to file_listbox, subsequent instantiations of the GUI would require you to change the statement to:

```
file_list = get(handles.file_listbox,'String');
```

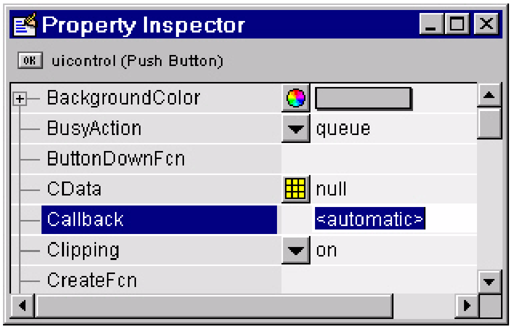
Avoiding Problems

The best approach is to set the Tag property on components when you add them to the layout. If you do change a Tag after generating the application M-file and want to rename callback subfunctions to maintain the consistent naming used by GUIDE, you should:

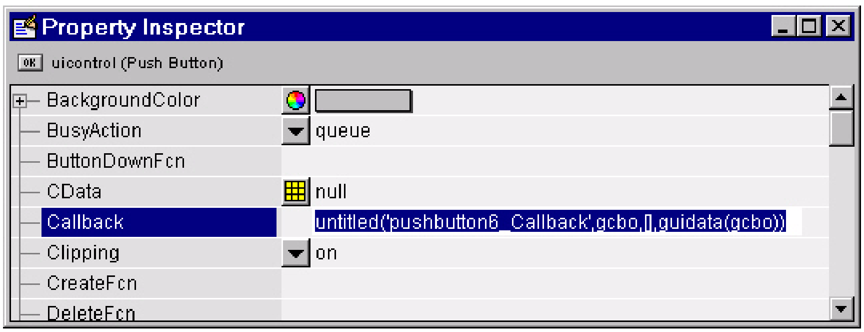
- Correct any out of date reference to the handles structure.
- See Changing the Name of Callback Subfunctions and follow the procedure described there.

Changing the Name of Callback Subfunctions

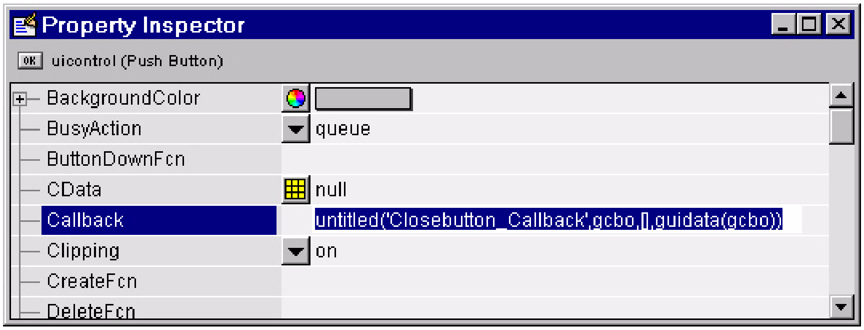
When you save or activate your GUI, GUIDE replaces the value of any Callback property that is set to <automatic> with a string that executes the callback subfunction in the application M-file. When first inserted into the layout, a push button's Callback property looks like this.



When you save or activate the figure, GUIDE changes the `Callback` property to a string that executes the callback. The following picture shows the string for the sixth push button added to the layout.



If you want to change the name of a callback, you must also change the string assigned to the `Callback` property of the uicontrol. This picture shows how the string should look after renaming the callback subfunction to `Closebutton_Callback`.



GUIDE generates similar strings for the other callback properties.

Changing the Name of the M-file and FIG-file

GUIDE gives the GUI FIG-file and its associated application M-file the same root name; only the extensions differ. When you execute the M-file to launch the GUI, the following statement uses the `mfilename` command to determine the name of the FIG-file from the name of the M-file.

```
fig = openfig(mfilename,'reuse');
```

If the FIG-file name differs from the M-file name, it is not called correctly.

2

GUI Building Tools

GUI Layout Tools

MATLAB includes a set of layout tools that simplify the process of creating graphical user interfaces (GUIs). These tools include:

- Layout Editor – add and arrange objects in the figure window.
- Alignment Tool – align objects with respect to each other.
- Property Inspector – inspect and set property values.
- Object Browser – observe a hierarchical list of the Handle Graphics objects in the current MATLAB session.
- Menu Editor – create window menus and context menus.

Access these tools from the Layout Editor. To start the Layout Editor, use the `guide` command. For example,

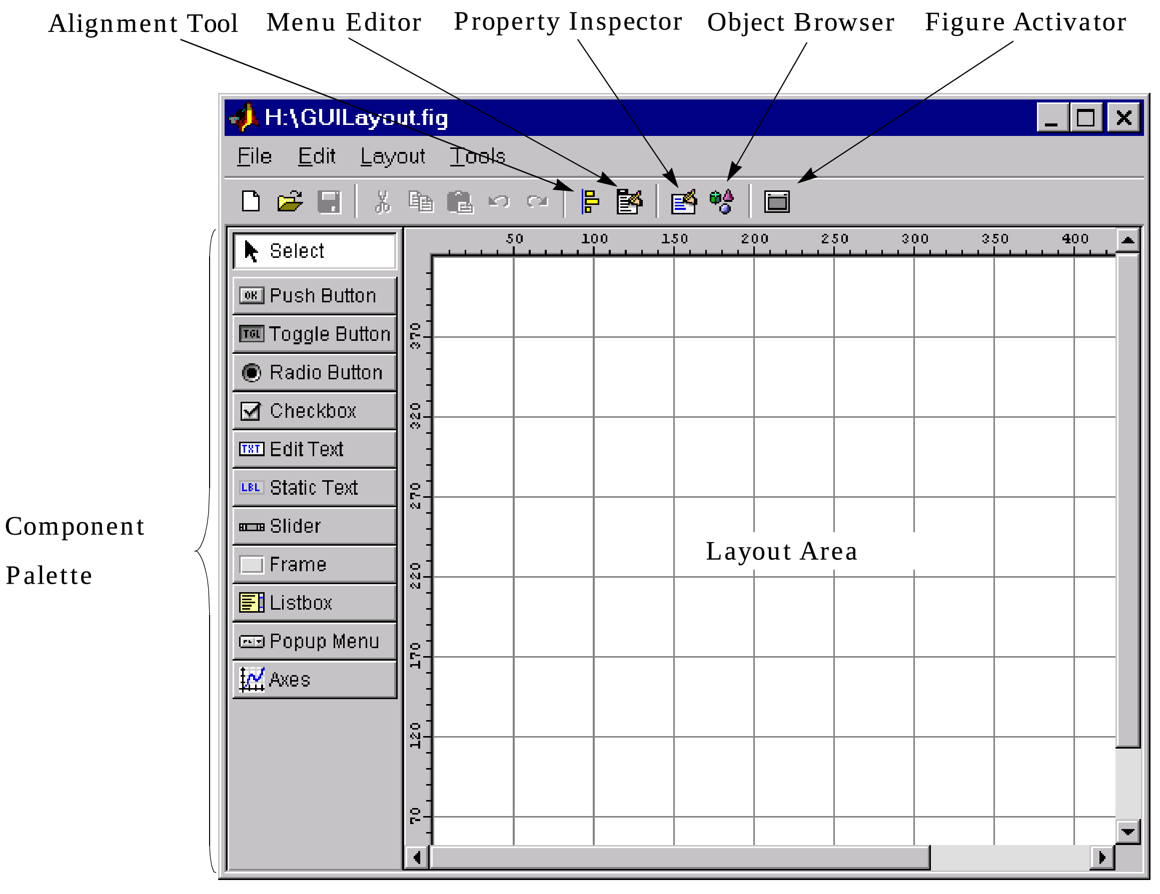
```
guide
```

displays an empty layout.

To load an existing GUI for editing, type (the `.fig` is not required)

```
guide mygui.fig
```

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



Saving Your 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 creates the application M-file automatically when you save or activate 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.

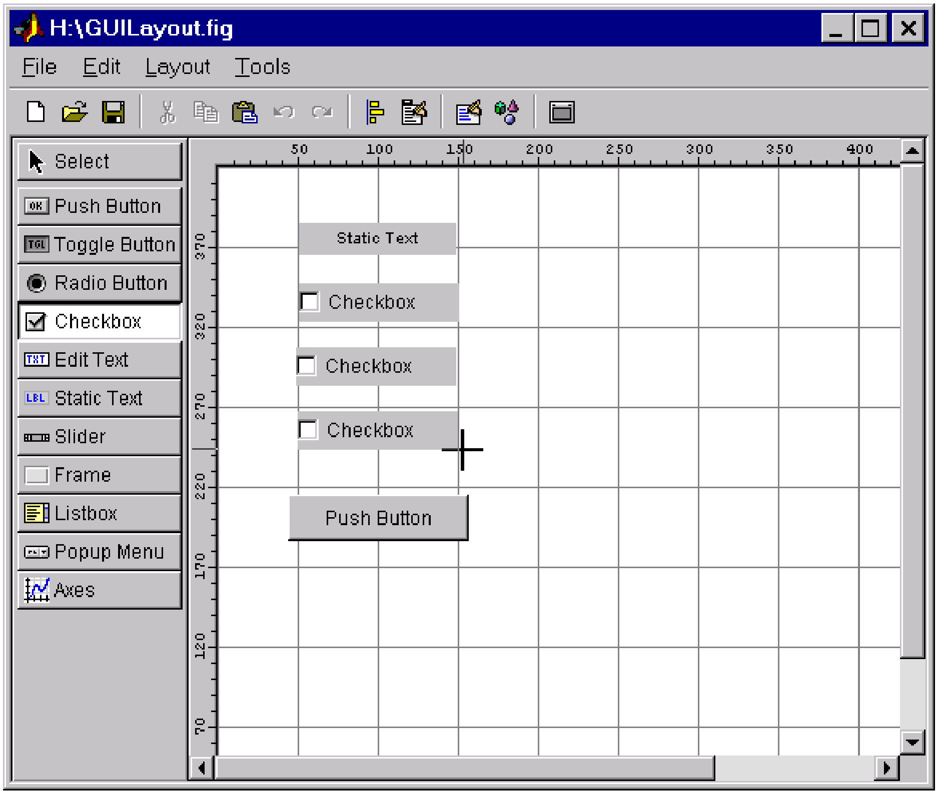
Generally, however, you launch your GUI by executing the application M-file that is generated by GUIDE. This M-file contains the commands to load the GUI and provides a framework for the component callbacks. See Configuring Application Options for more information.

Laying Out GUIs – The Layout Editor

The Layout Editor enables you to select GUI components from a palette and arrange them in a figure window. The *component palette* contains the GUI components (`uicontrol` objects) that are available for you to use in your user interface. The *layout area* becomes the figure window upon activation.

Placing an Object 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.



Activating the Figure

You can generate a functioning GUI by activating the figure you have designed with the Layout Editor. Activate the figure by selecting the **Activate Figure** item in the **Tools** menu or by clicking figure activator in the toolbar.

When you activate a figure, the following occurs:

- GUIDE first saves both the M-file and FIG-file. If you have not yet saved the layout, GUIDE opens a Save As dialog so you can select a name for the M-file GUIDE is going to generate. GUIDE then 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.

  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.

- MATLAB executes the M-file to display the GUI. The options specified in the Application Options dialog are functional in the GUI. Callbacks that you have not yet implemented, but that GUIDE inserted as stubs in the M-file, simply return a message to the command line indicating they are not yet implemented.

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

Layout Editor Context Menus

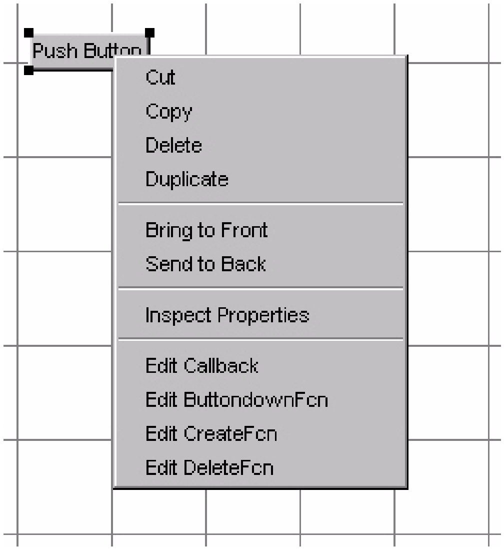
When working in the Layout Editor, you can select an object with the left mouse button and then click the right button to display a context menu. In addition to containing items found on the Layout Editor window menu, this context menu enables you to add a subfunction to your application M-file for any 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 lower panel.

GUI Component Context Menus

The following picture shows the context menu associated with uicontrol and axes objects. Note that all the properties that define callback routines for these objects are listed in the lower panel.


Push Button
Cut
Copy
Delete
Duplicate
Bring to Front
Send to Back
Inspect Properties
Edit Callback
Edit ButtondownFcn
Edit CreateFcn
Edit DeleteFcn

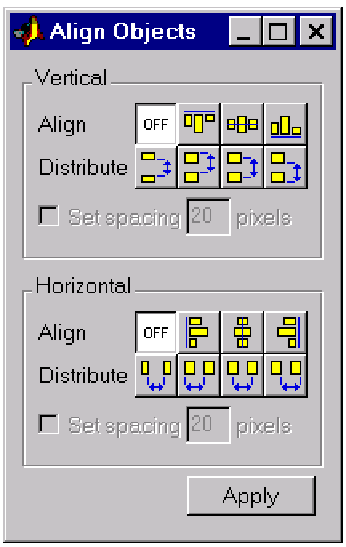
Aligning Components in the Layout Editor

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.



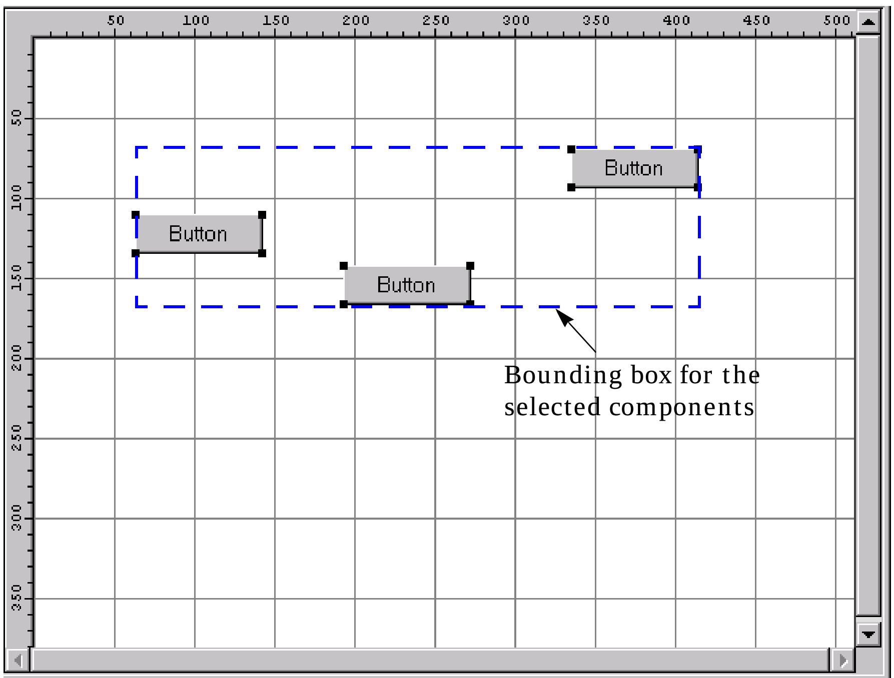
The alignment tool provides two types of alignment operations:

- **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:

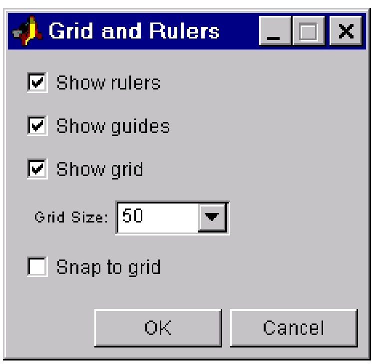
- Equally space selected components within the bounding box (default)
- Space selected components to a specified value in pixels (check **Set spacing** and specify a pixel value)

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.



Use the Grid and Rulers dialog box (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

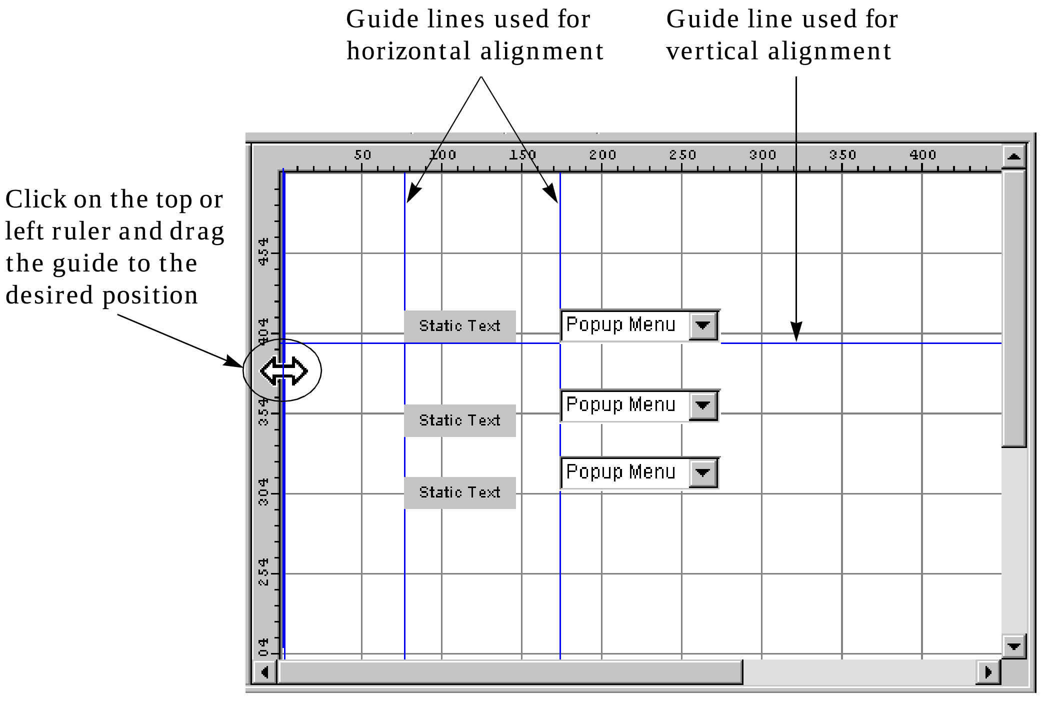
Aligning Components to Guide Lines

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.



Front to Back Positioning

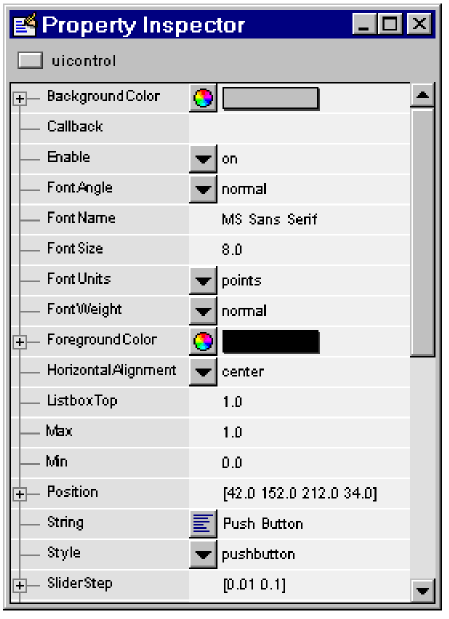
The Layout Editor provides four operations that enable you to control the front to back positioning of objects that overlap:

- 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 popup menu to set `FontAngle`, and a text field to specify the `Callback` string.



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

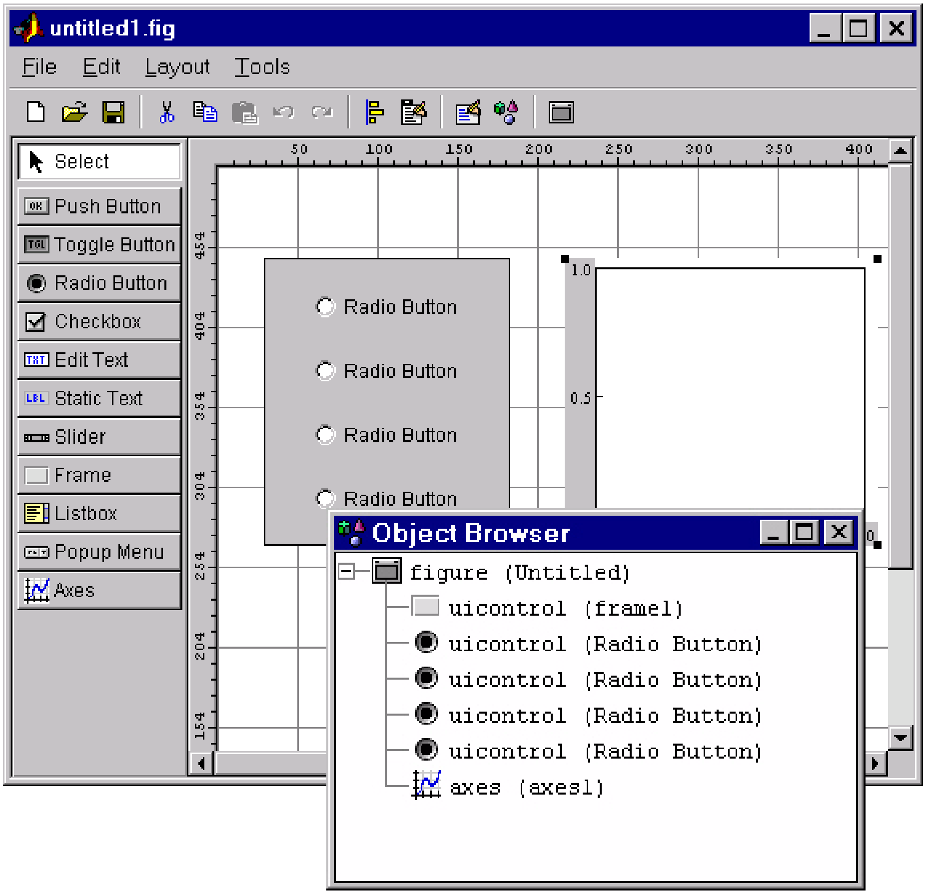
You can display the Property Inspector by:

- Double-clicking on a component in the Layout Editor.
- Selecting **Property Inspector** in the **Tools** menu.
- Selecting **Inspect Properties** in the **Edit** 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.

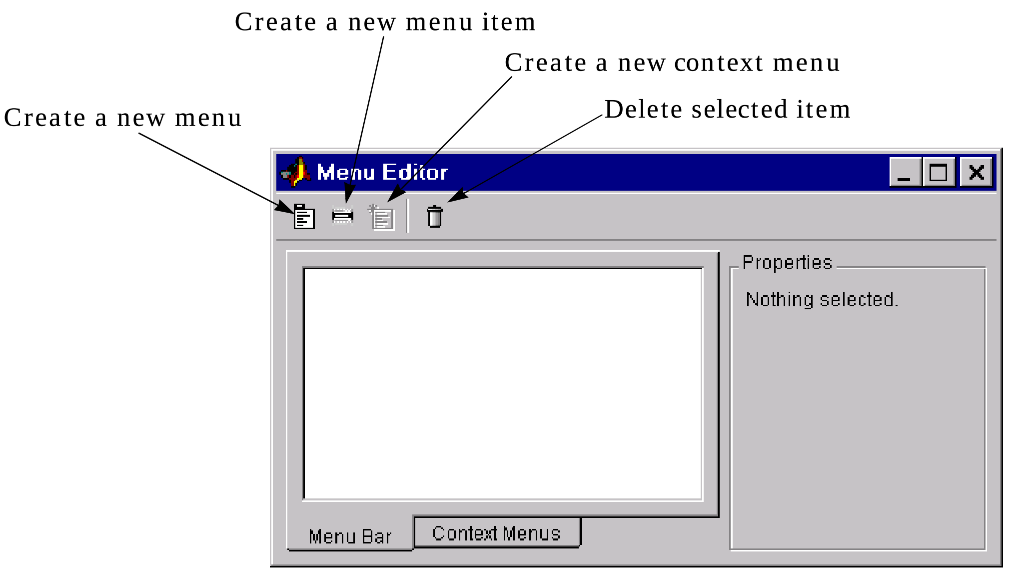


Creating Menus – The Menu Editor

MATLAB enables you to create two kinds of menus:

- Menubar 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 **Edit Menubar** item on the **Layout** menu and from the Layout Editor toolbar.



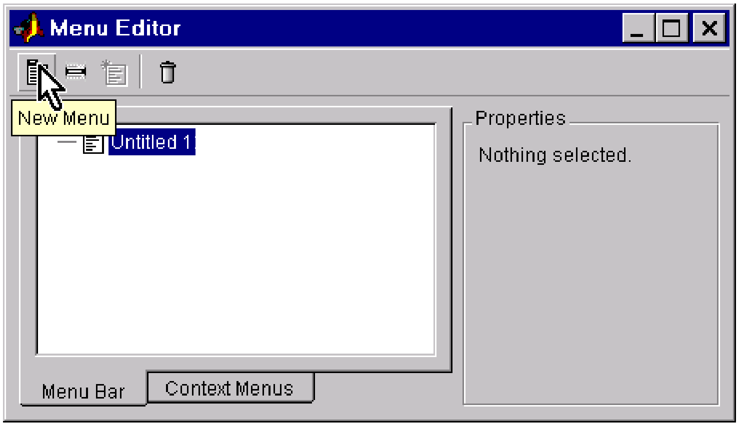
These menus are implemented with `uimenu` and `uicontextmenu` objects.

Defining Menus for the Menubar

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.

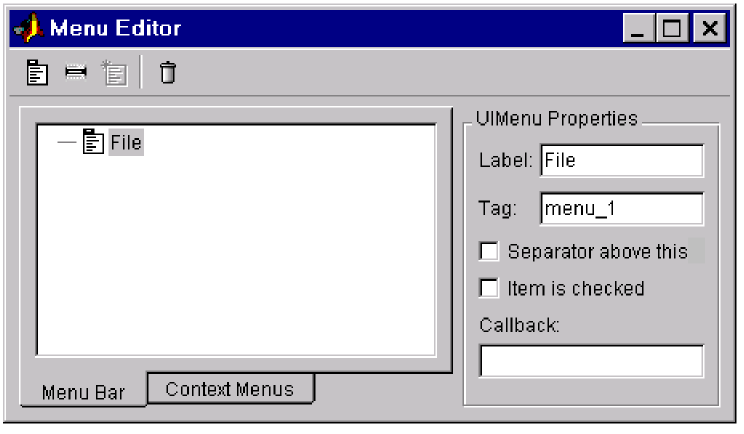
Creating a Menu

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



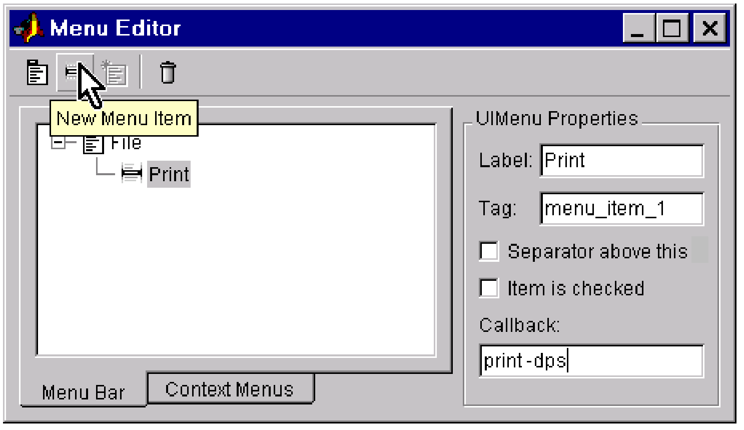
Specifying Menu Properties

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

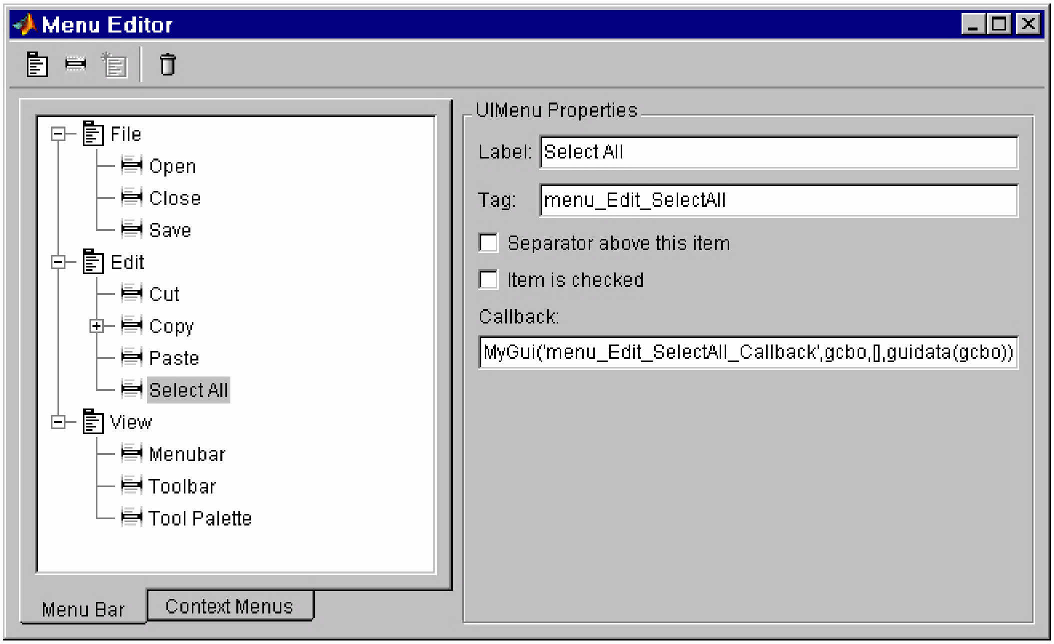
Use the New Menu Item tool to define the menu items that are displayed under the menu.



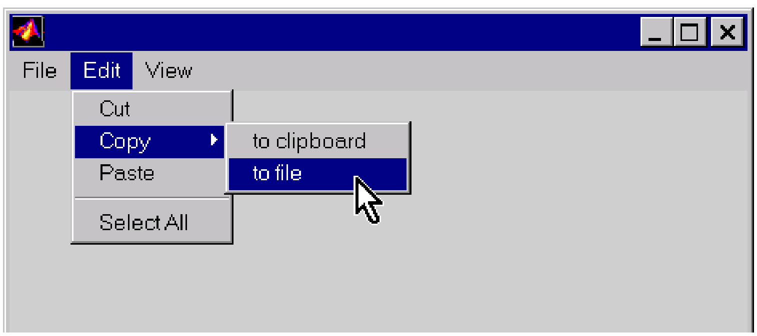
The New Menu Item adds a submenu to the selected item. For example, if you want to add more items to the **File** menu in the illustration above, select **File** before clicking on New Menu Item again.

Laying Out Three Menus

The following Menu Editor illustration show three menus defined for the figure menubar:



When you activate the figure, the menus appear in the menubar.



The Menu Callback

The menu callback executes when users select the menu item. You can type the MATLAB code to execute in the Menu Editor **Callback** text box. This approach is manageable if the callback is a very simple command (e.g., `print -dps`). However, it is generally better to add a subfunction to the application M-file, as GUIDE automatically does for uicontrol callbacks.

Note GUIDE does not automatically add a callback subfunction for menus to the application M-file. You must manually add menu callbacks.

Specifying the Callback String

The application M-file enables you to call a subfuction as a callback for components in the GUI. To do this, you must use the appropriate syntax for the callback. 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 application 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

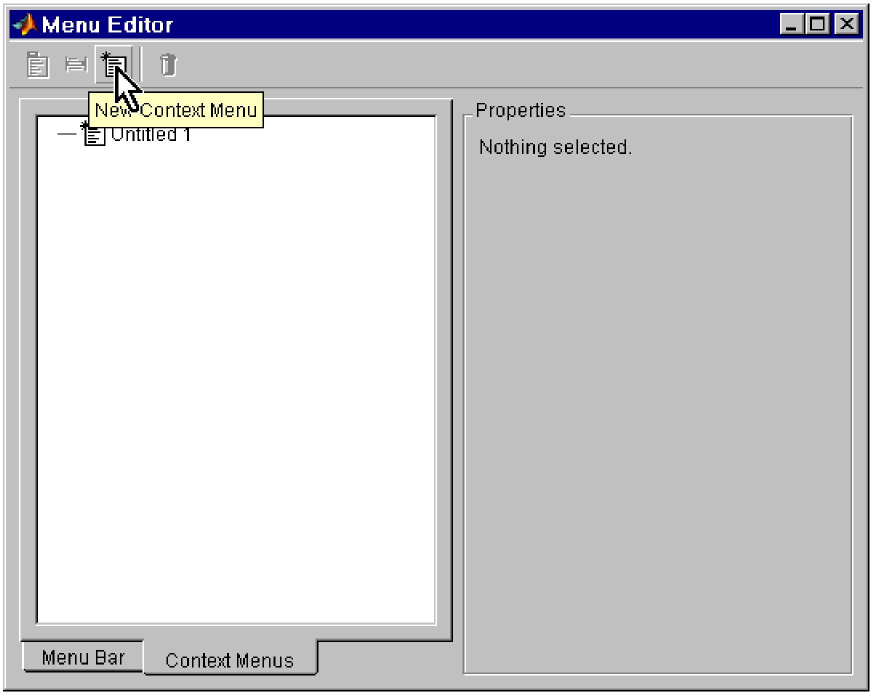
After determining the correct callback string, type it into the Menu Editor **Callback** text box. Then add the subfunction called `menu_Edit_SelectAll_Callback` to the `MyGui.m` file.

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 context menu, which is not displayed on the figure menubar. To define the parent menu, select **New Context Menu** from the Menu Editor's toolbar.

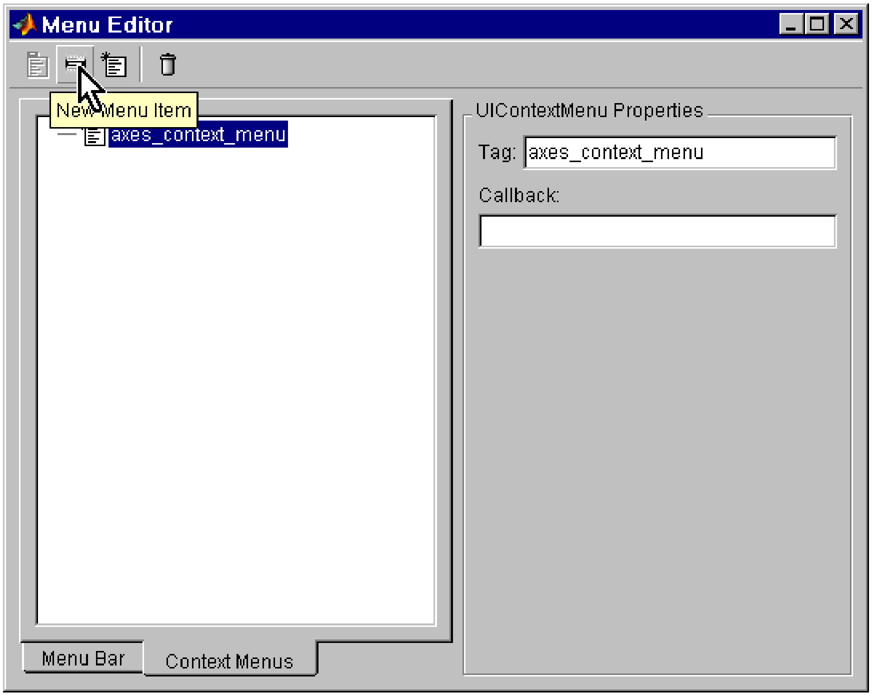


Note You must select the Menu Editor's **Context Menus** tab before you begin to define a context menu.

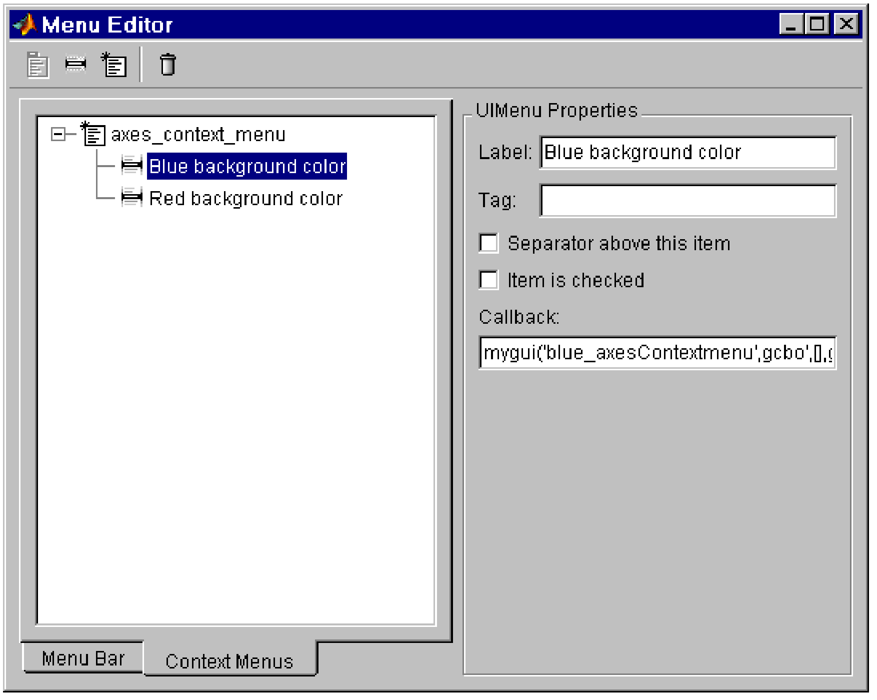
Add a `Tag` to identify the context menu (`axes_context_menu` in this example).

Adding Items to the Context Menu

Create the items on the menu using **New Menu Item** on the Menu Editor's toolbar.

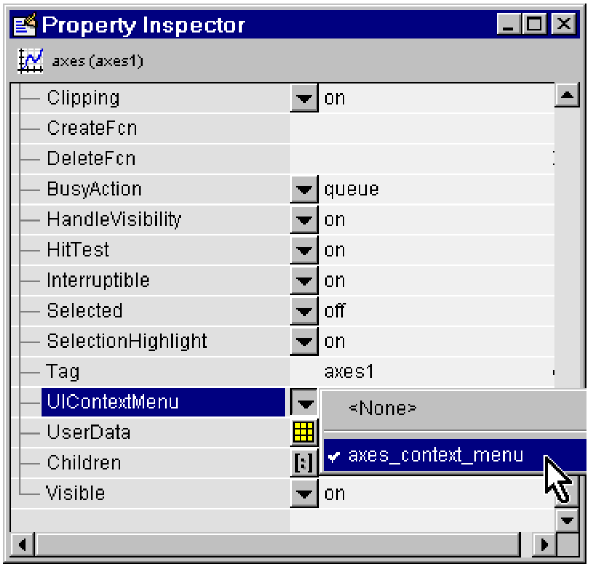


Next, assign a label and define the callback string.



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.



Add a callback routine subfunction to the application 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.

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 an overview of these components.

Push Buttons

Push buttons generate an action when pressed (e.g., an OK button may close a dialog box and apply settings). When you click down on a push button, it appears depressed; when you release the mouse, the button's appearance returns to its nondepressed state; and its callback executes on the button up event.

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 when you release the mouse button, at which point the callback executes. A subsequent mouse click returns the toggle button to the nondepressed state and again executes its callback.

The callback routine needs to query the toggle button to determine what state it is in. You can do this with a statement that uses the current callback object's handle (`gcbo`).

```
get(gcbo,'Value')
```

MATLAB sets the `Value` property to the value of the `Max` property when the toggle button is depressed (1 by default) and the value of the `Min` property when the toggle button is not depressed (0 by default).

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(h,'CData',a)
```

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.
- `Value = Min`, button is not selected.

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 application 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 deselected.

```
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. This structure is an input argument to all radio button callbacks.

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 varargout = radiobutton1_Callback(h,eventdata,handles,varargin)
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.

Checkboxes

Check boxes generate an action when clicked 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(h,eventdata,handles,varargin)
if (get(h,'Value') == get(h,'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.

Triggering Callback Execution

On UNIX systems, clicking on the menubar 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 menubar does not cause the editable text callback routine to execute. This behavior is consistent with the respective platform conventions. Clicking on other components in the GUI execute the callback.

Static Text

Static text controls displays 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

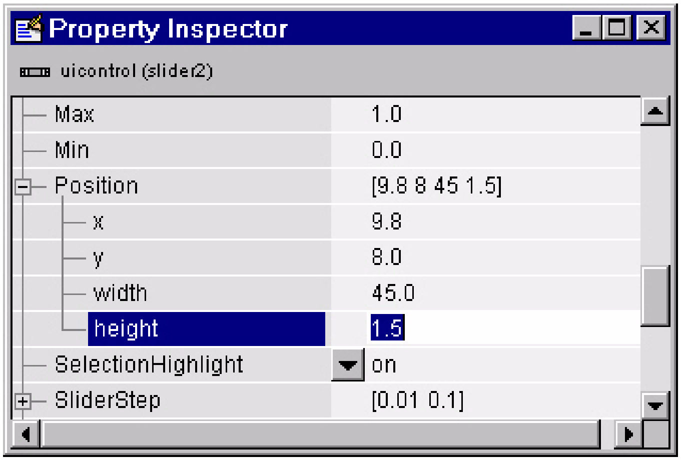
Sliders accept numeric input within a specific range by enabling the user to move a sliding bar. Users move the bar by pressing the mouse button and dragging the slide, by clicking in the trough, or by clicking an arrow. The location of the bar indicates a numeric value.

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.



Current Value, Range, and Stepsize

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 application 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)
```

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).

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 (defined using the `String` property) and enable users to select one or more items. The `Value` property contains the index into the list of strings that correspond to the selected item. If the user selected multiple items, then `Value` is a vector of indices. The first item in the list has an index of 1.

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, `[]`.

Single or Multiple Selection

The values of the `Min` and `Max` properties determine whether users can make single or multiple selections:

- If `Max - Min > 1`, then list boxes allow multiple item selection.
- If `Max - Min <= 1`, then list boxes do not allow multiple item selection.

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 for information on how to program multiple selection.

Triggering Callback Execution

MATLAB evaluates the list box's callback routine after any mouse button up or 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 application M-file option, you need to either:

- Set the list box `Callback` property to the empty string (`''`) and remove the callback subfunction from the application M-file.
- Leave the callback subfunction in the application M-file, but remove the default `disp` statement so that no code is executed 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 application 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 – 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 – shows how to access variables in the MATLAB base workspace from a list box GUI.

Popup Menus

Popup menus open to display a list of choices (defined using the `String` property) when users press the arrow. When not open, a popup 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. You can query the `Value` property in the callback routine to determine which choice the user made.

Popup 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.

Enable 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:

- First the figure `WindowButtonDownFcn` callback executes
- 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 output (e.g., plots and images). Axes are not uicontrol objects, but can be programmed to enable user interaction with the axes and graphics objects displayed in the axes. See the axes `ButtonDownFcn` property, which is also a property of all objects.

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.

Plotting Into the Hidden 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 illustrate 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 the steps taken.

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

Saving the GUI

The FIG-file that you create with the Layout Editor enables MATLAB to reconstruct your GUI when it is deployed. Generally, a functional GUI consists of two components:

- A FIG-file – containing a description of the GUI
- An M-file – containing the program that controls the GUI once it is deployed

FIG-Files

FIG-files (*filename*`.fig`) are binary files created as a result of saving a figure with the `hgsave` command or using **Save** from the Layout Editor's **File** menu. FIG-files replace the MAT-file/M-file combination that was previously used to save figures.

What Is In a FIG-File

A FIG-file contains a serialized figure object. That is, a complete description of the figure object and all of its children is saved in the file. This enables MATLAB to reconstruct the figure and all of its children when you open the file. All of the objects property values are set to the values they were saved with when the figure is recreated.

By default, FIG-files do not save the default figure toolbars and menus, although you can save these elements using the `hgsave` and `hgload` commands.

Handle Remapping

One of the most useful aspects of FIG-files is the fact that object handles saved, for example, in a `UserData` property are remapped to the newly created, equivalent object.

For example, suppose you have created a GUI that uses three radio buttons. Whenever a user selects one of the radio buttons, its callback routine must check the state of the other radio buttons and set them to `off` (as this is the standard behavior of radio buttons). To avoid having to search for the handles of the other radio buttons (with `findobj`), you could save these handles in a structure in the `UserData` property of each object.

When MATLAB reconstructs the figure and children (that is, deploys your GUI), the handles of the equivalent new objects are assigned to a structure with the same name and fields as in the original objects.

Opening FIG-Files

You can use the `open`, `openfig`, and `hgload` commands to open a file having a `.fig` extension. The application M-file uses `openfig` to display the GUI.

Programming GUIs

GUI Programming Topics

Graphical User Interfaces (GUIs) contain various user-interface components that enable software to communicate with an end user. One of the tasks of the GUI implementer is to control how the GUI responds to user actions. This section describes ways to approach the programming of the GUI.

- Understanding the Application M-File – The application M-file programs the GUI. This section describes the functioning of the application M-file generated by GUIDE.
- Managing GUI Data – The `handles` structure provides easy access to all component handles in the GUI. In addition, you can use this structure to store all global data required by your application M-file.
- Designing for Cross-Platform Compatibility – There are a settings (used by default with GUIDE) that enable you to make your GUI look better on multiple platforms. This section discusses these settings.
- Types of Callbacks – There are a number of callback properties besides the uicontrol `Callback`. This sections discusses the types available and their applications.
- Interrupting Executing Callbacks – The GUI programmer can decide if user actions can interrupt a callback that is currently executing. This section describes the process.
- Controlling GUI Figure Window Behavior – A GUI figure can block MATLAB execution and it can be modal. This section discusses options for GUI figures windows.

Understanding the Application M-File

MATLAB generates the application M-file to provide a framework for the program that controls the GUI. This framework fosters a programming style that is efficient and robust. All code, including the callbacks, is contained in the application M-file. Each callback is implemented as a subfunction in the M-file. This approach enables the M-file to have a single entry point that can initialize the GUI or can call the appropriate callback, or any helper subfunction you may want to use in your GUI.

Whether or not you use the GUIDE generated application M-file or create your own code, the programming techniques discussed here are useful approaches to GUI programming. The following sections discuss the architecture and functioning of the application M-file:

- Automatic Naming of Callback Routines
- Execution Paths in the Application M-File
- Initializing the GUI
- Managing GUI Data

Automatic Naming of Callback Routines

GUIDE automatically names the callback subfunctions it adds to the application M-file. It also sets the value of the `Callback` property to a string that causes this subfunction to execute when users activate the control.

Naming Callback Subfunctions

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

For example, the first push button you add to the layout is called `pushbutton1`. GUIDE adds a callback subfunction to the application M-file called `pushbutton1_Callback` when you save or activate the figure. If you define a `ButtonDownFcn` for the same push button, its subfunction is called `pushbutton1_ButtonDownFcn`.

Assigning the Callback 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 application M-file when you save or activate 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 application M-file.
- `pushbutton1_Callback` – is the name of the callback routine subfunction defined in `my_gui`.
- `gcbo` – is a command that returns the name 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 for more information on callback function arguments and Renaming Application Files and Tags for more information on how to change the names used by GUIDE.

Execution Paths in the Application M-File

The application M-file performs different actions depending on what arguments are passed to it when it is called. For example:

- Calling the M-file with no arguments launches the GUI (if you assign an output argument, the M-file returns the handle of the GUI figure).
- Calling the M-file with the name of a subfunction as the first argument executes that specific subfunction (typically, but not necessarily, a callback routine).

The application M-file contains a “switchyard” that enables it to switch to various execution paths depending on how it is called.

The Switchyard Code

The switchyard functionality in the application M-file is implemented using the `feval` function from within an `if` statement. `feval` evaluates (executes) the subfunction whose name is passed as a string argument when the application M-file is called. `feval` executes within a `try`/`catch` statement to catch errors caused by passing the name of nonexistent subfunctions. The following code generated by GUIDE implements the switchyard (you should not modify this code).

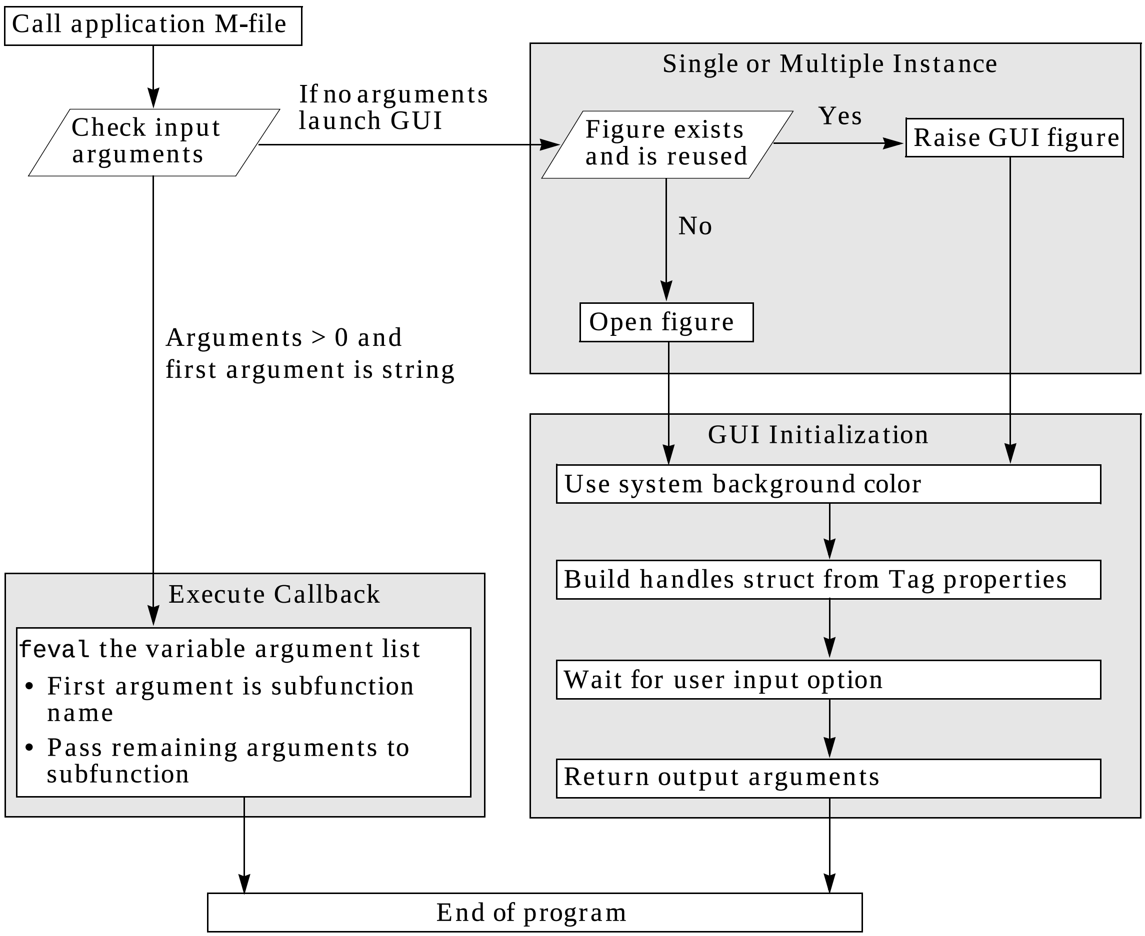
```
if nargin == 0 % If no arguments, open GUI
fig = openfig(mfilename,'reuse');
    .
    .
    .
elseif ischar(varargin{1}) % If string argument, call subfunction
    try
        [varargout{1:nargout}] = feval(varargin{:});
    catch
        disp(lasterr);
    end
end
```

Any output arguments returned by your subroutine are then returned though the main function. In addition, all input arguments specified in the `Callback` property string are included in the evaluated statement.

See Launching a Dialog to Confirm an Operation for an example that launches the GUI with zero or four arguments.

The following diagram illustrates the execution path for the application M-file.

Application M-File Execution Path



Adding Input Arguments to Subfunctions

Callback subfunctions added by GUIDE require certain arguments, but have a variable-length argument list. Since the last argument is varargin, you can add whatever arguments you want to the subfunction. To pass the additional arguments, edit the Callback property's string to include the arguments. For example, if the string added automatically to the Callback property is,

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

change it using the Property Inspector to include the additional arguments. For example,

```
my_gui('pushbutton1_Callback',gcbo,[],guidata(gcbo),arg1,arg2)
```

The subfunction, pushbutton1_Callback has the syntax

```
varargout = pushbutton1_Callback(h,eventdata,handles,varargin)
```

allowing a variable number of input arguments.

Initializing the GUI

The application M-file automatically includes some useful techniques for managing the GUI. These technique include:

- Single/multiple instance control.
- On screen placement of the GUI figure regardless of target computer screen size and resolution.
- Structure of GUI component handles created automatically.
- Automatic naming of Tag property, generation of subfunction prototype, and assignment of Callback property string.
- Single M-file contains code to launch GUI and execute callbacks.

Opening the FIG-File

The application M-file uses the openfig command to load the GUI figure. The actual command is

```
fig = openfig(mfilename,'reuse');
```

It is important to note that this statement derives the name of the FIG-file from the application M-file (the mfilename command returns the name of the currently executing M-file). If you are using the application M-file generated by

GUIDE, you must keep the names of the FIG-file and M-file the same. The `reuse` argument specifies that there can be only a single instance of the GUI displayed at any time (see below).

Single vs. Multiple Instance of the GUI

One of the decisions you must make when designing GUIs is whether you want to allow multiple instances of the GUI figure to exist at one time.

If you choose to allow only a single instance of the GUI, subsequent attempts to create another GUI figure simply bring the existing GUI to the front of other windows. Most informational dialogs (particularly if modal) should be created in singleton mode since it is not desirable to allow user actions to create more dialogs.

The GUIDE Layout Editor is an example of a GUI for which you can have multiple instances. This GUI was designed to enable users to have a number of layouts open simultaneously.

Positioning the GUI Onscreen

The application M-file uses the `movegui` command to ensure the GUI figure is visible on the screen of the target computer, regardless of the screen size and resolution. If the specified figure position would result in the GUI being placed off screen, `movegui` moves the figure to the nearest on-screen location with respect to the specified position.

The statement in the application M-file is

```
movegui(fig,'onscreen')
```

where `fig` is the handle of GUI figure returned by the `openfig` command.

`movegui` also provides other options for GUI placement.

Creating and Storing the Handles Structure

When you launch the GUI, the application M-file creates a structure that contains the handles of all the components in the GUI. It then saves this structure in the figure's application data so that it can be retrieved when needed (e.g., from a callback routine subfunction).

The name of the structure field containing a given object's handle is taken from the object's `Tag` property. For example, an object with a `Tag` value of `pushbutton1` is accessed with

```
handles.pushbutton1
```

You can access the figure’s hidden handle in a similar way. If the figure `Tag` is `figure1`, then

```
handles.figure1
```

is the figure’s handle.

The application M-files uses `guihandles` and `guidata` to create and store the structure.

```
handles = guihandles(fig); % Create handle struct
guidata(fig,handles); % Save struct in figure’s app data
```

Note that only those components whose `Tag` property is set to a string that is a valid variable name are included in this structure. Use `isvarname` to determine if a string is a valid name.

The `handles` structure is one of the arguments passed to each callback. You can also use this same structure to save data and pass it between subfunctions. See Managing GUI Data for a discussion of how to use the `handles` structure for other data.

Managing GUI Data

GUIDE provides a mechanism for storing and retrieving data using application-defined data stored on the GUI figure. GUIDE uses this mechanism to save a structure containing the handles of all the components in the GUI. Since this structure is passed to each callback subfunction, it is useful for saving other data as well, as is illustrated by the following example.

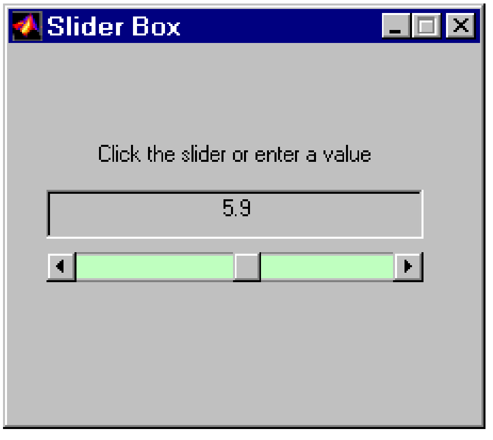
Passing Data in the Handles Structure

This example demonstrates how to use the `handles` structure to pass data between callback routines.

Suppose you want to create a GUI containing a slider and an editable text box that behave as follows:

- When a user moves the slider, the text box displays the new value.
- When a user types a value into the text box, the slider updates to this value.
- If they enter an out-of-range value in the text box, the application returns a message indicating how many erroneous values have been entered.

This picture shows the GUI with a static text field above the edit text box.



Defining the Data Fields During Initialization

The following excerpt from the GUI setup code show two additional fields defined in the `handles` structure – `errorString` and `numberOfErrors`:

- `guihandles` creates the structure and adds the handles of the slider and edit text using the `Tag` property to name the fields (`edit1` and `slider1`).
- `guidata` stores the structure in the figure's application data.

```
fig = openfig(mfilename,'reuse');
handles = guihandles(fig);
handles.errorString = 'Total number of errors: ';
handles.numberOfErrors = 0;
guidata(fig,handles);
```

Setting the Edit Text Value from the Slider Callback

Use the `handles` structure to obtain the handles of the edit text and the slider and then set the edit text `String` to the slider `Value`.

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

Note GUIDE-generated subfunctions take the `handles` structure as an argument. This eliminates the need to call `guidata` from within a subfunction to return the structure. However, if you make any changes to the `handles` structure, you must use `guidata` to save these changes.

Setting the Slider Value from the Edit Text Callback

The edit text callback routine sets the slider's value to the number the user types in, after checking to see if it is a single numeric value within the range of values allowed by the slider. If the value is out of range, then the error count is incremented and the error string and the error count are displayed.

```
val = str2num(get(handles.edit1,'String'));
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.numberOfErrors = handles.numberOfErrors+1;
    set(handles.edit1,'String',...
    [handles.errorString,num2str(handles.numberOfErrors)]);
    guidata(gcbo,handles); % store the changes...
end
```

Updating GUI Data. Note that you must use guidata to save the handles structure whenever you change values in that structure. The statement,

```
guidata(gcbo,handles);
```

makes use of the fact that guidata can determine the parent figure automatically from the handle of any child object (gcbo here). This is useful when you disable **Command-line accessibility** in the Application Options Dialog (the default); you cannot use findobj to obtain the figure handle.

If You Are Not Using a Handle Structure

If you are writing your own application M-file and are not generating a handle structure, you can still use the GUI figure's application data for storing any data that you want to pass between subfunctions. This mechanism involves:

- Creating a structure containing the data you want to store.
- Storing the structure in the figure's application data.
- Retrieving the structure within the subfunction when it is required.

Using the guidata Function

The guidata function provides a convenient interface to the figure's application data. It enables you to access the data without having to find the figure's handle (something that may be difficult when the handle is hidden) and avoids the need to create and maintain a hard-coded property name for the application data throughout your source code.

For example, you would set up the code similar to this.

In the initialization code:

```
fig = openfig(mfilename,'new'); % open GUI and save figue handle
                  .
                  .
                  .
```

```
data.field1 = value1; % create a structure
guidata(fig,data)     % save the structure
```

Within a callback subfunction:

```
data = guidata(gcbo);     % load the data
data.field1 = new_value; % change the structure
guidata(gcbo,data)        % save the structure
```

Note that, once a callback routine has begun execution, `guidata` can obtain the handle of the figure using `gcbo` (the handle of the object whose callback has been called). However, in the initialization section, no callback routine has been invoked so you cannot use `gcbo`. In this case, you can use the handle of the GUI figure returned by `openfig`.

Application-Defined Data

Application-defined 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.

Note that `guidata` provides an easy to use interface to application data. When using the GUIDE-generated application M-file, it is simpler to use `guidata` than to access application data directly. See Managing GUI Data for more information.

Functions for Accessing Application Data

The following functions provide access to application-defined data.

Functions for Accessing Application-Defined Data

Command	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 System Font

By default, uicontrols use the default font for the platform on which it is running. For example, when displaying your GUI on PCs, it 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 MATLAB uses the system default at run time.

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 must look correct on computers having different size screens and different 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.

System-Dependent Units

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

Setting figure `Units` to `characters` enables you to develop a GUI that automatically adjusts the relative spacing and size of components as the GUI displays on different computers. For example, if the size of the text label on a component becomes larger because the system font metrics are different, then the component and the relative spacing increase proportionally.

GUIDE sets the figure `Units` to `characters` by default.

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 when 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 whena 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 when the cursor is in 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 when the cursor is in 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 in 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, a figure, or other graphics objects having callbacks defined, MATLAB first executes the `WindowButtonDownFcn` of the figure (if defined) and then `ButtonDownFcn` of the object targeted by the mouse click.

Interrupting Executing Callbacks

By default, MATLAB allows an executing callback to be interrupted by subsequently envoked 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 MATLAB’s state before returning to the original callback.

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`, `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 actually 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 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 remaining events are processed.

Controlling GUI Figure Window Behavior

When designing a GUI you need to consider how you want the figure window to behave once it is displayed. The appropriate behavior for a particular GUI figure 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 requiring 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.
- A dialog 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 figure is both blocking and modal.

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

- Allow a single or multiple instances of the GUI at any one time.
- Block MATLAB execution while the GUI is displayed.
- 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 GUI Figure Modal

Set the GUI 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 application M-file, using the handle returned by `openfig` with the `set` command.

```
set(fig,'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(`*figure_handle*`)`
- Make the figure invisible.

  `set(`*figure_handle*`,'Visible','off')`
- Change the figure's `WindowStyle` property to `normal`.

  `set(`*figure_handle*`,'WindowStyle','normal')`

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

Obtaining the Figure Handle from Within a Callback. In general, dismissing a modal figure requires the handle of the figure. Since most GUIs hide figure handles to prevent accidental access, the `gcbf` (get callback figure) command provides the most effective method to get the figure handle from within a callback routine.

`gcbf` returns the handle of the figure containing the object whose callback is executing. This enables you to use `gcbf` in the callback of the component that will dismiss the dialog. For example, suppose your dialog includes a push button (tagged `pushbutton1`) that closes the dialog. Its callback could include a call to `delete` at the end of its callback subfunction.

```
function varargout = pushbutton1_Callback(h,eventdata,handles,varargin)
% Execute code according to dialog design
           .
           .
           .
% Delete figure after user responds to dialog
delete(gcbf)
```

4

Application Techniques

Example Applications

This section contains a series of examples that illustrate techniques that are useful for implemented GUIs. Each example provides a link to the actual GUI in the GUIDE Layout Editor and a link to the application M-file displayed in the MATLAB editor.

- Launching a Dialog to Confirm an Operation – display a second figure from your GUI to ask a question. This modal dialog blocks MATLAB until the user responds, then returns the answer to the calling GUI.
- List Box Directory Reader – list the contents of a directory, navigate to other directories, and define what command to execute when users double-click on a given type of file.
- Accessing Workspace Variables from a List Box – list variables in the base MATLAB workspace from a GUI and plot them. This example illustrates multiple item selection and executing commands in a different workspace.
- A GUI to Set Simulink Model Parameters – set parameters in a Simulink model, save and plot the data, and implement a help button.
- An Address Book Reader – read data from MAT-files, edit and save the data, manage GUI data using the `handles` structure.

Launching a Dialog to Confirm an Operation

This example illustrates how to display a dialog when users attempt to close a GUI. The purpose of the dialog is to force the user to confirm that they really want to proceed with the close operation.

Dialog Requirements

You want to protect users from unintentionally closing a GUI application by displaying a confirmation dialog when they press the **Close** button on the main application window.

You want the dialog to:

- Be launched by the application's **Close** button.
- Ask the user to confirm the close operation (i.e., respond yes or no).
- Block MATLAB execution until the user responds.
- Be modal to maintain focus since there is a pending operation.
- Handle the case where the user closes the dialog from the window manager close box without responding.

The following sections discuss the implementation of this dialog.

View the Layout and Application M-File

Use these links to display the FIG-file in the Layout Editor and the application M-file in the MATLAB Editor. This enables you to see the values of all component properties and to explore how the components are assembled to create the GUI. You can also see the complete code listing.

Note The following link adds a directory to the end of your MATLAB path.

Click here to display the layout for this GUI in the GUIDE Layout Editor.

Click here to load the application M-file into the MATLAB Editor/Debugger.

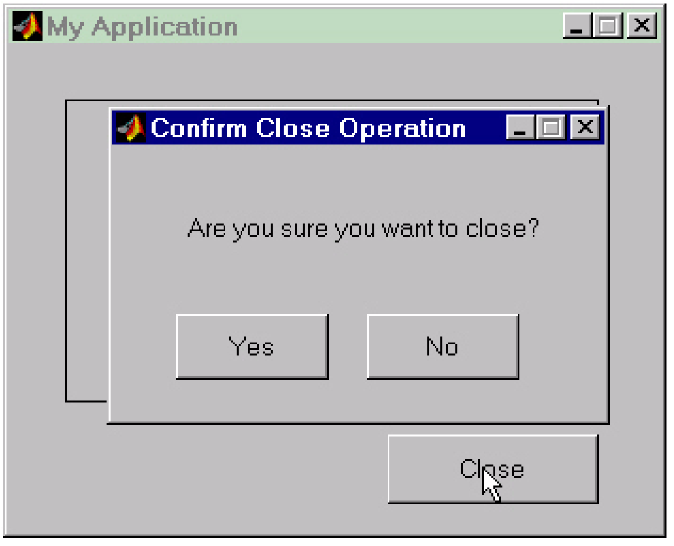
Implementing the GUI

This section describes the steps followed to implement the requirements described in the previous section.

Launching the Dialog

The **Close** button's callback needs to launch a dialog that asks for confirmation of the impending close operation. This callback must then wait for a dialog to return a value. To accomplish this, the dialog's application M-file assigns an output argument and the **Close** button's callback waits for it to be returned.

The following picture illustrates the dialog positioned over the GUI figure.

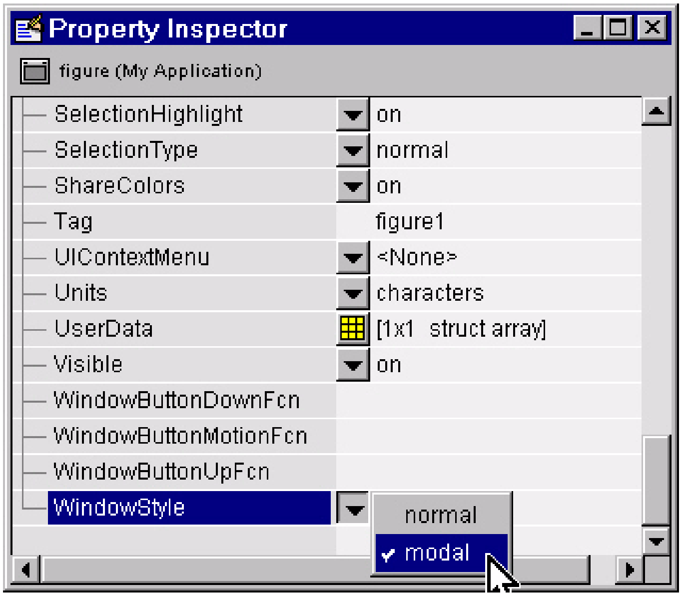


Wait for User Input

To make the dialog wait for user input, select **Function does not return until application window dismissed** in the GUIDE Application Options dialog. This option adds a call to `uiwait` in the dialog's application M-file.

Making the Dialog Figure Modal

To make the dialog figure modal, select the figure in the Layout Editor and then right-click to display the context menu. Select **Property Inspector** from the context menu. Use the Property Inspector to set the figure's `WindowStyle` property to `modal`.



Sequence Following a Close Button Press

The following sequence occurs when the user presses the **Close** button on the GUI application figure:

1. User clicks **Close** button – callback calls the M-file to launch the confirmation dialog and waits for a returned value.

2. Confirmation dialog M-file executes and waits for user to take one of three possible actions: click the `Yes` push button, the `No` push button, or the close box (X) on the window border. All other interactions with MATLAB are blocked.

3. Confirmation dialog callbacks resume M-file execution, causing output the value to be returned to the **Close** button callback.

4. **Close** button callback resumes execution and takes appropriate action based on user response to the confirmation dialog.

The Close Button Callback

The **Close** button callback performs the following steps:

- Determines the location of the confirmation dialog based on the current size and location of the GUI application figure.
- Calls the M-file that launches the confirmation dialog with an input argument that specifies the dialog location and an output argument that causes the callback to wait for the dialog to return the user response.
- Takes the appropriate action depending on the answer returned from dialog.

Here is the **Close** button callback:

```
function varargout = pushbutton1_Callback(h, eventdata, handles, varargin)
    pos_size = get(handles.figure1,'Position');
    dlg_pos = [pos_size(1)+pos_size(3)/5 pos_size(2)+pos_size(4)/5];
    user_response = modaldlg(dlg_pos);
    switch user_response
        case {'no','cancel'}
            return
        case 'yes'
            % Prepare to close GUI application figure
        %                 .
        %                 .
        %                 .
            delete(handles.figure1)
    end
```

The Confirmation Dialog Application M-file

The confirmation dialog has its own application M-file, which the main application calls to launch the dialog. This M-file can be called in three ways:

- No arguments – launch the dialog and wait for user input.
- One numeric argument – launch the dialog and place it at the location specified in the two-element vector.
- Four arguments – call the **Yes** or **No** button callback with the usual arguments (`h,eventdata,handles,varargin`).

With each calling syntax, the M-file returns a string output argument indicating the user response.

The application M-file performs various operations, which are described in the following sections:

- Launch the dialog
- Specify the location of the dialog
- Wait for user response
- Execute a callback
- Define the **Yes** button and **No** button callbacks

Launch the Dialog

This section of the application M-file launches the dialog if the number of input arguments is zero or one numeric value. This involves:

- Checking for the correct number of input arguments (callbacks have 4 arguments).
- Using `openfig` to load the FIG-file.
- Setting the figure color to the standard GUI color on the host system.
- Creating the `handles` structure.

Note that the function returns one output argument, `answer`, which is passed to the **Close** button callback.

```
function answer = modaldlg(varargin)
error(nargchk(0,4,nargin)) % function takes 0, 1, or 4 arguments
if nargin == 0 | isnumeric(varargin{1})
    fig = openfig(mfilename,'reuse');
    set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));
    handles = guihandles(fig);
    guidata(fig, handles);
```

Specify the Location of the Dialog

The dialog application M-file accepts an input argument that specifies where to display the dialog. This enables the **Close** button callback to locate the dialog with respect to the main application window. The argument is a two-element vector containing the left and bottom offsets from the right and lower edge of the screen, in character units. The **Close** button callback determines these values.

Preventing Figure Flash

In some cases, repositioning the dialog's figure may cause it to "flash" on the screen in its current position before the set command repositions it. To prevent this effect, save the dialog with it's figure Visible property set to off. You can then set the Visible property to on after specifying the position. Note that you must specify the Position property before setting visibility to on.

```
if nargin == 1
    pos_size = get(fig,'Position');
    pos = varargin{1};
    if length(pos) ~= 2
        error('Input argument must be a 2-element vector')
    end
    new_pos = [pos(1) pos(2) pos_size(3) pos_size(4)];
    set(fig,'Position',new_pos,'Visible','on')
end
```

Wait for User Response

uiwait causes modaldlg to wait before returning execution to the **Close** button callback. During this time, the dialog's callbacks can execute in response to user action.

uiwait waits until the dialog figure is deleted or a uiresume executes. This can be caused when:

- The user clicks the X in the close box on the window border. If this happens, uiwait returns. Since the handle stored in the variable fig no longer corresponds to a figure, modaldlg uses an ishandle test to return 'cancel' to the **Close** button callback.
- The **Yes** button callback executes uiresume after setting handles.answer to 'yes'.
- The **No** button callback executes uiresume after setting handles.answer to 'no'.

```
uiwait(fig);
if ~ishandle(fig)
    answer = 'cancel';
else
    handles = guidata(fig);
    answer = handles.answer;
```

```
        delete(fig);
    end
```

Executing a Callback

This is the feval switchyard that enables modaldlg to execute the callback subfunctions. It relies on the fact that when modaldlg is called to execute a callback, the first argument is a string (the name of the callback).

```
elseif ischar(varargin{1}) % Invoke named subfunction or callback
    try
        [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
    catch
        disp(lasterr);
    end
end
```

Defining the Yes and No Buttons Callbacks

The callbacks for the **Yes** and **No** buttons perform the same basic steps:

- Assign the user response in the handles structure answer field.
- Use guidata to save the modified handles structure, which is then read by the main function.
- Use uiresume to continue the blocked code in the main function.

The Tag property of each push button uicontrol was changed before saving the application M-file so that the callback function names are more descriptive. The following code illustrates the implementation of the callbacks.

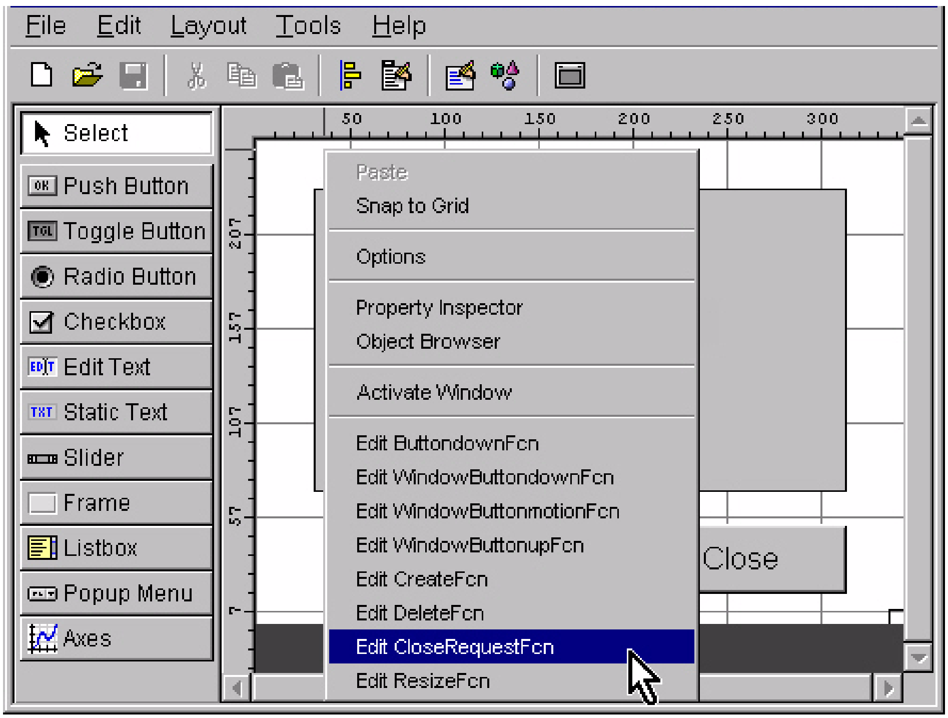
```
function varargout = noButton_Callback(h, eventdata, handles, varargin)
handles.answer = 'no';
guidata(h, handles);
uiresume(handles.figure1);


function varargout = yesButton_Callback(h, eventdata, handles, varargin)
handles.answer = 'yes';
guidata(h,handles);
uiresume(handles.figure1);
```

Protecting the GUI with a Close Request Function

Whenever a user closes a figure, MATLAB first executes the figure's close request function, as defined by the `CloseRequestFcn` figure property. The default close request function simply deletes the figure. However, a GUI may want to protect the user from unintentionally deleting a figure if they click on the `X` in the close box of the window border.You can change the default close request function by redefining the figure's `CloseRequesFcn`.

The "The Close Button Callback" section shows a callback for the GUI's **Close** button that you could also use as a close request function. To add the new close request function to your application M-file, select the figure in the Layout Editor and right click to display the context menu.



Select **Edit CloseRequestFcn** from the context menu. GUIDE automatically places a new subfunction in the application M-file for the GUI and changes the figure's CloseRequesFcn property to execute this subfunction as the close request function.

The Redefined Close Request Function

The GUIs close request function simply calls the Close button's callback.

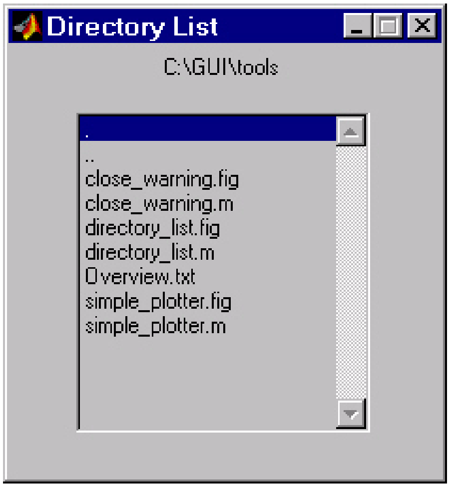
```
function varargout = figure1_CloseRequestFcn(h,eventdata,handles,varargin)
pushbutton1_Callback(h,eventdata,handles)
```

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 a double dot (..), the GUI changes to the directory up one level and populates the list box with the contents of that directory.

The following picture illustrates the GUI.



View the Layout and Application M-File

Use these links to display the FIG-file in the Layout Editor and the application M-file in the MATLAB Editor. This enables you to see the values of all component properties and to explore how the components are assembled to create the GUI. You can also see the complete code listing.

Note The following link adds a directory to the end of your MATLAB path.

Click here to display the layout in GUIDE.

Click here to display the application 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 launched.
- 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 full pathname as a string input argument. If you do not specify a directory (i.e., if you call the application M-file with no input arguments), the GUI then uses MATLAB's current directory.

As generated, the application M-file launches the GUI when there are no input arguments and calls 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 – launch the GUI using MATLAB's current directory.
- First input argument is a valid directory – launch 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 entire initialization section of the application M-file. The statements in bold are the additions made to the generated code:

```
function varargout = lbox2(varargin)
if nargin <= 1   % LAUNCH GUI
    if nargin == 0
        initial_dir = pwd;
    elseif nargin == 1 & exist(varargin{1},'dir')
        initial_dir = varargin{1};
    else
        errordlg('Input argument must be a valid directory',...
            'Input Argument Error!')
        return
    end
    fig = openfig(mfilename,'reuse');
    % Use system color scheme for figure:
    set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));
    % Generate a structure of handles to pass to callbacks, and store it.
    handles = guihandles(fig);
    guidata(fig, handles);
    % Populate the listbox
    load_listbox(varargin{1},handles)
    if nargout > 0
        varargout{1} = fig;
    end
elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK
    try
        [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
    catch
        disp(lasterr);
    end
end
```

Loading the List Box

This example uses a separate 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 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 `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 in the initialization section of the application M-file as well as 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 the contents.

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. An error dialog captures any errors that occur when opening a file, instead of displaying them on the command line.

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.

- 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 enable errors to be returned in an error dialog (errordlg), instead of at the command line.

```
function varargout = listbox1_Callback(h,eventdata,handles,varargin)
if strcmp(get(handles.figure1,'SelectionType'),'open')
    index_selected = get(handles.listbox1,'Value');
    file_list = get(handles.listbox1,'String');
```

```
    filename = file_list{index_selected};
    if  handles.is_dir(handles.sorted_index(index_selected))
        cd (filename)
        load_listbox(pwd,handles)
    else
        [path,name,ext,ver] = fileparts(filename);
        switch ext
        case '.fig'
            guide (filename)
        otherwise
            try
                    open(filename)
            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 application M-file. See open for more information.

Accessing Workspace Variables from a List Box

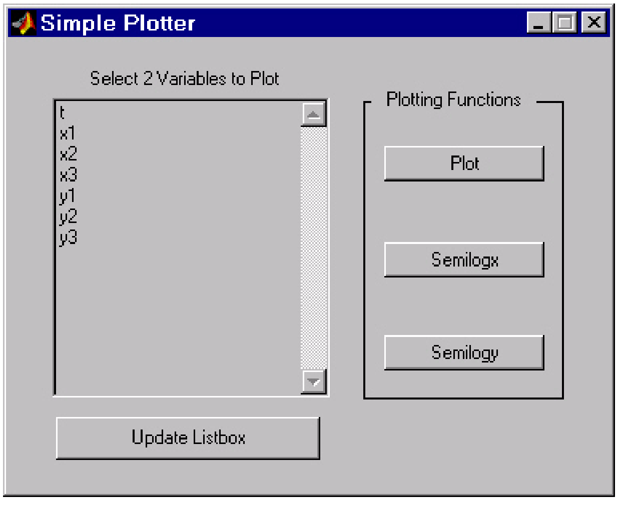
This GUI uses a list box to display workspace variables, which the user can then plot.

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 application M-file.

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

View the Layout and Application M-File

Use these links to display the FIG-file in the Layout Editor and the application M-file in the MATLAB Editor. This enables you to see the values of all component properties and to explore how the components are assembled to create the GUI. You can also see the complete code listing.

Note The following link adds a directory to the end of your MATLAB path.

Click here to display the layout in GUIDE.

Click here to display the application 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. The following subfunction added to the application 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 application 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`, `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.

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 two items.

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 variable, 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:

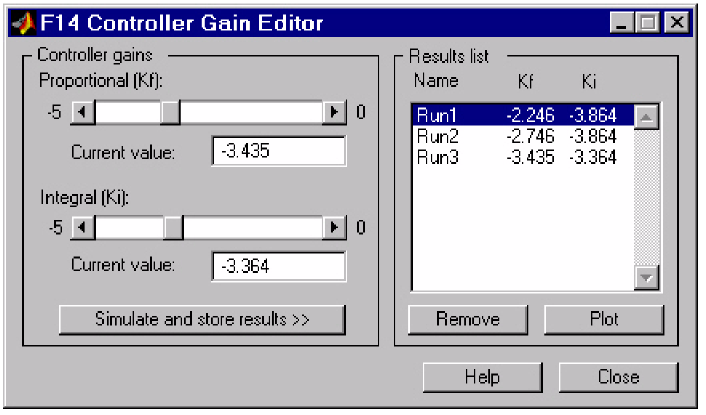
```
function varargout = plot_button_Callback(h,eventdata,handles,varargin)
[x,y] = get_var_names(handles);
evalin('base',['plot(' x ',' y ')'])
```

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.



Techniques Used in This Example

This example illustrate 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 global data using the `handles` structure.
- Directing graphics output to a figures with hidden handles.
- Adding a help button that displays `.html` files in the MATLAB Help Browser.

View the Layout and Application M-File

Use these links to display the FIG-file in the Layout Editor and the application M-file in the MATLAB Editor. This enables you to see the values of all component properties and to explore how the components are assembled to create the GUI. You can also see the complete code listing.

Note The following link adds a directory to the end of your MATLAB path.

Click here to display the layout in GUIDE.

Click here to display the application 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 pressing 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.

Launching the GUI

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

Application Options Settings

This GUI uses the following application option settings:

- Resize behavior: Non-resizable
- Command-line accessibility: Off
- Application M-file options selected:

  Generate callback function prototypes

  Application allows only one instance to run

  Use system color scheme for background

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 application M-file launches 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 the gain value can be displayed (`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.

This 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).

The following code lists the callback for the **Proportional (Kf)** slider.

```
function varargout = KfValueSlider_Callback(h,eventdata,handles,varargin)
% Ensure model is open
model_open(handles)
% Get the new value for the Kf Gain from the slider
NewVal = get(h,'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 simulation parameters can be set.
- 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).

The following code lists the callback for the Kf **Current value** text box.

```
function varargout = KfCurrentValue_Callback(h,eventdata,handles,varargin)
% Ensure model is open
```

```
model_open(handles)
% Get the new value for the Kf Gain
NewStrVal = get(h,'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(h,'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.

The following code lists the **Simulate and store results** button callback.

```
function varargout = SimulateButton_Callback(h,eventdata,handles,varargin)
[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(h,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`).

The following code is the **Remove** button callback.

```
function varargout = RemoveButton_Callback(h, eventdata, handles, varargin)
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(h,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 in use 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 once the plot and legend have been created. To prevent this figure from becoming the target of 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

The following code is the **Plot** button callback.

```
function varargout = PlotButton_Callback(h, eventdata, handles, varargin)
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(h,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.

The following code is the **Help** button callback.

```
function varargout = HelpButton_Callback(h,eventdata,handles,varargin)
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

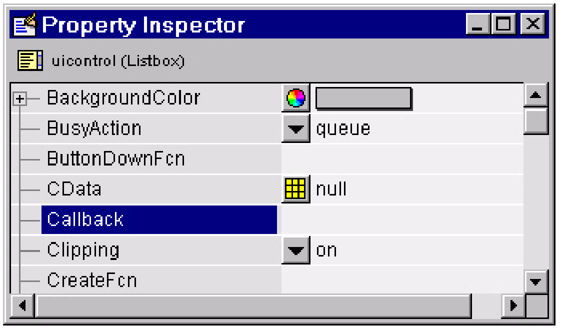
The following code is the **Close** button callback.

```
function varargout = CloseButton_Callback(h,eventdata,handles,varargin)
% 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

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).

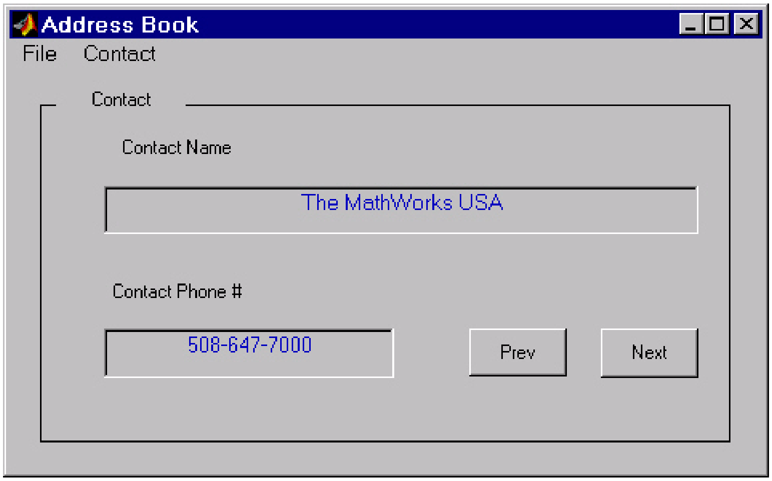
In this case, there is no need for the list box callback to execute, so it is deleted from the application M-file. It is important to remember to also delete the Callback property string so MATLAB does not attempt to execute the callback. You can do this using the property inspector:



See the description of list box triggering for more information.

An Address Book Reader

This example implements a GUI that displays names and phone numbers, which it reads from a MAT-file. You can add new entries, which are then saved to the same MAT-file or to a new one.



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 global data.
- Uses a GUI figure resize function.

View the Layout and Application M-File

Use these links to display the FIG-file in the Layout Editor and the application M-file in the MATLAB Editor. This enables you to see the values of all component properties and to explore how the components are assembled to create the GUI. You can also see the complete code listing.

Note The following link adds a directory to the end of your MATLAB path.

Click here to display the layout in GUIDE.

Click here to display the application M-file in the editor.

Managing Global 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 in the MAT-file
- An index pointer that indicates the currently name and phone number, which must be updated as the user pages through the address book.
- The figure position and size
- The handle of all GUI components

The descriptions of the subfunctions that follow illustrate how to save and retrieve information from the `handles` structure. See Managing GUI Data for background information on this structure.

Launching the GUI

The GUI is nonblocking and nonmodal since it is designed to be displayed while you perform other MATLAB tasks.

Application Options Settings

This GUI uses the following application option settings:

- Resize behavior: User-specified
- Command-line accessibility: Off
- Application M-file options selected:

  Generate callback function prototypes

  Application allows only one instance to run

Launching the GUI

You can call the application M-file with no arguments, in which case, the GUI uses the default address book MAT-file, or you can specify a MAT-file as an argument. Specifying a MAT-file as an input argument requires modification to the default GUI initialization section of the application M-file. The following code shows these changes in bold.

```
function varargout = address_book(varargin)
if nargin <= 1  % LAUNCH GUI
    fig = openfig(mfilename,'reuse');
    set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));
    handles = guihandles(fig);
    guidata(fig, handles);
    if nargin == 0
        % Load the default address book
        Check_And_Load([],handles)
    elseif exist(varargin{1},'file')
        Check_And_Load(varargin{1},handles)
    else
        % If the file does not exist, return an error dialog
        % and set the text to empty strings
        errordlg('File Not Found','File Load Error')
        set(handles.Contact_Name,'String','')
        set(handles.Contact_Phone,'String','')
    end
if nargout > 0
    varargout{1} = fig;
  end
elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK
  try
    [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
  catch
    disp(lasterr);
  end
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 launching 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

Here is the listing for Check_And_Load.

```
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
% fileds 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 varargout = Open_Callback(h, eventdata, handles, varargin)
[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(h,handles)
    end
end
```

See the Menu Editor 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 launched.

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.

Code Listing

Here is the listing for Contact_Name_Callback.

```
function varargout = Contact_Name_Callback(h, eventdata, handles, varargin)
% 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(h,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(h,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).Phone)
      return
end
```

The Contact Phone # 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 and press enter, the callback launches 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.

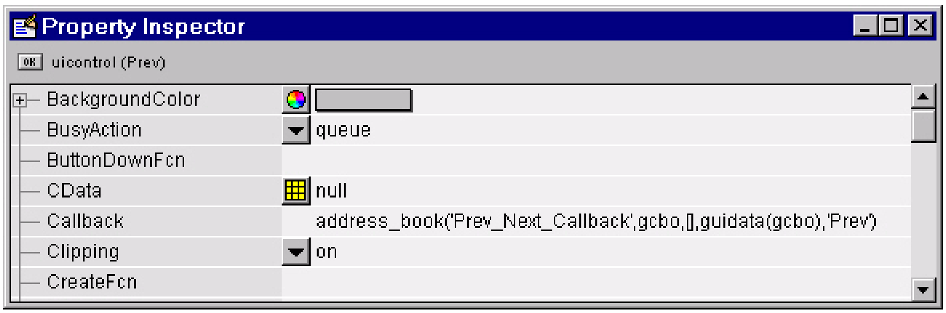
Code Listing

```
function varargout = Contact_Phone_Callback(h, eventdata, handles, varargin)
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(h,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:



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 Foward 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 varargout = Prev_Next_Callback(h,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(h,handles)
```

Saving Changes to the Address Book from the Menu

When you make changes to an address book, you need to save the 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.

The `handles` structure contains the structure to 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` updated `handles.Addresses`.

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.

Code Listing

```
function varargout = Save_Callback(h, eventdata, handles, varargin)
% Get the Tag of the menu selected
Tag = get(h,'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(h,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 varargout = New_Callback(h, eventdata, handles, varargin)
set(handles.Contact_Name,'String','')
set(handles.Contact_Phone,'String','')
```

The Address Book Resize Function

The address book defines it's own resize function. This requires the application option dialog **Resize behavior** to be set 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 application 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 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, the figure is set to the new width. The size of the **Contact Name** text box changes in proportion to the 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, it is set to the original width.

Changing the Height

If the user attempts to change the height, the height is set to 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 **–** 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 release the mouse.

When the GUI is first launched, 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 varargout = ResizeFcn(h, eventdata, handles, varargin)
% Get the figure size and position
Figure_Size = get(h,'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(h,'Position',...
    [Figure_Size(1) Figure_Size(2) Original_Size(3) Original_Size(4)])
           Figure_Size = get(h,'Position');
      end
      if Figure_Size(4) ~= Original_Size(4)
           % Do not allow the height to change
           set(h,'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(h,'onscreen')
```

Index

S

T

U