MATLAB®
Runtime Server

The Language of Technical Computing

Computation

Visualization

Programming

Application Developer's Guide

Version 6

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

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

MATLAB Runtime Server Application Developer's Guide

Printing History:			
	August 1997	First printing	New for MATLAB 5.1
	May 1998	Second printing	Revised for MATLAB 5.2
	January 1999	Third printing	Revised for MATLAB 5.3 (Release 11)
	September 2000	Fourth printing	Revised for MATLAB 6.0 (Release 12)
	June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
	July 2002	Online only	Revised for MATLAB 6.1.1 (Release 13)

Contents

Preface

1 Design Issues for a Runtime Application

2 Developing a MATLAB Runtime GUI Application

3 Developing a MATLAB Runtime Engine Application

4 Shipping a MATLAB Runtime Application

5 Function Reference

Preface

What Is the MATLAB Runtime Server?

The MATLAB Runtime Server is a variant of MATLAB® that software developers can ship together with an application that uses MATLAB. The MATLAB Runtime Server contains all of the computational and graphical capabilities of commercial MATLAB, but is designed to run stand-alone applications that are based on MATLAB. End users of Runtime Server applications do not need to own MATLAB and do not need any specific knowledge about MATLAB. They cannot access the source code of Runtime Server applications.

Key Features of the MATLAB Runtime Server

The key features that distinguish the Runtime Server from commercial MATLAB are:

- On startup, the Runtime Server shows a developer-designated splash screen instead of the standard MATLAB splash screen.
- The MATLAB command window is *not* available for end users of Runtime Server applications. This means that when you develop an application for use with the MATLAB Runtime Server, you must supply a graphical user interface (GUI) or other interface for the end user.
- The Runtime Server recognizes neither M-files nor standard P-files. It can execute only built-in MATLAB functions, MEX-files and *runtime P-files*.

  Runtime P-files are generated by using the `buildp` function, described in "Compiling the Application with One Command" on page 2-8. The section "Configuration Information" on page x discusses the difference between runtime P-files and standard P-files.
- On startup, the Runtime Server executes `matlabrt.p` instead of `matlabrc.m`.

Overview of MATLAB Runtime Applications

It is easy to adapt an application based on MATLAB so that it will run with the Runtime Server. The instructions in this *Application Developer's Guide* assume that you already have a working application that uses MATLAB and that you want the Runtime Server to execute. If you are still planning and building your MATLAB based application, then the instructions in this guide will still be useful because they can help shape your design process.

The MATLAB Runtime Server can perform two categories of tasks:

- Run an entire application by executing MEX-files and runtime P-files. This type of runtime application is called a *MATLAB runtime GUI application*. Its front end is usually a MATLAB GUI. For information about developing GUI-based applications in MATLAB, see the MATLAB documentation set.
- Act as the *computational engine* for an application that is dependent on MATLAB and that is partially written in another language. In this case, the application's front end is developed in a language such as Visual Basic, and MATLAB is incorporated as part of the application's back end. After its adaptation for use with the Runtime Server, this type of application is called a *MATLAB runtime engine application*. For information about using MATLAB as a computational engine, see the MATLAB documentation set.

These two types of MATLAB runtime applications share some common features, but also differ in several important ways. The next section explains how to use this book for the type of runtime application you want to develop.

Note Because the MATLAB command window is inactive in the runtime variant, MATLAB runtime applications *must* provide their own user interfaces for the end users. For example, these front-end user interfaces might be GUIs created with MATLAB Handle Graphics® or with other visual development tools.

Note The MATLAB Runtime Server supports the Engine Application Program Interface (API) Library, pipes on UNIX, and ActiveX on PC. It *does not* support dynamic data exchange (DDE), the MATLAB Notebook, or Simulink®.

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the MATLAB Runtime Server.

For more information about any of these products, see either:

- The online documentation for that product, if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section.

Product	Description
Database Toolbox	Exchange data with relational databases
Financial Time Series Toolbox	Analyze and manage financial time series data
GARCH Toolbox	Analyze financial volatility using univariate GARCH models
MATLAB Compiler	Convert MATLAB M-files to C and C++ code
MATLAB Web Server	Use MATLAB with HTML Web applications

Using This Guide

This guide includes instructions for developing both MATLAB runtime GUI applications and MATLAB runtime engine applications. Therefore, depending on which type of runtime application you are developing, certain sections might not apply to your project. The table below provides some suggestions for which sections you should read, and which you might be able to skip.

For a...	Do this:
MATLAB runtime GUI application	• Read Chapter 2, "Developing a MATLAB Runtime GUI Application." • *Skip* Developing a MATLAB Runtime Engine Application. • Read Chapter 4, "Shipping a MATLAB Runtime Application."
MATLAB runtime engine application	• Read most of Chapter 2, "Developing a MATLAB Runtime GUI Application" but *skip* the example in the section "Matlab Runtime GUI Application." • Read Chapter 3, "Developing a MATLAB Runtime Engine Application." • Read Chapter 4, "Shipping a MATLAB Runtime Application."

Configuration Information

Before you use the Runtime Server, you must install it and also stamp it with a password that you choose. First follow the instructions in the *MATLAB Installation Guide* for your platform. Your `toolbox` directory now contains a `runtime` subdirectory. This subdirectory contains tools for developing and debugging MATLAB runtime applications, along with some sample files.

To stamp MATLAB with your password, follow these instructions:

1. Go to the system prompt.
2. If `matlabroot` is the directory in which you installed MATLAB, then navigate to

   `matlabroot\toolbox\runtime\bin\win32` **(PC)**

   `matlabroot/toolbox/runtime` **(UNIX)**
3. At the system prompt, type

   `rtsetup -f matlabroot\bin\win32\matlab.exe -s string` **(PC)**

   `rtsetup` **(UNIX)**

   where `string` is a password that you choose. On UNIX platforms, `rtsetup` is an interactive script that prompts you for the password; for more details, see the text file `matlabroot/toolbox/runtime/README`.

   The utility `rtsetup` stamps your copy of MATLAB with this password. The password can be up to 32 characters long and may include spaces.

Note You cannot restamp a previously stamped copy of MATLAB.

If you anticipate creating multiple runtime applications, each with a different password, then you might want to preserve an unstamped copy of `matlabroot\bin\win32\matlab.exe` (PC) or `matlabroot/bin/`*`arch`*`/matlab` (UNIX, where *`arch`* is a directory name specific to your architecture). Then whenever you need to change the password, copy the unstamped file into its proper place and run `rtsetup`. Alternatively, you can reinstall MATLAB and run `rtsetup` each time you want to change the password.

Password Consistency Rules

Your runtime application will work properly on the end-user's machine *only if* you follow these consistency rules:

Rule 1. Stamp your development copy of MATLAB *before* converting to P-files any of the files that you will be testing or shipping with the Runtime Server. If you or others on your development team use several copies of MATLAB for a single runtime application, then stamp all of them with the *same* password. The stamping procedure was described in "Configuration Information".

Rule 2. When you generate runtime P-files for use with the Runtime Server, generate them using a stamped development copy that has the same password mentioned in Rule 1. The generation of runtime P-files is described in "Compiling the Application (GUI)" in Chapter 2.

Rule 3. When you duplicate MATLAB files for shipping, duplicate those of a stamped development copy that has the same password mentioned in Rule 1. Duplication of files for shipping is described in "Automatically Packaging Files for Shipping" in Chapter 4.

Technical Conventions

In this *Application Developer's Guide*, an application based on MATLAB that you create, or adapt, for use with the Runtime Server is called a *MATLAB runtime application*. The section "Overview of MATLAB Runtime Applications" on page vi introduces the two types of MATLAB runtime applications.

The commercial MATLAB that you use to develop the application is called your *development copy of MATLAB* or just MATLAB. The subset of your development copy of MATLAB that you ship as part of your MATLAB runtime application is called the *runtime variant* or the *shipping variant*.

The executable file named either `matlab.exe` on PC platforms or `matlab` on UNIX platforms is called the *MATLAB executable*.

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, user input, items in drop-down lists	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'whole')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace` font	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'method')`

1

Design Issues for a Runtime Application

Preventing Command Window Input/Output

The MATLAB Runtime Server does not provide a command-line interface to the end user, so your application should not depend on the command window. To eliminate dependence on the command window, you should:

- Make certain that your application does not require input via the command window, and does not direct output to the command window.
  - To avoid the need for command-line *input*, make sure the user can control the application solely through its graphical user interface (GUI). For example, if your application uses the MATLAB `input` function to acquire command-line input from the user, replace it with the `inputdlg` function, which uses a dialog box instead.
  - To prevent command window *output*, trap errors where possible, and use the semicolon (`;`) operator at the end of statements to suppress the display of results to the command window. Use the GUI window, or additional figure windows and dialog boxes, to display results, error messages, help text, etc. See "Trapping Errors" on page 1-8 for more information about handling errors. Also, see "Setting the Global Warning Behavior on UNIX" on page 1-13 for information about UNIX warnings.

    If you need to use an evaluation function, then use `evalc` instead of `eval`, thus capturing output text. However, be aware that evaluation functions can make it more difficult to determine which files the application depends on. See "Analyzing Functions Called by eval" in Chapter 2 for information.
- Disable most or all default menu options on all figure windows that your application uses.
  - *(Recommended)* To disable the default menus completely, set the figure's `Menubar` property to `'none'`. You can still add your own menus.
  - To disable some default menu options but not others, see "Disabling Default Menu Options Selectively" on page 1-2. This option requires you to maintain your application carefully if you use it with future versions of MATLAB. See the cautionary note in that section.

Disabling Default Menu Options Selectively

You may want to take advantage of the built-in MATLAB tools for tasks such as printing the contents of figure windows that your application uses. This

section describes how to include selected functionality from MATLAB figure menus in your own application.

Note If you use menu callbacks from the default MATLAB figure menus and later ship your application with a future version of MATLAB, then you should test those callbacks with the future version. The names and functionality of MATLAB default menu callbacks are *not guaranteed* to remain unchanged from one version of MATLAB to the next.

You must disable these default menu options:

- From the **File** menu:
  - **New Figure**
  - **Preferences...**
- **Edit** menu
- **View** menu
- **Insert** menu
- From the **Tools** menu:
  - **Select & Edit**
  - **Move Camera**
  - **Camera Motion**
  - **Camera Axis**
  - **Camera Reset**
  - **Basic Fitting**
  - **Data Statistics**
- **Window** menu
- **Help** menu

To use a menu bar on your own figure window that is a modification of the default menu bar:

1. Close all figures that may have been open from previous MATLAB work in the current session.

2. Prepare a modified version of the default menu bar by navigating to your application's working directory and executing these commands.

```
figure('menubar','figure','toolbar','none')
set(findall(gcf),'handlevisibility','on','serializable','on')
m = get(gcf,'children');
figure('menubar','none','toolbar','none')
copyobj(flipud(m),2)
set(findobj(gcf,'label','&Edit'),'visible','off')
set(findobj(gcf,'label','&View'),'visible','off')
set(findobj(gcf,'label','&Insert'),'visible','off')
set(findobj(gcf,'label','&Window'),'visible','off')
set(findobj(gcf,'label','&Help'),'visible','off')
set(findobj(gcf,'label','&New Figure'),'visible','off')
set(findobj(gcf,'label','Pre&ferences...'),'separator',...
   'off','visible','off')
set(findobj(gcf,'label','Select && &Edit'),'visible','off')
set(findobj(gcf,'label','Move &Camera'),'visible','off')
set(findobj(gcf,'label','Camera &Motion'),'visible','off')
set(findobj(gcf,'label','Camera A&xis'),'visible','off')
set(findobj(gcf,'label','Camera Re&set'),'visible','off')
set(findobj(gcf,'label','&Basic Fitting'),'visible','off')
set(findobj(gcf,'label','&Data Statistics'),'visible','off')
hgsave(gcf,'mymenufile.fig')
close gcf; close gcf
```

3 In your application, issue this command to open a new figure window with the customized menu.

```
newfig = hgload('mymenufile.fig');
```

You can also make other modifications using MATLAB layout tools, such as changing the callbacks for menu options that are not disabled, and adding new menu options of your own. See “Creating GUIs” for details on using the layout tools.

Providing a Way to Exit the Application

Since the user cannot access the command window to type the usual MATLAB `quit` command, you need to provide a mechanism for the user to exit the application (including quitting MATLAB). Two common ways to do this are by setting the GUI's `CloseRequestFcn` property to quit the application, and by providing buttons and menus on the interface that enable the user to quit the application. Both techniques are described below.

Tip You might want the exit mechanism to quit MATLAB for the end user but not quit your own MATLAB session while you're still developing the application. If so, use the `isruntime` function to test whether or not the application is running with commercial MATLAB. For example, your GUI might have a button whose callback function includes lines like these:

```
if isruntime
   close all
   quit force
else
   close all
end
```

Using the CloseRequestFcn to Exit the Application

As a minimal measure, you should configure the `CloseRequestFcn` property of the main GUI figure window to exit the application. This allows the user to exit the application by clicking the GUI window's Close box.

Note The default `CloseRequestFcn` for a MATLAB GUI is the function `closereq.m`, which simply deletes the figure window. Since deleting the main GUI window leaves the user with no means to exit the Runtime Server, you should substitute an alternate `CloseRequestFcn` for the main GUI figure window. If you do not want the GUI's Close box to exit the application (as shown below), you can specify some other action for the `CloseRequestFcn` (use an empty `CloseRequestFcn` string to disable the Close box entirely).

For example, the following command creates a GUI window whose Close box executes a function called `shutdown` when clicked.

```
fig = figure('HandleVisibility','Callback','Menubar','none',...
'CloseRequestFcn','shutdown');
```

This shut-down function, which you write, can perform final operations (such as saving the user's work and settings) and then explicitly execute the `quit` command. Note that the shut-down function should close the GUI figure window *before* executing the `quit` command. This prevents possible recursion between the `quit` function and `CloseRequestFcn`.

Using a Uicontrol or Uimenu to Exit the Application

In addition to adapting the `CloseRequestFcn`, you might want to provide a button or menu option on the GUI that allows the user to exit the Runtime Server.

For example, a common GUI convention on most platforms is a **File** menu containing a **Quit** option. You can create a menu like this by using a uimenu object with `'shutdown'` as the callback, where `shutdown.m` is a shut-down function that you write. For example,

```
fig = figure('HandleVisibility','Callback','Menubar','none',...
'CloseRequestFcn','shutdown');
filemmenu = uimenu(fig,'Label','&File');
quitmenu = uimenu(filemmenu,'Label','&Quit',...
'Accelerator','Q','Callback','shutdown');
```

Note Although MATLAB figure windows are created with a default **File** menu that includes a **Quit** option, you may have disabled this menu by setting the figure's `Menubar` property to `'none'` (see "Preventing Command Window Input/Output" on page 1-2).

You can create a **Quit** button in a similar way.

```
fig = figure('HandleVisibility','Callback','Menubar','none',...
'CloseRequestFcn','shutdown');
uicontrol(fig,'Style','Pushbutton','String','Quit',...
'Callback','shutdown')
```

Trapping Errors

Error trapping helps suppress command-window output and helps your Runtime Server application run more smoothly. It also allows you to design user-friendly features such as context-sensitive error dialog boxes. Wherever the application executes a MATLAB command that could potentially generate an error, use a `try-catch-end` structure.

In a `try-catch-end` structure (below), the `try` block contains the MATLAB commands that you want the application to evaluate and execute (*`expression1`* and *`expression2`*). If the `try` block executes successfully, then the `catch` block is ignored. If the `try` block generates an error during execution, then the `catch` block (*`expression3`* and *`expression4`*) is executed in its place, and the error is suppressed. Use the `lasterr` command to find out what the error was.

```
try
   expression1;
   expression2;
catch
   expression3;
   expression4;
end
```

Tip Even if you avoid using the `error` command, you might invoke MATLAB functions that use it. Therefore, you should use these error trapping techniques at a high structural level in your application so that the application can trap all errors and handle them in an appropriate way.

Setting the Global Error Behavior on a PC

When an untrapped error occurs on PC platforms, the Runtime Server reacts with the error behavior that you specify in the `matlabrt.m` file. The syntax is

```
runtime errormode mode
```

The three possible *mode* values and the corresponding responses to an untrapped error are in the table below.

Table 1-1: Values of mode and Responses to Untrapped Errors

mode	Response to Untrapped Error
`continue`	Ignore the error; do not inform the user
`quit`	Prompt the user to quit the application
`dialog`	Prompt the user to choose between ignoring the error and quitting the application

The Runtime Server defaults to the `dialog` mode if `matlabrt` does not specify an error behavior; `dialog` is the *recommended* error behavior for a runtime application.

These three modes are described below. Note that the error mode setting is only effective when the application runs with the Runtime Server; if the application is running in commercial MATLAB, whether MATLAB emulates the Runtime Server or not, then untrapped errors are displayed in the command window. The difference between the Runtime Server and commercial MATLAB runtime emulation is explained in “Emulating the Runtime Server” on page 2-11. Also, the error mode setting does not apply to the execution of the `matlabrt` file. When executing `matlabrt`, the Runtime Server always halts when an error occurs.

Ignore Errors

In `continue` mode, the Runtime Server does not suspend execution when an error is encountered, and does not inform the user of the error. As a result of the error, the application might lose some portion of its functionality. The degree to which the application is affected typically depends on the source of the error; a few possibilities are:

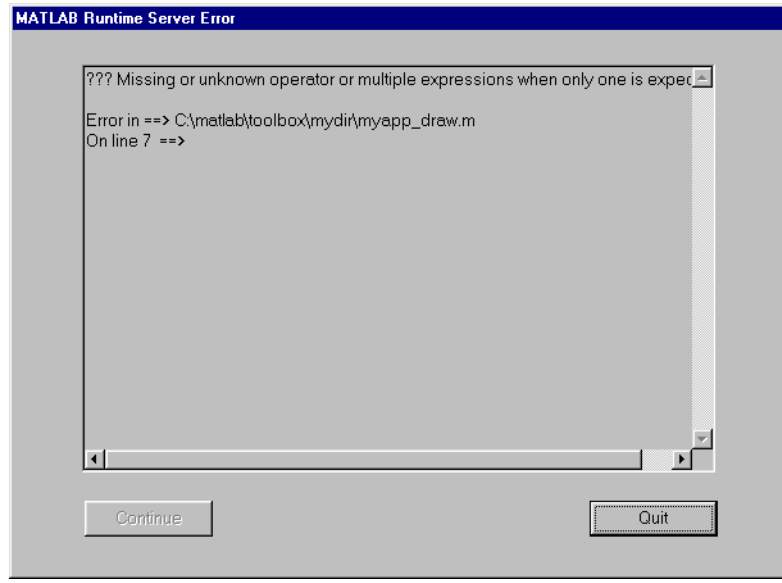
- Limited loss of application functionality. For example, if a button callback string contains a misspelled function name, the Runtime Server might generate the following error when the button is pressed:

  ```
  ??? Undefined function or variable 'function'.
  ??? Error while evaluating uicontrol Callback.
  ```

  The error message is not visible to the user, and the error itself does not impact any other area of the application, although the affected button might be unresponsive.
- Unexpected application behavior. For example, if an error is generated as the result of invalid input to a mathematical function, the application might generate inaccurate results. Since the error message is suppressed, the user might not be aware that a problem has occurred.
- Failure of the application. A severe error might lead to a substantial loss of functionality. The user might be forced to quit the application.

Prompt to Quit

In `quit` mode, every untrapped error generates an error dialog box (like the one shown below) containing the MATLAB usual diagnostic message.



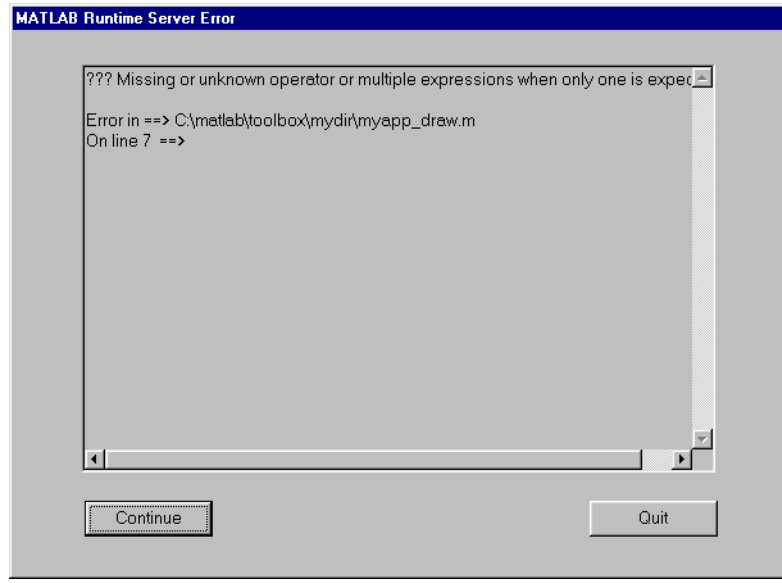
When the user dismisses this dialog the application quits.

Use the “Prompt to Quit” error mode if you want to ensure that the user is not able to continue using the application after an error occurs.

Note Use the “Prompt to Quit” error mode for backwards-compatibility with version 5.1 Runtime Server applications.

Prompt to Choose Between Continuing and Quitting

In `dialog` mode, every untrapped error generates an error dialog box (like the one shown below) containing the MATLAB usual diagnostic message and two buttons, **Continue** and **Quit**.



If the user selects **Continue**, then the dialog box disappears and the Runtime Server continues running the application. If the user selects **Quit**, then the application quits.

Since the user might not be able to distinguish a superficial error from a serious one, you should not rely on this global error setting as a substitute for trapping errors locally.

Setting the Global Warning Behavior on UNIX

The UNIX Runtime Server's default warning behavior, like that of commercial MATLAB, directs warning messages to the command window. Since a Runtime Server application displays these warning messages in the terminal window, you might want to change the application's warning behavior.

You can instruct the Runtime Server to suppress warning messages by using the statement `warning off` in `matlabrt.m`, instead of `warning backtrace`.

2

Developing a MATLAB Runtime GUI Application

Process: MATLAB Runtime GUI Application

Now that you have a MATLAB application that you developed while considering the design issues from Chapter 1, "Design Issues for a Runtime Application," these steps will convert it into a MATLAB runtime GUI application:

- *Organizing* files and managing startup tasks
- *Compiling* the application M-files into runtime P-files that the MATLAB Runtime Server recognizes
- *Testing and debugging* the application
  - By having commercial MATLAB emulate the Runtime Server
  - With the Runtime Server variant (See also "Final Testing" on page 4-9)

Overview of This Chapter

This chapter discusses these steps as they relate to MATLAB runtime GUI applications. The example in the section "Example: MATLAB Runtime GUI Application" on page 2-16 works through the first three steps for a sample MATLAB runtime GUI application. The section "Summary List: MATLAB Runtime GUI Application" on page 2-24 summarizes the process of converting an application into a MATLAB runtime GUI application, for quick reference.

Organizing Files and Managing Startup Tasks (GUI)

This section discusses the locations of the files you write for the runtime application. It also discusses the special startup and path definition functions that the Runtime Server invokes when it first runs. These utility functions, `matlabrt` and `pathdefrt`, are variations of the functions `matlabrc` and `pathdef` that commercial MATLAB invokes upon startup. The table below compares the startup sequences of the two versions of MATLAB; the file shown to the left of an arrow (->) launches the file shown to the right.

MATLAB Variant	Startup Sequence
Commercial	`matlab -> matlabrc.m -> pathdef.m`
Runtime Server	`matlab -> matlabrt.p -> pathdefrt.p`

Where to Place Your Files

When you package and ship the application, your files must reside in some directory underneath the MATLAB `toolbox` directory. This restriction has two important consequences, however:

- Because MATLAB caches the functions underneath `toolbox`, each time you change a file there you must either restart MATLAB or execute the command

  ```
  rehash toolboxreset
  ```

  to register the changes. In particular, this applies if you edit, recompile, delete, or move a file.
- If you decide to keep your own files in another location while developing the application (in order to avoid having to use `rehash` frequently) and move the P-files at the last minute to a subdirectory of `toolbox`, then you must remember to:
  - List the destination subdirectory in the path definition P-file `pathdefrt.p`. The `pathdefrt` function is discussed below in "Creating the Path Definition Function" on page 2-4.
  - Move the P-files to their `toolbox` destination during your testing process. Then use `rehash toolboxreset` to register the changes.
  - Delete, regenerate, and move P-files to their `toolbox` destination each time you change the source M-files. Use `rehash toolboxreset` as necessary to register the changes to `toolbox` subdirectories.

Tip You should avoid having multiple files on your path with the same name. You can use the `which fun -all` syntax to find out whether you have multiple functions named `fun` on your path.

Creating the Startup Function

The utility function `matlabrt`, essential for a MATLAB Runtime Server application, is a variation of the `matlabrc` function that commercial MATLAB uses. To create a `matlabrt` function for your application, you can modify the template `toolbox\runtime\matlabrt_template.m`, and place the modified file in `toolbox\local`. This section describes the properties that your `matlabrt` function should have.

Properties of the matlabrt Function. The `matlabrt` function must:

- Reside in the `toolbox\local` directory
- Perform the ordinary startup tasks of `matlabrc`, such as calling the path definition function
- Launch the rest of the application, because `matlabrt.p` is the only file that the Runtime Server directly calls. For example, if your top-level application M-file is called `myapp.m`, then `matlabrt.m` should contain the line

  ```
  myapp
  ```

You can also choose to have `matlabrt` perform these optional tasks:

- (*PC only*) Set the global error behavior for the application. Include the command

  ```
  runtime errormode mode
  ```

  in `matlabrt.m`, where *`mode`* can be `continue`, `quit`, or `dialog`. This setting controls how the Runtime Server responds to untrapped errors: by ignoring them, prompting the user to quit, or prompting the user to decide between continuing and quitting. See "Setting the Global Error Behavior on a PC" on page 1-9 for a complete description of these options. If you do not specify the global error behavior in `matlabrt.m`, then the Runtime Server defaults to the `dialog` setting.
- (*UNIX only*) Set the application's warning behavior. You might want to replace the `warning backtrace` statement in `matlabrt.m` with `warning off`. See "Setting the Global Warning Behavior on UNIX" on page 1-13 and the sample `matlabrt.m` file.

Creating the Path Definition Function

The utility function `pathdefrt`, essential for a MATLAB Runtime Server application, is a variation of the `pathdef` function that commercial MATLAB uses. This section describes the variations involved and gives information on how you should design `pathdefrt`.

Properties of the pathdefrt Function. The `pathdefrt` function should reside in the `toolbox\local` directory. It stores the path information and is called from `matlabrt` when the Runtime Server starts up.

All files that your application uses need to be on the Runtime Server *path* so that the Runtime Server can find them. This path should include only

directories under the `toolbox` directory. For example, perhaps your own files will reside in `toolbox\myapp` in the end user's installation and perhaps your files depend on MATLAB functions from `toolbox\matlab\general`, `toolbox\matlab\ops`, etc.

The path should also include `toolbox\local`. Furthermore, if an application uses files from private or class directories (which `depfun` and `depdir` will indicate), then these directories should *not* be added to the path. However, their parent directories should be included on the path. For example, if `depdir` lists

```
C:\matlab\toolbox\matlab\funfun\@inline
```

then the path should include the `toolbox\matlab\funfun` directory.

Private and class directories are described in the "Programming and Data Types" part of the MATLAB documentation. To find out whether your application uses private or class directories, apply `depfun` or `depdir` to the top-level application file and check the results for directories that either are named `private` or have the `@` symbol at the beginning of their names.

Creating the pathdefrt.m Function. You can adapt the template `pathdefrt_template.m` provided in the `toolbox\runtime` directory. To modify it for your application:

1. Copy `pathdefrt_template.m` into `toolbox\local` and rename it `pathdefrt.m`.

2. Use the `depdir` function to find out which directories contain necessary application files. If `matlabrt.m` is the top-level M-file for your application, then use the command below.

   ```
   list = depdir('matlabrt');
   ```

3. Omit private and class directories from `list`.

4. Paste the contents of `list` into the `pathdefrt.m` file that you are building, and edit the lines as necessary to adjust formatting. Path elements should have the form

   ```
   '$toolbox/matlab/general:'
   ```

   and not something like `'C:\Apps\matlab\toolbox\matlab\general'`

5 If your own application files are not currently in a subdirectory of `toolbox`, then add entries to reflect your ultimate shipping structure. See "Where to Place Your Files" on page 2-3 for more information about where to place your application files.

Other Path Specification Considerations

The Runtime Server generates warnings if it cannot locate all of the directories that are specified on the runtime path. For example, if you specify the `specfun` directory on the runtime path but do not ship it, then the Runtime Server generates the following warning at startup.

```
Name is nonexistent or not a directory:
C:\matlab\toolbox\matlab\specfun
```

However, on PC platforms, this warning message is not actually visible to your end user because the command window is minimized. On UNIX platforms, if the warning level is not set to `off`, then the message is displayed in the terminal window.

Compiling the Application (GUI)

In order for the application to work with the MATLAB Runtime Server, you must compile the application M-files into *runtime* P-files. A runtime P-file differs from a standard P-file in that the former is stamped with the same password that you used when executing `rtsetup`.

This section discusses:

- The compilation process in general
- How to compile the application with one command
- How to compile selected M-files
- How to remove compiled files

Overview of Compilation

A shipping runtime application contains compiled versions of these files:

- Files you write for your application
- M-files from `toolbox\matlab` that your application uses
- `matlabrt.m` and `pathdefrt.m`

Note Runtime P-files can be created only by a *stamped* commercial copy of MATLAB. Also, the Runtime Server inherits the password of its commercial MATLAB parent and can execute only those runtime P-files having that *same* password. See "Password Consistency Rules" on page xi to ensure compatibility. (Commercial MATLAB can execute both standard and runtime P-files.)

Note Purchase of the MATLAB Runtime Server does not imply a license to compile and ship functions from MATLAB add-on products. Please contact The MathWorks for information on shipping components of add-on products.

The easiest way to compile the application is to use the `buildp` function. This function determines which files to compile and compiles them. Thus you do not have to compile each file individually. To learn about this approach, see "Compiling the Application with One Command" on page 2-8 below.

Alternatively, you can use `depfun` in conjunction with `pcode`. To learn about this approach, see "Compiling Selected M-Files" on page 2-9.

Note When compiling your application and creating your deployment structure, you must use the `-inplace` flag with `pcode` or the default `'develStruct'` option of `buildp` to have the P-files mirror the directory structure of the M-files.

Tips for Compiling. Here are a few tips to keep in mind when compiling M-files:

- Before compiling files or seeking dependencies, you should use `cleanp` to remove all files with a `.p` extension on the MATLAB path. This function also removes `.p` files in the current directory.
- If you do not remove existing P-files, then `buildp` and `pcode` overwrite them if necessary.
- To register the changes after creating or deleting P-files, you should execute a `rehash toolboxreset` command or restart MATLAB. If you use `cleanp` or

buildp, then you do not need to issue this command since cleanp and buildp do it for you.

- You might want to save the output of buildp or depfun in a MAT-file for later use. For example, before packaging your application to ship to users, you need a list of all files that the application depends on.

Compiling the Application with One Command

The buildp function creates P-code for an entire runtime application once you specify the key files in the application. Typically, you specify only the top-level file, though you might need to specify a small number of other files, as mentioned in the troubleshooting section below. The buildp function determines which files to compile, compiles them into runtime P-files, and places the runtime-ready files alongside their uncompiled counterparts.

The buildp command below creates P-code for a runtime application whose top-level file is matlabrt.m.

```
log = buildp({'matlabrt'});
```

The output log is a string containing the name of a file that details the various phases of execution of buildp.

Troubleshooting After Using buildp. For some applications, one call to buildp creates the entire set of runtime-ready files. However, for some applications, you might need to use buildp more than once to determine the best input list for buildp. Check the the log file (whose filename is the output string log) and/or the output for information and diagnostics. Here are some troubleshooting tips in case buildp either fails, indicates a potential problem, or gives incomplete results:

- If your application uses toolbar items from the MATLAB default figure window, then include 'FigureToolBar.fig' in the first input argument of buildp.
- If your application uses menu items from the MATLAB default figure window, then include 'FigureMenuBar.fig' in the first input argument of buildp.
- If your application uses GUI elements and creates .fig files, then include those .fig files in the first input argument of buildp.

- If `buildp` cannot parse a file or cannot resolve a symbol, then its results might be incomplete. Check your files for syntax errors, misspelled variable or function names, and other errors. Then invoke `buildp` again.
- If your application (including functions on which the application depends) uses an evaluation function (`eval`, `evalc`, `evalin`, or `feval`), then `buildp` cannot determine whether the evaluation string contains the name of a function that needs to be compiled. `buildp` reports instances of evaluation functions. You should check each instance manually and decide whether any additional files should be compiled. Then invoke `buildp` again and include those additional files in the first input argument.

  For example, if you use an evaluation command to execute either `comp1.m` or `comp2.m`, then you can modify a `buildp` command like

  ```
  log = buildp({'matlabrt'});
  ```

  so that it becomes

  ```
  log = buildp({'matlabrt','comp1','comp2'});
  ```

  For additional suggestions for dealing with evaluation functions, see "Analyzing Functions Called by eval" on page 2-13.

Compiling Selected M-Files

To compile selected M-files into runtime P-files, use the `pcode` command with the `runtime` flag. The `runtime` flag tells MATLAB to create a *runtime* P-file, instead of the standard P-file. Other flags and options can control which files are compiled and where the runtime P-files are placed, as explained below.

Use the `-inplace` flag with the `pcode` command, as follows. For the file `myfile.m`, type

```
pcode myfile -inplace -runtime
```

To compile the entire application, apply `depfun` to the top-level M-file (`matlabrt.m`) and then apply `pcode` to the output. This is an alternative to using the `buildp` command as explained earlier. Here, the purpose of using `depfun` is to determine which M-files to compile. The commands are below.

```
list = depfun('matlabrt');
pcode(list{:},'-inplace','-runtime')
```

Troubleshooting After Using pcode. If your application uses toolbar or menu items from the MATLAB default figure window, then you need to include

`'FigureMenuBar.fig'` and `'FigureToolBar.fig'` in your input to `depfun`. If your application uses GUI elements and creates `.fig` files, then you should include those `.fig` files in your input to `depfun` as well.

Also note that there are some circumstances in which `depfun` provides an incomplete list. For details, see "Analyzing Functions Called by eval" on page 2-13.

Removing P-Files

To remove *all* files with a `.p` extension on the path and in the current directory, type

```
cleanp
```

Caution `cleanp` looks only at names, not contents, of files. Use caution if you have files other than MATLAB P-files that use a `.p` filename extension.

Testing While Emulating the Runtime Server (GUI)

Once you have compiled all of the M-files of a MATLAB runtime GUI application, you can test the application with the Runtime Server. However, since the Runtime Server does not provide a command window, for the purpose of debugging it is much easier to have your development version of MATLAB *emulate* the Runtime Server as a first step in the testing process. You should still test the application with the actual Runtime Server variant later. This section discusses both kinds of testing and includes a troubleshooting section.

Moving P-Files to Final Locations

If you store your source M-files outside of the `toolbox` directory, then at some point during your testing you should move the corresponding P-files to their final destinations under `toolbox`. Then use `rehash toolboxreset` to register the changes.

Each time you change the source M-files, remember to delete, regenerate, and move the P-files to their `toolbox` destination. Again, use `rehash toolboxreset` as necessary to register the changes to `toolbox` subdirectories.

Emulating the Runtime Server

Commercial MATLAB can emulate the Runtime Server environment by disabling the MATLAB ability to read M-files and standard P-files. The command window remains active, however. This section describes the procedure and some tips for testing while MATLAB emulates the Runtime Server.

To test and debug while MATLAB emulates the Runtime Server, follow this procedure:

1. At the MATLAB prompt, type

   ```
   runtime on
   ```

   to start emulating the Runtime Server.

   To find out whether MATLAB is emulating the Runtime Server at a given time, type

   ```
   runtime status
   ```

   at the command line.

2. Run your application from the command line. For the most accurate simulation of the Runtime Server environment, launch the application through `matlabrt.p`, as the Runtime Server does. At the command line type

   ```
   matlabrt
   ```

3. If there are errors, the usual error report in the MATLAB command window shows you where they occurred. To execute M-files and debug your functions, first turn off Runtime Server emulation by typing

   ```
   runtime off
   ```

   and then debug the problematic M-files as you normally would.

4. Whenever you change an M-file, be sure that runtime MATLAB registers those changes: If the M-file is under the `toolbox` directory, then use `rehash toolboxreset`. Then, use `buildp` to recompile the M-file into a runtime P-file. Finally, test the application again with MATLAB emulating the Runtime Server.

5 To exit the application without quitting MATLAB, close the application's GUI from the command line by typing

```
close force
```

Runtime Server Emulation Considerations

There are a few things you should keep in mind when using MATLAB to emulate the Runtime Server:

- Even when emulating the Runtime Server, commercial MATLAB displays untrapped errors in the command window. To see the effects of the global error behavior that you specified in `matlabrt.m`, you must run the application with the Runtime Server variant, as described in "Testing with the Runtime Server Variant (GUI)" on page 2-15.
- It might help to clear memory-resident functions by typing `rehash toolboxreset` before starting to emulate the Runtime Server. This ensures that MATLAB does not use a P-file already in memory from an earlier run.
- You might want to save your development path before running the application, since `matlabrt` replaces it with the runtime path. Type

  ```
  devpath = path;
  save devpath devpath
  ```

  After ending Runtime Server emulation, you can restore your development path by typing

  ```
  load devpath
  path(devpath)
  ```

  If the `path.m` function is not on the runtime path, then you can use `matlabpath(devpath)` instead. Restarting MATLAB also restores your original path.

Troubleshooting

If the application runs normally with commercial MATLAB but does not run as expected with MATLAB emulating the Runtime Server, then the problem might be one of the following:

- Some M-files did not compile. If necessary M-files were not specified in the compile list for any reason, then the corresponding P-files will be missing and the application might not run properly.

  - If you did not use `cleanp` before compiling, then the application might be using a P-file that was not generated from the current version of the M-file. When this happens, you might need to delete all P-files and recompile them.
  - If your application uses toolbar or menu items from the MATLAB default figure window, then you need to include `'FigureMenuBar.fig'` and `'FigureToolBar.fig'` in your input to `buildp` or `depfun`. If your application uses GUI elements and creates `.fig` files, then you should include those `.fig` files in your input to `buildp` or `depfun` as well.
- Some directories are missing from the path. If the path specified in `pathdefrt` is lacking directories that contain application files, then the application might not run properly.

You might be able to determine the source of the problem by running the application using the runtime path with commercial MATLAB. Type

```
runtime off
rehash toolboxreset
matlabrt
```

If the application now runs as expected, then there are probably uncompiled M-files. If the problem persists even with commercial MATLAB, then the path specification in `pathdefrt` might be incomplete. In either case, the MATLAB error messages should provide a good indication of which files are not being found.

Analyzing Functions Called by eval. Calls to `buildp` or `depfun` might not find functions whose names are assembled at runtime within the calling function, and which are executed using `eval`, `evalc`, or `evalin`. For example, if your application calls a function named `action1.m` by concatenating the strings `'action'` and `'1'` in an `eval` statement,

```
x = 1;
eval(strcat('action',int2str(x)));
```

then `depfun` does not recognize that there is a function called `action1.m`.

To make certain that `depfun` *does* analyze this file, insert the following code in one of the other application files, such as `matlabrt`.

```
if 0
   action1
end
```

The `if` statement above does not execute at runtime, but it allows `depfun` to see the full `action1` function name when it analyzes `matlabrt`.

Using inmem to Find Files That depfun Misses. To find out whether `buildp` or `depfun` is missing certain application functions, you can compare the `depfun` diagnostics to the list of functions that are in MATLAB memory after the application is executed. You can view this list by using the `inmem` function, as described below.

1. Type `rehash toolboxreset` to renew the session, and then type

   ```
   matlabrt
   ```

   to launch the application.

2. Use the application as thoroughly as possible by pressing buttons, selecting menus, etc. The goal of this is to force MATLAB to load all or most of the application's functions into memory. If there is a particular area of the application that is a source of problems for `depfun`, then use it especially heavily to make sure that MATLAB loads all the relevant functions.

3. When you have used the application sufficiently to load the relevant functions into memory, quit the application without exiting MATLAB.

4. Generate the list of functions in memory by typing

   ```
   memlist = inmem
   ```

   The listed functions are those that MATLAB used to carry out the tasks that you just performed in Step 2.

5. Use `depfun`, as before, to generate a list of the application's dependent functions.

   ```
   list = depfun('matlabrt')
   ```

6 Compare the functions in `list` and `memlist`. You can do this visually, or by using a script like the one below.

```
% Extract filenames from DEPFUN output
for i = 1:length(list)
   [path,name,ext,ver] = fileparts(list{i});
   list{i} = name;
end

% DEPFUN results not listed by INMEM:
depfiles = setdiff(char(list),char(memlist),'rows')

% INMEM results not listed by DEPFUN:
memfiles = setdiff(char(memlist),char(list),'rows')
```

The `memfiles` variable contains a list of functions found by `inmem` but not by `depfun`. To force `depfun` to analyze these functions, you can place a nonexecuting `if` conditional in an application file such as `matlabrt.m`, as shown in "Analyzing Functions Called by eval" on page 2-13.

The `depfiles` variable contains a list of functions found by `depfun` but not by `inmem`. These are probably valid application files that were not called by MATLAB (and not loaded into memory) when you ran the application.

Compiling Extra M-Files. If you cannot anticipate what functions might be invoked via an evaluation command, then you might decide to compile some extra M-files. As a last resort, you can compile *all* functions on the path and in the current directory, using the `pcodeall` function.

Testing with the Runtime Server Variant (GUI)

Although commercial MATLAB can emulate the Runtime Server environment, testing your application with the actual Runtime Server variant is important. To launch the application with Runtime Server, at the system prompt, type

```
matlab -runtime
```

The `-runtime` flag is an argument that tells your development copy of MATLAB to start itself as the Runtime Server. Use this flag while testing your application, typically in the final stages of testing. The end user's syntax for invoking MATLAB does not need to use the `-runtime` flag. This is because the absence of a `license.dat` file already signals that that copy of MATLAB must be a Runtime Server variant instead of commercial MATLAB.

Example: MATLAB Runtime GUI Application

This example illustrates these typical steps involved in preparing a MATLAB runtime GUI application for the Runtime Server:

- *Organizing* files and managing startup tasks
- *Compiling* the application M-files into runtime P-files that the MATLAB Runtime Server recognizes
- *Testing and debugging* the application
  - By having commercial MATLAB emulate the Runtime Server
  - With the Runtime Server variant (See also “Final Testing” on page 4-9)

This section begins with instructions for installing the example files included in the software and an overview of the example application.

While the example as described here is structured as a runtime GUI application, you can also use the same set of M-files to form the back end of a runtime engine application. See the file `toolbox\runtime\examples\activex\Readme` for more details.

Installing the Example Files

The example consists of seven files that can be found in the `toolbox\runtime\examples\gui` directory.

File	Description
`amortsched.m`	MATLAB Runtime Server application files
`amortsched_cb.m`	
`datagrid.m`	
`datagrid_cb.m`	
`loansched.m`	
`matlabrt.m`	MATLAB Runtime Server startup files
`pathdefrt.m`	

Follow the instructions below to install the files. To prevent any confusion about which files you are working with, try to avoid having duplicates of these files in other locations on your path.

1. Move `matlabrt.m` and `pathdefrt.m` from `toolbox\runtime\examples\gui` into the `toolbox\local` directory of your development copy of MATLAB.

2. Add the directory containing the other example files to the MATLAB path using the command below.

```
addpath(fullfile(matlabroot,'toolbox','runtime','examples','gui'));
```

Directory Structure of Application

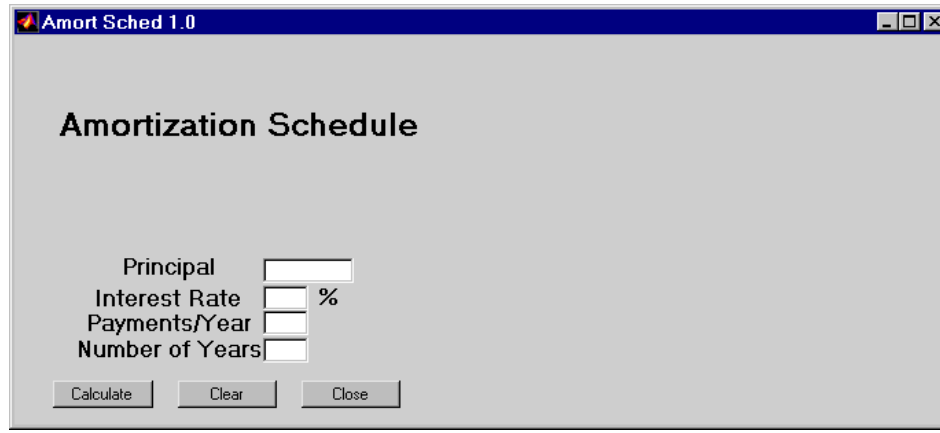
This example maintains the existing directory structure throughout the compiling and testing process. Compiled versions of the M-files in `toolbox\runtime\examples\gui` reside there; compiled versions of any other M-files that the application uses (e.g., functions in `toolbox\matlab`) reside alongside their respective M-files.

Overview of the Application

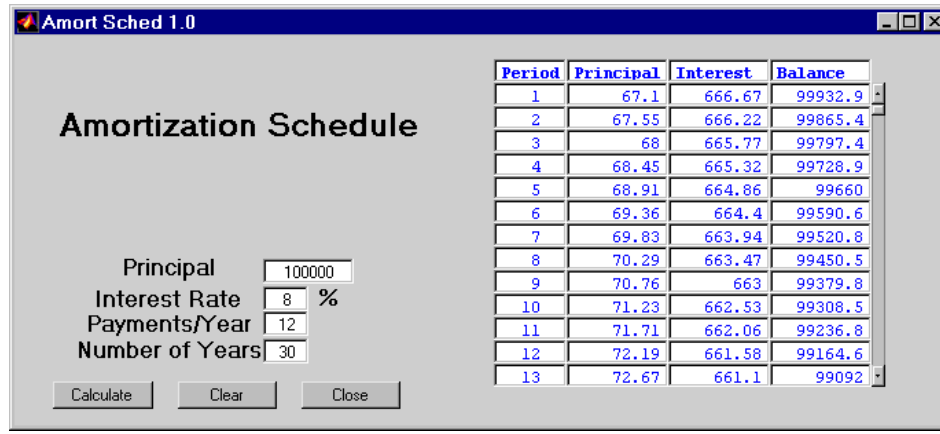
This example is a simple GUI-driven application that calculates and displays an amortization schedule. If you have added `toolbox\runtime\examples\gui` to the MATLAB path, then you can run the application by typing

```
amortsched
```

at the command line. At first, the GUI looks like this.



To use the application, enter expressions in the GUI fields and press the **Calculate** button. After you enter values and press **Calculate**, the GUI looks like this.



The **Clear** button deletes the numbers that you entered in the four input fields. The **Close** button exits the application. If the application is running with the Runtime Server or with commercial MATLAB in runtime emulation mode, then the **Close** button also exits MATLAB.

How the Application Files Interact

If you run the application with the Runtime Server, then the Runtime Server first executes `matlabrt`, which in turn invokes `pathdefrt`. The function `matlabrt` also executes `amortsched`, which sets up the GUI and the buttons' callback functions. Below is an excerpt from `amortsched.m`.

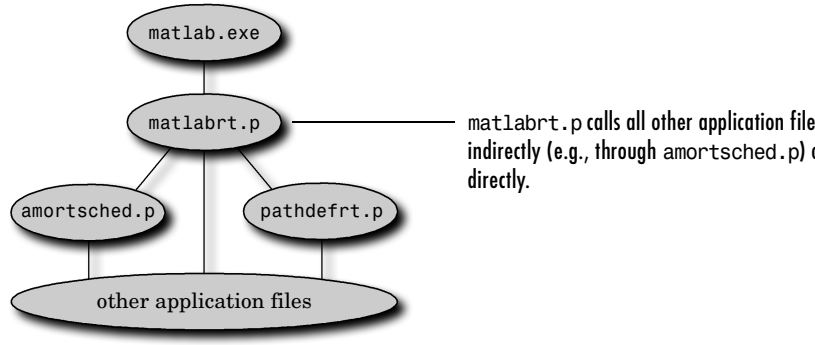
```
h0 = figure('Color',[0.8 0.8 0.8], ...
   'CloseRequestFcn','amortsched_cb(''close_amortsched'')', ...
   'MenuBar','none', ...
   'Name','Amort Sched 1.0', ...
   'NumberTitle','off', ...
   'PaperPosition',[18 180 576 432], ...
   'PaperUnits','points', ...
   'Units','characters', ...
   'Position',[68.6 7.3077 136 21.7692], ...
   'Resize','off', ...
   'Tag','Fig1', ...
   'ToolBar','none');
h1 = uicontrol('Parent',h0, ...
   etc...
```

Pressing the buttons invokes `amortsched_cb` with an argument indicating which button was pressed. The function `amortsched_cb` uses a `switch` structure to perform the actions associated with each GUI button. The lines below show the structure of `amortsched_cb.m`.

```
switch action
case 'clear_values'
   % Insert code here that clears the GUI fields.
case 'calculate_values'
   % Insert code here that calculates and displays
   % the amortization schedule.
case 'close_amortsched'
   % Insert code here that exits the application.
end
```

When necessary, `amortsched_cb` invokes other functions to compute or display results.

The figure below illustrates schematically which functions call each other in this application.



Adapting the Design for Runtime Execution

Although `amortsched` is a simple application, its files incorporate several adaptations for Runtime Server execution:

- The `figure` command in `amortsched.m` includes the `'Menubar','none'` specification. This deactivates the GUI's default menus, as discussed in "Preventing Command Window Input/Output" on page 1-2.
- The application requires no input from the command window and sends no output to the command window. It uses a dialog box to display errors.
- The figure window definition in `amortsched.m` specifies a `CloseRequestFcn`. Since the **Amort Sched 1.0** window is the top-level GUI figure window, its **Close** button exits the application.

Note The code that the `CloseRequestFcn` executes avoids quitting MATLAB when commercial MATLAB is *not* in runtime emulation mode. Logic such as

```
close('force','Amort Sched 1.0') % Close the GUI window.
if isruntime, exit, end
```

conveniently prevents you from having to restart MATLAB many times while your application is in an early development stage.

Organizing Files and Managing Startup Tasks

This example keeps its own non-startup files in `toolbox\runtime\examples\gui` during the entire development, compilation, and testing process. If you change any application files while exploring this example, then use `rehash toolboxreset` to register the changes.

The Startup Function

The example `matlabrt.m` file, which the section "Installing the Example Files" on page 2-16 instructed you to install in the `toolbox\local` directory, is appropriate for this application because it:

- Does not display anything in the command window
- Sets the global error behavior for the application
- Sets the warning level to `off`
- Invokes `pathdefrt` (not `pathdef`)
- Launches the top-level application function, `amortsched`

Creating the Path Definition Function

The example `pathdefrt.m` file that you installed in `toolbox\local` uses paths relative to the MATLAB root directory. Each distinct directory that contains a function used in this application is included on the runtime path.

Notice that the list of directories in the example `pathdefrt.m` matches the list of (non-private, non-class) directories in the output of the `depdir` function.

```
cleanp; % Remove all .p files from path.
p = depdir('matlabrt')

p =

    'C:\matlab\toolbox\matlab\datafun'
    'C:\matlab\toolbox\matlab\datafun\@cell'
    'C:\matlab\toolbox\matlab\datatypes'
    'C:\matlab\toolbox\matlab\elfun'
    'C:\matlab\toolbox\matlab\elmat'
    'C:\matlab\toolbox\matlab\general'
    'C:\matlab\toolbox\matlab\general\@char'
    'C:\matlab\toolbox\matlab\graph3d'
    'C:\matlab\toolbox\matlab\graphics'
```

```
'C:\matlab\toolbox\matlab\graphics\private'
'C:\matlab\toolbox\matlab\iofun'
'C:\matlab\toolbox\matlab\lang'
'C:\matlab\toolbox\matlab\ops'
'C:\matlab\toolbox\matlab\ops\@cell'
'C:\matlab\toolbox\matlab\specfun'
'C:\matlab\toolbox\matlab\strfun'
'C:\matlab\toolbox\matlab\strfun\@cell'
'C:\matlab\toolbox\matlab\uitools'
'C:\matlab\toolbox\local'
'C:\matlab\toolbox\runtime\examples\gui'
```

The path omits private and class directories, while listing their parent directories instead. For example, `pathdefrt.m` lists `toolbox\matlab\ops`, but not `toolbox\matlab\ops\@cell`.

Compiling the Application

This example uses the `buildp` function to place each P-file in the same directory as its uncompiled counterpart. Since `matlabrt.m` is the top-level M-file that invokes all other M-files, the command below compiles the entire application.

```
[log,depfunout,pcodeout] = buildp({'matlabrt'});
```

The output indicates that there might be problem symbols or `eval`-like constructs.

```
BUILDP finished but there may be file(s) with either problem symbols
or EVAL-like constructs.  Check BUILDP log for details.
```

This is normal for this example, since the example uses files that contain `eval` commands.

Testing While Emulating the Runtime Server

It is useful to test and debug an application while your development copy of MATLAB emulates the Runtime Server, before actually running the application with the Runtime Server. Follow these steps:

1. If you have just compiled P-files in a `toolbox` directory, then restart MATLAB or type `rehash toolboxreset`. Either action updates the internal file list.

2 Save your development path so that you can restore it later. Type

```
devpath = path;
save devpath devpath
```

3 Tell MATLAB to start emulating the Runtime Server. Type

```
runtime on
```

4 Run the example application by typing

```
matlabrt
```

The example's GUI opens. Test the application by filling in the GUI fields and pressing the **Calculate** button.

5 Exit the application by typing `close force` at the MATLAB command line. This closes the figure window without quitting MATLAB.

6 Tell MATLAB to stop emulating the Runtime Server. Type

```
runtime off
```

7 Restore the development path by typing

```
load devpath
path(devpath)
```

Testing with the Runtime Server Variant

To launch the application with the actual Runtime Server variant, use the command

```
matlab -runtime
```

instead of the ordinary command `matlab`.

Summary List: MATLAB Runtime GUI Application

This list summarizes the steps for creating a MATLAB runtime GUI application from a GUI-based application in MATLAB. The hyperlinks lead to the sections that describe the steps in more detail.

1. Stamp your development copy of MATLAB.

2. Adapt the design of the MATLAB portion of the application as necessary. See Chapter 1, "Design Issues for a Runtime Application."

3. Decide where to place the files you write for the runtime application.
   - If you keep all files under `toolbox`, then remember to use `rehash toolboxreset` to register each change you make.
   - If you keep the M-files outside of `toolbox` and move the P-files under `toolbox` only towards the end of your development and testing process, then remember to include the final destination directories in `pathdefrt`.

4. Create the startup function `matlabrt.m` by copying `toolbox\runtime\matlabrt_template.m` to `toolbox\local` and renaming it `matlabrt.m`. Then modify it as necessary and use `rehash toolboxreset` to register the change. It must:
   - Invoke `pathdefrt`.
   - Launch the application.

5. Create the path definition function `pathdefrt.m` by copying `toolbox\runtime\pathdefrt_template.m` to `toolbox\local` and renaming it `pathdefrt.m`. Then modify it as necessary:
   - Use `cleanp` to remove any `.p` files from your path and current directory.
   - Use `depdir('matlabrt')` to determine the runtime path.
   - If you plan to move your P-files under `toolbox` later on, then include their destination directories in the runtime path.
   - Use `rehash toolboxreset` to register the change in `pathdefrt.m`.

6. Use `buildp` to compile all application M-files into runtime P-files.

   Possible complications: toolbar or menu items, functions invoked by `eval`, MATLAB add-on products

7 Test the application:

- With your P-files in their final locations under `toolbox`, if they are not already there. After moving them, use `rehash toolboxreset` to register the changes.
- By having MATLAB emulate the Runtime Server. Use `runtime on` and `runtime off`.
- With the Runtime Server variant.

Return to earlier steps as necessary. When debugging, remember to stay current by deleting old P-files, producing new P-files whenever an M-file changes and using `rehash toolboxreset` as necessary to register changes.

3

Developing a MATLAB Runtime Engine Application

Process: MATLAB Runtime Engine Application

If you have an application that uses MATLAB and that you developed while considered the design issues mentioned in Chapter 1, "Design Issues for a Runtime Application," then the process of converting it into a MATLAB runtime engine application is largely the same as the corresponding process for a MATLAB runtime GUI application. (See "Overview of MATLAB Runtime Applications" on page vi for definitions of the two types of runtime applications.)

As with a MATLAB runtime GUI application, there are four main steps in converting an application that uses MATLAB into a MATLAB runtime engine application:

- *Organizing* files and managing startup tasks
- *Compiling* the application M-files into runtime P-files that the MATLAB Runtime Server recognizes
- *Testing and debugging* the application
  - By having commercial MATLAB emulate the Runtime Server
  - With the Runtime Server variant (See also "Final Testing" on page 4-9)

Overview of This Chapter

This chapter discusses the relationship between the Runtime Server and the MATLAB Engine API, describes the two parts of a MATLAB runtime engine application, and then discusses the steps listed above. It also highlights the important differences between developing a MATLAB runtime GUI application and developing a MATLAB runtime engine application. The examples in:

- "ActiveX Automation Example" on page 3-9 and
- "Engine API Example" on page 3-21

work through the first three steps for sample MATLAB runtime engine applications. For an additional example of a runtime engine application that uses ActiveX Automation, see the description in the file `toolbox\runtime\examples\activex\Readme`. Finally, the section "Summary List: MATLAB Runtime Engine Application" on page 3-29 summarizes the process of converting an application into a MATLAB runtime engine application, for quick reference.

Computation and the MATLAB Engine API

The MATLAB Runtime Server uses the same MATLAB Engine Application Program Interface (API) that commercial MATLAB uses. The MATLAB documentation contains useful information about the MATLAB Engine API and interprocess communication. You might need to consult MATLAB documentation while developing your MATLAB runtime engine application. However, some special issues affect the design, coding, testing, and packaging of applications that are destined for use with the Runtime Server rather than with the full MATLAB Engine API. This chapter describes such special issues as they relate to MATLAB runtime engine applications.

Note MATLAB runtime engine applications can communicate with MATLAB via routines from the Engine API library (e.g., `engGetArray`) or via other means such as ActiveX and pipes.

Parts of a MATLAB Runtime Engine Application

A typical MATLAB runtime engine application has two distinct parts:

- The front-end GUI, which is usually implemented in a GUI-oriented language like Visual Basic
- The back-end computational engine, which is implemented with MEX-files and/or MATLAB runtime P-files

If the front-end GUI that you are planning to use together with the MATLAB runtime engine application already works with commercial MATLAB, then you do not need to change this front-end code for Runtime Server operation.

The back-end MATLAB part of the application should comply with the design principles discussed in Chapter 1, “Design Issues for a Runtime Application.”

Organizing Files and Managing Startup Tasks

This section discusses the locations of the files you write for the runtime application, and special startup issues. The startup issues include the startup and path definition functions that the Runtime Server invokes when it first runs. These utility functions, `matlabrt` and `pathdefrt`, are variations of the

functions `matlabrc` and `pathdef` that commercial MATLAB invokes upon startup. This section also discusses startup considerations for PC applications.

Where to Place Your Files

You can arrange the MATLAB portion of your application just as you would in the case of a runtime GUI application. This is described in the earlier section "Where to Place Your Files" on page 2-3. MATLAB places no restriction on the directory structure of the non-MATLAB portion of your application and no restriction on the relative locations of the MATLAB and non-MATLAB portions of your application.

Creating the Startup Function

The startup function `matlabrt` is the runtime analogue of `matlabrc`. When the front end of the application launches MATLAB, MATLAB executes `matlabrt.p`. Furthermore, `matlabrt` acts as the gateway to the MATLAB portion of the application.

The `matlabrt.p` file should reside in the `toolbox\local` directory, and should perform these tasks (in addition to ordinary tasks that `matlabrc` performs):

- (*PC*) Set global error behavior for the MATLAB portion of the application. Include the command

  `runtime errormode` *`mode`*

  in `matlabrt.m`, where *`mode`* can be `continue`, `quit`, or `dialog`. See "Setting the Global Error Behavior on a PC" on page 1-9 for more information. If you do not specify the global error behavior in `matlabrt.m`, then the Runtime Server defaults to the `dialog` setting.
- (*UNIX*) Set global warning behavior. You might want to replace the `warning backtrace` statement in `matlabrt.m` with `warning off`. See "Setting the Global Warning Behavior on UNIX" on page 1-13 and the sample `matlabrt.m` file.
- Invoke the path definition function, `pathdefrt`
- (*Optional*) Include a variant test using the `isruntime` function, if it is important that the application *not* use commercial MATLAB
- Invoke other MATLAB functions as necessary

Either `matlabrt` or another M-file should be a top-level M-file in the sense that it acts as an intermediary between the front end and the application's M-file

functions. The front end then communicates *only* with this top-level M-file. When an event occurs in the front end that requires a MATLAB function to execute, the front end calls the top-level M-file with a *switch* that indicates which action should occur. The top-level M-file then executes the appropriate function to accomplish the action.

Creating the Path Definition Function

The startup function `matlabrt` invokes the path definition function `pathdefrt`. Create `pathdefrt` as described in "Creating the Path Definition Function" in Chapter 2.

PC Startup Considerations

During a MATLAB runtime engine application, MATLAB registers itself as an as an Automation server.

Default Executable. If MATLAB is not currently running as an Automation server on the system, then the MATLAB executable file (`matlab.exe`) that is launched by the controller is the one most recently run. If MATLAB is already running on the system, then the ActiveX controller uses the currently running instance of MATLAB.

Users Who Have MATLAB Installed on the System. If a copy of commercial MATLAB is installed on the same system as the Runtime Server application (for example, if the user works with MATLAB), then there is a possibility that your application will invoke commercial MATLAB instead of the Runtime Server. This will happen if the user launches the application while commercial MATLAB is running as an Automation server, or when commercial MATLAB was run as an Automation server more recently than the MATLAB Runtime Server.

The application should run normally with commercial MATLAB, although certain Runtime Server adaptations will be absent (splash screen, global error behavior, etc.). If it is important that the application *not use* commercial MATLAB, then you can include a variant test in the startup procedure using the `isruntime` command.

Handling Multiple Application Instances and Multiple Versions of MATLAB. Your application can launch the MATLAB Automation server either as a multiple-client server or as a dedicated server. Also, if a user of your application has two different versions of MATLAB installed, then your

application can launch either the default version, as described above, or a specific version. You make these choices when you decide what `ProgID` to use when coding your application. The possible values for `ProgID`, and the corresponding results when MATLAB is launched, are summarized in the table below.

ProgID	Version of MATLAB Launched	Type of Server
`Matlab.Application`	Default	Multiple-client
`Matlab.Application.6`	Version 6	Multiple-client
`Matlab.Application.Single`	Default	Dedicated
`Matlab.Application.Single.6`	Version 6	Dedicated

Suppose that the user starts up multiple instances of the application. If you have chosen to use a multiple-client server, then each new client instance shares the same instance of the Runtime Server. You should therefore either design the MATLAB side of the application to service multiple clients, or prevent multiple clients from simultaneously using the Runtime Server. If, on the other hand, you have chosen to use a dedicated server, then each instance will be served by a different instance of the Runtime Server.

Compiling the Application

Compiling M-files into runtime P-files is the same for a MATLAB runtime engine application as for a MATLAB runtime GUI application; see the earlier section “Compiling the Application (GUI)” on page 2-6.

Testing While Emulating the Runtime Server

The procedure for testing the MATLAB portion of a MATLAB runtime engine application while commercial MATLAB emulates the Runtime Server is slightly different from that for a MATLAB runtime GUI application. This section discusses Runtime Server emulation on PC and UNIX platforms, as well as testing with the Runtime Server variant. See “Troubleshooting” on page 2-12, in the earlier section “Testing While Emulating the Runtime Server (GUI),” if the application does not run as expected.

Moving P-Files to Final Locations

If you store your source M-files outside of the `toolbox` directory, then at some point during your testing you should move the corresponding P-files to their final destinations under `toolbox`. Then use `rehash toolboxreset` to register the changes.

Each time you change the source M-files, remember to delete, regenerate, and move the P-files to their `toolbox` destination. Again, use `rehash toolboxreset` as necessary to register the changes to `toolbox` subdirectories.

Emulating the Runtime Server Using ActiveX Automation (PC)

Type the following two commands at the MATLAB prompt to start emulating the Runtime Server and to set the runtime path.

```
runtime on
matlabrt
```

Then launch the application by starting up the front-end executable file. The application now uses the running MATLAB as its Automation server, and you can test the program by making the appropriate call to the runtime application.

To execute M-files and debug your functions, tell MATLAB to stop emulating the Runtime Server by typing

```
runtime off
```

Emulating the Runtime Server Using Engine API Commands

If your application is based on the MATLAB Engine API, then Runtime Server emulation enables you to test only the back-end MATLAB portion of your application. To test the front and back ends together, MATLAB must execute as a Runtime Server variant, as described in "Testing with the Runtime Server Variant" on page 3-8.

To test the MATLAB portion while MATLAB emulates the Runtime Server, follow these steps:

1. Type

   ```
   runtime on
   ```

   at the MATLAB prompt to start emulating the Runtime Server.

2 Type

```
matlabrt
```

to set the runtime path, perform other startup tasks, and invoke your application P-files.

3 To execute M-files and debug your functions, first turn off Runtime Server emulation by typing

```
runtime off
```

Then use commercial MATLAB as you ordinarily would.

Testing with the Runtime Server Variant

Although commercial MATLAB can emulate the Runtime Server environment, testing your application with the actual Runtime Server variant is still important.

On UNIX, type

```
matlab -runtime -c dummy
```

at the system prompt. Then start the front end (without quitting MATLAB).

On PC, rename the license file, launch MATLAB, and then start the front end. The license file on PC is `matlabroot\bin\win32\license.dat`. Remember to restore the license file when you want to use commercial MATLAB.

ActiveX Automation Example

This example illustrates the steps involved in preparing an ActiveX Automation-based Visual Basic application for the MATLAB Runtime Server on PC platforms. The major steps in the procedure are the same as those for the MATLAB GUI example in Chapter 2, "Developing a MATLAB Runtime GUI Application":

- Creating the startup files `matlabrt.m` and `pathdefrt.m`
- Compiling the application with the `buildp` command
- Testing the application with the Runtime Server variant

The example application uses two of the MATLAB Engine's ActiveX API methods, `Execute` and `GetFullMatrix`. The `Execute` method accepts a string argument and evaluates it in the MATLAB workspace. The `GetFullMatrix` method returns the real and imaginary components of a workspace matrix in two separate variables. See *External Interfaces* in the MATLAB documentation set for more information about these methods.

Note This example works with version 5 and later of Microsoft Visual Basic.

This section begins with instructions for installing the example files included in the software.

Installing the Example Files

This example consists of the files listed below, located in `toolbox\runtime\examples\activex` directory.

File	Description
`myappVB.frm`	Visual Basic files
`myappVB.vbp`	
`myappVB.vbw`	

File	Description
myapp.m	MATLAB Runtime Server application files
myapp_init.m	
myapp_draw.m	
myapp_erase.m	
matlabrt.m	MATLAB Runtime Server startup files
pathdefrt.m	

To install the files, follow the instructions below. To prevent any confusion about which files you are working with, try to avoid having duplicates of these files in other locations on your path.

Visual Basic Files. Move the Visual Basic files to any convenient location on your hard disk. For example, you can put them in a directory called `c:\myappVB`.

MATLAB Application Files. Create a directory called `mydir` in the `toolbox` directory of your development copy of MATLAB.

Add it to the MATLAB path using the command below.

```
addpath(fullfile(matlabroot,'toolbox','mydir'));
```

The new `mydir` directory should be at the same level as the `toolbox\local` and `toolbox\runtime` directories. Move the following files from `toolbox\runtime\examples\activex` into `toolbox\mydir`.

```
myapp.m
myapp_init.m
myapp_draw.m
myapp_erase.m
```

MATLAB Startup Functions. Move `matlabrt.m` and `pathdefrt.m` into the `toolbox\local` directory of your development copy of MATLAB, alongside the `matlabrc.m` and `pathdef.m` files.

Adapting the Design for Runtime Execution

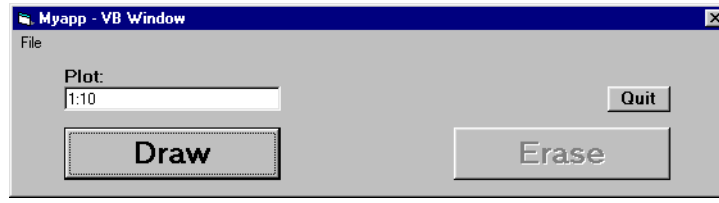
Two aspects of this example make it suitable for runtime execution:

- The way its front-end GUI design passes information among the user, the Visual Basic portion, and the MATLAB portion
- The way its top-level M-file passes information between the Visual Basic portion and the rest of the MATLAB portion

This section describes each of these two aspects in turn.

The Front-End GUI

This example is a simple GUI-driven application that evaluates and plots a MATLAB expression. Its GUI is shown below.



To use the application, enter a valid MATLAB expression in the **Plot** field of the GUI and press the **Draw** button. The application opens a MATLAB figure window containing the plot, and the **Draw** button becomes inactive.

When you press the **Erase** button, the plot is erased and the **Erase** button becomes inactive. This illustrates how the Runtime Server can return information to the Visual Basic controller.

The following sections discuss the Visual Basic code that is executed for the GUI events and form initialization. You can look at the code (as well as the objects' properties) by opening project `myappVB.Frm` in Visual Basic 5 or later.

Note You can view the available Automation methods for MATLAB from within the Visual Basic environment. First check the **Matlab Application (Version 6.0) Type Library** in the **References** dialog box (from the **Project** menu). Then use the **Object Browser** (from the **View** menu) to list the members of the **MLApp** library.

General Declarations Section. This section makes four declarations. `MatLab` is declared as a variable of type `Object`, `MLResReal` and `MLResImag` are declared as variables of type `Double`, and `MLStatus` is declared as a variable of type `String`.

```
Option Explicit
Option Base 1

Dim MatLab As Object

Dim MLResReal(1) As Double
Dim MLResImag(1) As Double
Dim MLStatus(1) As String
```

`MatLab` is the Visual Basic variable that represents the MATLAB Automation object; the actual assignment is made in the `Form_Load` procedure below. `MLResReal` and `MLResImag` are the Visual Basic variables that will contain the real and imaginary parts of MATLAB workspace variables returned by the `GetFullMatrix` Engine method. `MLStatus` is the `String` variable that will contain any error message returned from MATLAB.

Declaring these variables in the General Declarations section (outside of any Visual Basic functions) ensures that they do not lose scope during program execution, which is especially important for the MATLAB Automation object.

Form Load Procedure: Form_Load. The `Form_Load` procedure is executed as the Visual Basic program starts up.

```
Private Sub Form_Load()

Set MatLab = CreateObject("MatLab.Application")
MatLab.Execute ("myapp('init')")

End Sub
```

The MATLAB Automation object is now assigned to the `MatLab` variable, which causes MATLAB to launch. The copy of MATLAB that launches at this time is the copy that was most recently run as an Automation server on the system. If MATLAB is currently running as an Automation server on the system, the running MATLAB is used by the Automation controller.

The `MatLab.Execute` statement contains a function call to the top-level M-file, `myapp`. The string `"myapp('init')"` is evaluated in the MATLAB workspace,

and myapp is called with the argument 'init'. See “Initialization Function: myapp_init” on page 3-16 for more information about this function call.

Button Draw Procedure: btn_Draw_Click. The btn_Draw_Click procedure is executed when a button-click event occurs at the **Draw** button.

```
Private Sub btn_Draw_Click()

Dim MsgBoxText As String
Dim MsgBoxReturn As Integer

MatLab.Execute ("myapp('draw','" + txt_Input.Text + "')")
Call MatLab.GetFullMatrix("draw", "base", MLResReal, MLResImag)

If MLResReal(1) = 1 Then
   btn_Draw.Enabled = False
   btn_Erase.Enabled = True
Else
   MLStatus(1) = MatLab.Execute("disp(lasterr)")
   MsgBoxText = "Myapp cannot evaluate your input." _
                + Chr(13) & Chr(10) + _
                Left(MLStatus(1), (Len(MLStatus(1)) - 1))
   MsgBoxReturn = MsgBox(MsgBoxText, 0, "Error")
   btn_Draw.Enabled = True
   btn_Erase.Enabled = False
End If

End Sub
```

The first Execute command (line 6) evaluates the string “myapp('draw','input')” in the MATLAB workspace, where input is the expression the user enters in the **Plot** field. This calls myapp with two arguments. The first argument, 'draw', instructs myapp to pass the second argument, 'input', to function myapp_draw (see “Draw Function: myapp_draw” on page 3-17). myapp_draw attempts to plot the expression, and stores a value of 1 in the base workspace variable draw if it is successful. If plotting generates an error, myapp_draw stores a value of 0 in draw instead.

The GetFullMatrix statement then retrieves the value of draw into MLResReal; MLResImag remains empty because there is no imaginary component.

The If-Then-Else statement checks the value of MLResReal to determine the result of the plotting operation:

- If successful (`MLResReal=1`), the following statements change the state of the buttons accordingly.

  ```
  btn_Draw.Enabled = False
  btn_Erase.Enabled = True
  ```

- If an error occurred (`MLResReal=0`), the `Else` block performs the following operations:
  - Evaluate the string "`disp(lasterr)`" in the MATLAB workspace. Since the `disp` command here is invoked via the Engine's `Execute` method, its output is returned in variable `MLStatus` rather than being displayed in the command window. `MLStatus` therefore contains the text of the most recent MATLAB error message.
  - Compose an error message (`MsgBoxText`) based on the contents of `MLStatus`
  - Display the message in a Visual Basic message box
  - Change the state of the buttons appropriately

Button Erase Procedure: btn_Erase_Click. The `btn_Erase_Click` procedure is executed when a button-click event occurs at the **Erase** button. Its `Execute` command passes the `'erase'` argument to `myapp`, and again changes the state of the buttons depending on the value of `draw` in the workspace (see "Erase Function: myapp_erase" on page 3-17).

```
Private Sub btn_Erase_Click()

MatLab.Execute ("myapp('erase')")
Call MatLab.GetFullMatrix("draw", "base", MLResReal, MLResImag)

If MLResReal(1) = 1 Then
   btn_Draw.Enabled = False
   btn_Erase.Enabled = True
Else
   btn_Draw.Enabled = True
   btn_Erase.Enabled = False
End If

End Sub
```

Button Quit Procedure: btn_Quit_Click. The `btn_Quit_Click` procedure is executed when a button-click event occurs at the **Quit** button.

```
Private Sub btn_Quit_Click()

Form_Terminate

End Sub
```

The `Form_Terminate` function (see below) quits the Visual Basic application.

Menu Quit Procedure: menu_Quit_Click. The `menu_Quit_Click` procedure is executed when the **Quit** menu item is selected.

```
Private Sub menu_Quit_Click()

Form_Terminate

End Sub
```

The `Form_Terminate` function (see below) quits the Visual Basic application.

Form Terminate Procedure: Form_Terminate. The `Form_Terminate` procedure is executed when the Visual Basic GUI's Close box is clicked (or when this procedure is called from `btn_Quit_Click` or `menu_Quit_Click`). The `End` command quits the Visual Basic application.

```
Private Sub Form_Terminate()

End

End Sub
```

The MATLAB Runtime Server detects that the Automation controller has quit, and automatically quits as well.

Creating the Top-Level M-File

This example uses a top-level M-file, `myapp`, to act as an intermediary between the Visual Basic front end and the application's M-files (runtime P-files, when compiled). The contents of this function are shown below. When an event occurs in the GUI that requires a MATLAB function to execute, the GUI calls `myapp` with an appropriate argument indicating the action to be taken. This function is the *only* MATLAB function that this application's GUI calls directly.

```
function myapp(action,varargin)
% Top-level M-file for an Engine-based Visual Basic application.
```

```
switch action

case 'init'
   myapp_init

case 'draw'
   myapp_draw(varargin{:})

case 'erase'
   myapp_erase

end

% Other application M-files, not called from myapp.m
if 0
   matlabrt
end
```

This top-level M-file responds to three possible calls from the Visual Basic controller: `'init'`, `'draw'`, and `'erase'`. Depending on which of these `action` arguments Visual Basic calls it with, `myapp` executes one of three functions: `myapp_init`, `myapp_draw`, or `myapp_erase`. When `myapp` calls `myapp_draw`, it also passes the string in cell array `varargin`, which is the expression in the GUI's **Plot** field.

These three functions are discussed below. Note that the top-level M-file here does not create a GUI, since this has already been done by the Visual Basic front end.

One additional feature of the `myapp` function is the nonexecuting `if` statement containing a call to `matlabrt`. By including a dummy invocation of `matlabrt` in `myapp`, you can avoid having to use `buildp` or `depfun` on both files in order to analyze the entire application – you only need to apply those functions to `myapp.m`. The `if` statement is ignored at runtime.

Initialization Function: myapp_init. This function sets up the figure window.

```
function myapp_init
%MYAPP_INIT Initialize the Figure window.

fig = figure('Units','points','Position',[36 560 420 315],...
'Resize','off','HandleVisibility','on','Menubar','none',...
'NumberTitle','off','Name','Myapp',...
'CloseRequestFcn','close force');
```

```
ax = axes('Parent',fig,'Units','points',...
'Position',[42 50 335 230],'HandleVisibility','on','Box','on');
```

The `figure` command in the above code specifies a `CloseRequestFcn` for the figure and includes the `'Menubar','none'` specification to deactivate the default menus.

Draw Function: myapp_draw. This function evaluates and plots the expression contained in the `plottext` input argument.

```
function myapp_draw(plottext)
%MYAPP_DRAW Evaluate input string and plot in the Figure window.

Out = evalc('figure(findobj(''Name'',''Myapp''))',...
'myapp(''init'')');

try
   plot(eval(plottext));
   evalin('base','draw = 1;');
catch
   evalin('base','draw = 0;');
end
```

The `'try'` part of the `evalc('try','catch')` statement in the definition of `Out` makes the **Myapp** figure window active if it already exists. If it does not, the `'catch'` calls `myapp` with the `'init'` switch to create a new **Myapp** figure window.

The `try` block of code then attempts to plot the `plottext` expression (which contains the string in the GUI's **Plot** field). If the plotting operation is successful, the second line sets the value of the workspace variable `draw` to `1`. The Visual Basic application uses this as an indication that the expression was plotted. If an error occurs during plotting, the `catch` block of code sets the value of `draw` to `0`. The Visual Basic application uses this as an indication that an error occurred and the expression was not drawn; see "Button Draw Procedure: btn_Draw_Click" on page 3-13.

Erase Function: myapp_erase. This function clears the axes in the figure window.

```
function myapp_erase
%MYAPP_ERASE Erase the plot in the Figure window.

Out = evalc('figure(findobj(''Name'',''Myapp''))',...
```

```
'myapp(''init'')');

cla;

evalin('base','draw = 0;');
```

The `eval('try','catch')` statement is the same as that in function `myapp_draw`. After making **Myapp** the active figure window, the function clears the axes and sets the value of the workspace variable `draw` to `0`. The Visual Basic application uses this as an indication that the plot area was erased; see "Button Erase Procedure: btn_Erase_Click" on page 3-14.

Organizing Files and Managing Startup Tasks

This example places its own MATLAB files in `toolbox\mydir` during the entire development, compilation, and testing process. If you change any application file while exploring this example, use `rehash toolboxreset` to register the changes.

The Visual Basic files for this example can be in any convenient location.

The Startup Function

The example `matlabrt.m` file, which the section "MATLAB Startup Functions" on page 3-10 instructed you to install in the `toolbox\local` directory, is appropriate for this application because it:

- Does not display anything in the command window
- Does not set the global error behavior for the application. MATLAB errors do not suspend execution of a MATLAB runtime engine application; that is, the error behavior of the application is fixed in `'continue'` mode.
- Sets the warning level to `off`
- Invokes `pathdefrt` (not `pathdef`)
- Does not launch other MATLAB application functions because the front end of this application ultimately launches `myapp.p`

Creating the Path Definition Function

The example `pathdefrt.m` file that you installed in `toolbox\local` uses paths relative to the MATLAB root directory. Each distinct directory that contains a function used in this application is included on the runtime path.

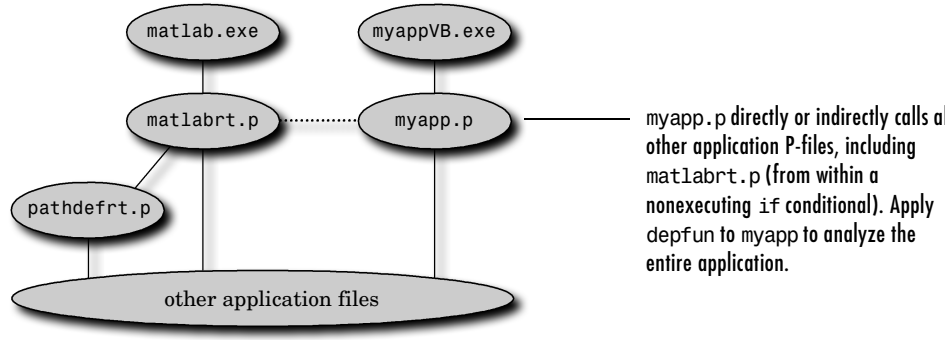
Notice that the list of directories in the example `pathdefrt.m` matches the list of (non-private, non-class) directories in the output of the `depdir` function.

```
cleanp; % Remove all .p files from path
p = depdir('myapp')

p = ['$toolbox/local:',...
     '$toolbox/mydir:',...
     '$toolbox/matlab/graphics:',...
     '$toolbox/matlab/uitools:',...
     '$toolbox/matlab/graph3d:',...
     '$toolbox/matlab/general:',...
     '$toolbox/matlab/lang:',...
     '$toolbox/matlab/specfun:',...
     '$toolbox/matlab/strfun:',...
     '$toolbox/matlab/ops:',...
     '$toolbox/matlab/datafun:',...
     '$toolbox/matlab/elmat:'];
```

Compiling the Application

Because this runtime engine application has a top-level M-file, `myapp`, that calls all other application M-files, it is easy to create all necessary P-files by applying the `buildp` function to `myapp`. The command below compiles the entire MATLAB portion of the application.

```
[log,depfunout,pcodeout] = buildp({'myapp'});
```



Testing with the Runtime Server Variant

To ensure that the application's front end launches the Runtime Server instead of commercial MATLAB, rename `matlabroot/bin/win32/license.dat`, which is the license file. Then launch MATLAB and start the front end of the application.

Exit the application by selecting **Quit** from the **File** menu, or by clicking the **Quit** button or Close box. This will quit both the Visual Basic application and MATLAB. Clicking the Close box in the figure window will only close that window, and will not quit the application.

When you want to use commercial MATLAB again, restore the license file.

Engine API Example

This example illustrates how to prepare an engine application for the MATLAB Runtime Server when the application uses the MATLAB Engine API library. The major steps in the procedure are:

- Creating the startup files `matlabrt.m` and `pathdefrt.m`
- Compiling the application with the `buildp` command
- Testing the application with the Runtime Server variant

The example application uses two of theMATLAB Engine API library commands, `eng Open` and `engClose`. These commands open and close, respectively, a communications link between the front-end C program and the back-end MATLAB engine. See *External Interfaces* in the MATLAB documentation set for more information about the Engine API library commands.

This section begins with instructions for installing the example files included in the software.

Preparing the Example Files

The example consists of these nine files, located in the `toolbox\runtime\examples\engineAPI` directory.

File	Description
`Readme_win`, `Readme_unix`	Platform-specific ASCII file with information about the example
`DrawAppWin.c`, `DrawApp.c`	Platform-specific C file
`myapp.m`	MATLAB Runtime Server application files
`myapp_draw.m`	
`myapp_erase.m`	
`myapp_error.m`	
`myapp_quit.m`	
`matlabrt.m`	MATLAB Runtime Server startup files
`pathdefrt.m`	

C File. Compile and link the C program, `DrawApp.c`, which is in the directory `toolbox\runtime\examples\engineAPI`. Keep the compiled file in the same directory.

For example, on Solaris machines, use these two commands at the system prompt.

```
setenv LD_LIBRARY_PATH matlabroot/extern/lib/sol2:$LD_LIBRARY_PATH
cc -Imatlabroot/extern/include -o DrawApp DrawApp.c
          -Lmatlabroot/extern/lib/sol2 -leng -lmx
```

Here, `matlabroot` is the directory in which MATLAB is installed.

As another example, on Windows NT 4.0 using Microsoft Visual C 6.0, execute the following commands at the MATLAB prompt.

```
opts = [matlabroot '\bin/win32\mexopts\msvc60engmatopts.bat'];
out = [matlabroot '\toolbox\runtime\examples\engineAPI'];
```

```
cfile = [matlabroot ...
'\toolbox\runtime\examples\engineAPI\DrawAppWin.c'];
mex('-f',opts,'-outdir',out,cfile);
```

MATLAB Startup Functions. Move `matlabrt.m` and `pathdefrt.m` into the `toolbox\local` directory of your development copy of MATLAB, alongside the `matlabrc.m` and `pathdef.m` files.

MATLAB Application Files. To prepare to use these files, start MATLAB and execute this command.

```
cd(fullfile(matlabroot,'toolbox','runtime','examples','engineAPI'))
```

Since the runtime P-files that you create will reside in the `toolbox\runtime\examples\engineAPI` directory, it is important that this directory is *either*:

- The one from which MATLAB is launched, or
- Part of the runtime path

In this example, `toolbox\runtime\examples\engineAPI` is the directory from which the C program will launch MATLAB.

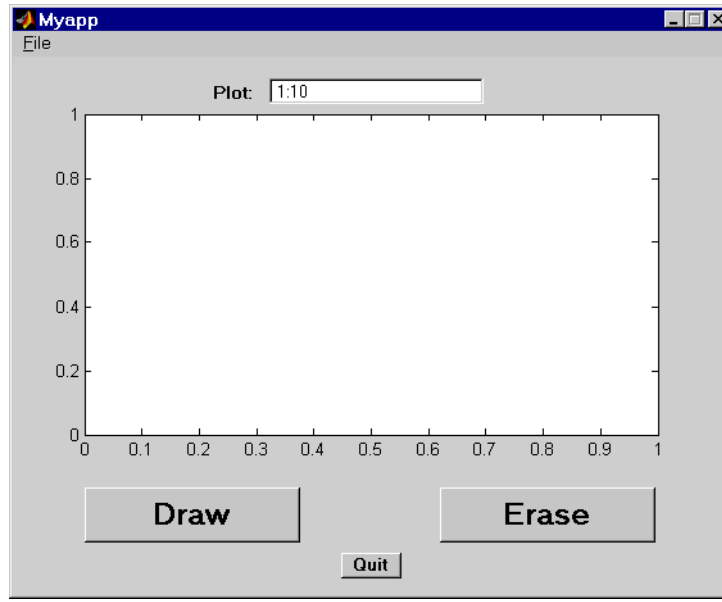
On PC platforms, you should also modify the DOS path so that MATLAB can find all the files it needs. At the DOS prompt, type

```
path = matlabroot/bin/win32;%path%
```

where `matlabroot` is the directory in which MATLAB is installed.

Adapting the Design for Runtime Execution

This example is a simple GUI-driven application that evaluates and plots a MATLAB expression. Its GUI is shown below.



Note Quitting the `myapp` application by any of the means provided on the GUI (**Quit** button, **Quit** menu item, or Close box) causes MATLAB to quit. To exit the application without exiting MATLAB, type `close force` at the command line.

To use the application, enter a valid MATLAB expression in the **Plot** field of the GUI and press the **Draw** button. The application plots the expression in its GUI. To erase the plot, press the **Erase** button.

Overview of Adaptations

`myapp.m` incorporates several adaptations for Runtime Server execution:

- Its `figure` command includes the `'Menubar','none'` specification. This deactivates the GUI's default menus, as discussed in "Preventing Command Window Input/Output" on page 1-2.
- The call to `myapp_draw.m` is embedded within an `eval(`*`try`*`,`*`catch`*`)` statement in order to trap possible errors. If MATLAB encounters an error while trying to evaluate or plot the expression in `myapp_draw`, then it executes the *`catch`* expression, which calls `myapp_error.m` to display an appropriate error message.
- The figure window definition specifies a `CloseRequestFcn`. Since the **Myapp** window is the top-level GUI figure window, its `CloseRequestFcn` exits the application by calling `myapp_quit.m`.

Also, the C program for this example is adapted for Runtime Server execution because its `engOpen` command uses one of the flags

```
-runtime (PC)
-runtime -c dummy (UNIX)
```

when invoking MATLAB.

Organizing Files and Managing Startup Tasks

This example's own MATLAB files are in `toolbox\runtime\examples\engineAPI` during the entire development, compilation, and testing process. If you change any application file while exploring this example, use `rehash toolboxreset` to register the changes.

The C files for this example are in `toolbox\runtime\examples\engineAPI`.

The Startup Function

The `matlabrt.m` file that you moved from the directory `toolbox\runtime\examples\engineAPI` is similar to the one described in Chapter 2, "Developing a MATLAB Runtime GUI Application." See the GUI example section "The Startup Function" on page 2-21.

Creating the Path Definition Function

The example `pathdefrt.m` file that you moved into `toolbox\local` for this example uses paths relative to the MATLAB root directory. Each distinct directory that contains a function used in this application is included on the runtime path.

Notice that the list of directories in the example `pathdefrt.m` matches the list of (non-private, non-class) directories in the output of the `depdir` function.

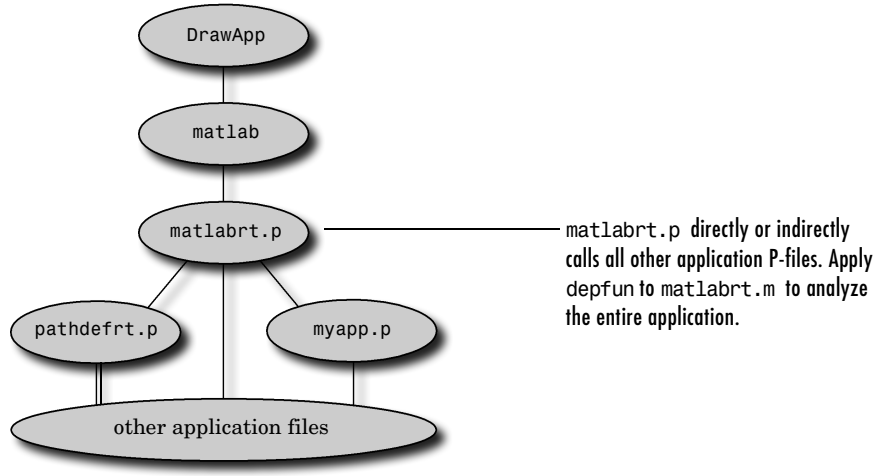
```
cleanp; % Remove all .p files from path
p = depdir('matlabrt')

p =

    'C:\matlab\toolbox\local'
    'C:\matlab\toolbox\matlab\datafun'
    'C:\matlab\toolbox\matlab\datafun\@cell'
    'C:\matlab\toolbox\matlab\datatypes'
    'C:\matlab\toolbox\matlab\elmat'
    'C:\matlab\toolbox\matlab\general'
    'C:\matlab\toolbox\matlab\general\@char'
    'C:\matlab\toolbox\matlab\graph3d'
    'C:\matlab\toolbox\matlab\graphics'
    'C:\matlab\toolbox\matlab\graphics\private'
    'C:\matlab\toolbox\matlab\iofun'
    'C:\matlab\toolbox\matlab\lang'
    'C:\matlab\toolbox\matlab\ops'
    'C:\matlab\toolbox\matlab\ops\@cell'
    'C:\matlab\toolbox\matlab\specfun'
    'C:\matlab\toolbox\matlab\strfun'
    'C:\matlab\toolbox\matlab\strfun\@cell'
    'C:\matlab\toolbox\matlab\uitools'
    'C:\matlab\toolbox\runtime\examples\engineAPI'
```

Compiling the Application

Because `matlabrt` directly or indirectly calls all other M-files in this runtime application, you can compile all application M-files with a single `buildp` command.

```
[log,depfunout,pcodeout] = buildp({'matlabrt'});
```

The figure below indicates schematically which parts of this application invoke which other parts.



Testing with the Runtime Server Variant

To test the application by using the Runtime Server variant, follow these steps:

1. (*PC only*) Rename the license file, which is `matlabroot\bin\win32\license.dat`

2. Start the MATLAB Runtime Server by using this command at the system prompt.

   ```
   matlab -runtime (PC)
   matlab -runtime -c dummy (UNIX)
   ```

3. From the system prompt, navigate to the `toolbox\runtime\examples\engineAPI` directory

4. Type `DrawAppWin` (*PC*), or `DrawApp` (*UNIX*)

The C program displays this message in the Xterm or DOS window.

```
Press Enter or Return to exit engine application.
```

The program also invokes MATLAB, which in turn invokes `myapp.p` and the application's other P-files as appropriate.

You should see the application's GUI and figure window. Test the application by typing expressions in the **Plot** field and pressing the buttons. To quit the `DrawApp` application, activate the system window from which you started `DrawApp`, and press the **Enter** or **Return** key.

Remember to restore the license file when you want to use commercial MATLAB.

Summary List: MATLAB Runtime Engine Application

This list summarizes the steps for creating a MATLAB runtime engine application. The hyperlinks lead to the sections that describe the steps in more detail. Note that some links point to the chapter about MATLAB runtime GUI applications because the topic is the same for both types of application.

1. Stamp your development copy of MATLAB.

2. Adapt the design of the MATLAB portion of the application as necessary. See Chapter 1, "Design Issues for a Runtime Application."

3. Decide where to place the MATLAB files you write for the runtime application.
   - If you keep all files under `toolbox`, then remember to use `rehash toolboxreset` to register each change you make.
   - If you keep the M-files outside of `toolbox` and move the P-files under `toolbox` only towards the end of your development and testing process, then remember to include the final destination directories in `pathdefrt`.

   MATLAB places no restriction on the directory structure of the non-MATLAB portion of your application and no restriction on the relative locations of the MATLAB and non-MATLAB portions of your application.

4. Create the startup function `matlabrt.m` by copying `toolbox\runtime\matlabrt_template.m` to `toolbox\local` and renaming it `matlabrt.m`. Then modify it as necessary and use `rehash toolboxreset` to register the change.

   Note that PC applications have special startup considerations arising from multiple versions or instances of MATLAB.

5. Create the path definition function `pathdefrt.m` by copying `toolbox\runtime\pathdefrt_template.m` to `toolbox\local` and renaming it `pathdefrt.m`. Then modify it as necessary and use `rehash toolboxreset` to register the change.
   - Use `cleanp` to remove any `.p` files from your path and current directory.
   - Use `depdir` to determine the runtime path.

- If you plan to move your P-files under `toolbox` later on, then include their destination directories in the runtime path.
- Place `pathdefrt.m` in `toolbox\local` and use `rehash toolboxreset` to register the change.

6 Use `buildp` to compile all application M-files into runtime P-files.

- Possible complications: toolbar or menu items, functions invoked by `eval`, MATLAB P-files, MATLAB add-on products
- Also see the `buildp` log file for information about the compilation process

7 Test the application.

- With your P-files in their final locations under `toolbox`, if they are not already there. After moving them, use `rehash toolboxreset` to register the changes.
- Test the MATLAB portion by having MATLAB emulate the Runtime Server (different procedures for ActiveX and Engine API applications).
- Test the entire application with the Runtime Server variant.

Return to earlier steps as necessary. When debugging, remember to stay current by deleting old P-files, producing new P-files whenever an M-file changes, and using `rehash toolboxreset` as necessary to register changes.

4

Shipping a MATLAB Runtime Application

Shipping a MATLAB Runtime Application

This chapter includes some final considerations for packaging and shipping a professional-looking product.

Splash Screen

A MATLAB runtime GUI application can display a custom splash screen at startup. The splash screen image is shown for a few seconds, and typically contains the product name and logo, company name, and copyright information.

Creating the Splash Screen

You can create a splash screen with the MATLAB `imwrite` function, or with any graphics application that can save an image file in the BMP format (8-bit). Name the splash screen file `splash.bmp`, and place it in either the `toolbox\local` directory or the startup directory. An example `splash.bmp` file is included in the `toolbox\runtime` directory. Note that the amount of space that the splash screen occupies on the user's display depends on the resolution of the device. Note also that on both UNIX and PC platforms, if the splash screen has an 8-bit image (256 colors or fewer), then it will still look right even if your users have only 256 colors available.

Note The MATLAB Runtime Server does not display a splash screen if neither the `toolbox\local` directory nor the startup directory contains a `splash.bmp` file.

Organizing Files for Shipping

Your MATLAB files must reside in some directory underneath the MATLAB `toolbox` directory. If you kept them somewhere else while developing the application, then you must move them under `toolbox` before you package the application for shipping.

A large MATLAB runtime engine application might have non-MATLAB application files (for example, Visual Basic files, libraries, or data files) that require a particular organizational structure. See the documentation for your development environment to find out more about organizing those files. Note

that the two file sets are generally independent. For example, an application's Visual Basic files do not need to know where the application's MATLAB P-files are, and vice versa, because the Automation controller locates the Automation server by checking the Windows Registry instead of by searching a predefined file path.

Automatically Packaging Files for Shipping

The Runtime Server software includes a packaging utility that can automatically archive your application files and create all the executable files that your users need in order to install your application on their machines.

Note The packaging utility is intended to be used only with the MATLAB portion of your application, however, you can integrate the packaging utility into your overall installation suite.

The packaging utility is called `package` and is located in the `toolbox\runtime\oem` directory. When you execute `package`, it prompts you for all necessary information. It reports all relevant information to you, including the files it creates and the instructions that you should follow in order to test the installation. These are the same instructions that end users need to follow in order to install your application. "Installing and Running the Application" on page 4-7 contains more information about the installer software.

The rest of this section contains more detail for PC and UNIX versions of `package`.

Packaging Utility on PC

After you have used `buildp` to create your P-files, follow these steps to run the automatic packaging utility on PC platforms:

1. Use `makeconfig` to create a *configuration file*, which is a text file that lists the names of all files that your application uses. This file includes runtime P-files and any MEX-files on which your application depends. It should not include MATLAB and files that MATLAB uses to run.

Note If `buildp` or `depfun` indicates that your application uses Java classes, then you must edit the configuration file manually to include: `matlabroot\toolbox\local\classpath.txt`, `matlabroot\sys\java\jre\win32\jre\*`, as well as whatever files your application requires from the `matlabroot\java\jar` and `matlabroot\java\jarext` hierarchies of directories.

A sample configuration file is `toolbox\runtime\oem\user.config_pc`. Notice from the sample `user.config_pc` file that the path of each listed file starts from `toolbox` (for example, `toolbox\local\matlabrt.p`).

The configuration file can contain `*` as a wildcard, `#` to preface comments, and blank lines to improve readability.

2 Go to the system prompt and navigate to the `toolbox\runtime\oem` directory. Run the packaging utility using one of the commands below.

`package` (if your application does not use Java classes)

`package -java` (if your application uses Java classes)

The packaging utility prompts you for:

a The path to your stamped MATLAB root directory

b The path of your configuration file

c A name for your application. This is the name of the batch file that the automatic installer creates when your end users install your application. If you do not specify a name, then the batch file is called `demo_app.bat`.

Alternatively, the syntaxes below eliminate the need for prompts.

```
package -r matlabroot -c config_file -n app_name
```

```
package -java -r matlabroot -c config_file -n app_name
```

The packaging utility produces a file called `setup.exe` in the directory from which it was launched. This is the file you should ship to end users.

Packaging Utility on UNIX

After you have used `buildp` to create your P-files, follow these steps to run the automatic packaging utility on UNIX platforms:

1. Use `makeconfig` to create a *configuration file*, which is a text file that lists the names of all files that your application uses. This file includes runtime P-files and any MEX-files on which your application depends. It should not include MATLAB and files that MATLAB uses to run.

Note If `buildp` or `depfun` indicates that your application uses Java classes, then you must edit the configuration file manually to include: `matlabroot/toolbox/local/classpath.txt`; `matlabroot/sys/java/jre/`*`arch`*`/jre` and the directory that this symbolic link points to; and whatever files your application requires from the `matlabroot/java/jar` and `matlabroot/java/jarext` hierarchies of directories. Here, *`arch`* is an architecture-specific directory name.

A sample configuration file is `toolbox/runtime/oem/user.files`. The format for lines in this configuration file is

```
name_of_source_file   location_of_source_file   target_location
```

where `target_location` refers to the destination directory in the end user's installation, relative to the directory from which the user runs the installer program.

In your configuration file, you can use `$SCRIPTDIR` to indicate the directory location of the `package` script and `$MATLAB` to indicate the MATLAB root directory. You can also use Bourne shell wildcards in filenames.

Note If you use MEX-files on multiple UNIX platforms, then you must package the files that are appropriate for each of the platforms. For example, if you use `buildp` on a Solaris machine and the `depfunout` output indicates that your application uses `myapp.mexsol`, then your shipping version for Linux users must include `myapp.mexglx`. In the configuration file for the packaging utility, you can use wildcards, as in `myapp.mex*`, to avoid omitting any files.

2 Run the script

```
toolbox/runtime/oem/package
```

using the `-java` switch if your application uses Java classes. The script prompts you for the MATLAB root directory, the architecture, and the name of your configuration file.

For information on command line arguments you can use to prevent prompting while running the package utility, see

```
package -help
```

The `package` script produces two files:

- `app.tar.files`, which lists the files in the application
- `app.tar`, which is a tar file made from the application

You should ship `app.tar` to end users. For more details about the UNIX packaging utility, see the `README` file in `toolbox/runtime/oem`.

Manually Packaging Files for Shipping (PC)

On PC platforms, you can archive your files either by using the provided automatic packaging utility, or by copying them manually. If you copy them manually, then you should ship the files whose names are listed in:

- `matlabroot\toolbox\runtime\oem\bin\win32\binaries.lst`
- `matlabroot\toolbox\runtime\oem\bin\win32\stubs.lst`

Do not modify `binaries.lst` or `stubs.lst` if you expect to use the automatic packaging utility in the future.

Of course, you also need to ship the application-specific files that you wrote.

Additional Files for Applications Using Java

If `buildp` or `depfun` indicates that your application uses Java classes, then you must also include these files among the files that you ship and that your installer copies:

- `matlabroot\toolbox\local\classpath.txt`
- `matlabroot\sys\java\jre\win32\jre\*`
- Whatever files your application requires from the `matlabroot\java\jar` and `matlabroot\java\jarext` hierarchies of directories

Installing and Running the Application

This section discusses the installers provided on PC and UNIX platforms, tips for building an installer yourself for PC applications, and other special considerations for PC applications.

Automatically Built Installer on PC

If you use `package` to organize your files for shipping, then it automatically builds an installer for your end user. To install the application in a given directory, copy `setup.exe` into that directory and execute `setup.exe`. This creates a batch file whose name is the application name that you specified when you first used `package`. To launch the application, type the application name.

Sample Installer on UNIX

When you use `package` to organize your files for shipping, your `user.files` listing includes a file called `user.install`. This file is a sample installer for your end users. If your application has any special installation or execution requirements, then you might need to modify `user.install` accordingly. When your end users want to install and run your application, they should follow these instructions:

1. Unpackage the application with these commands.

```
mkdir prog
cd prog
tar -xvf ../app.tar
```

2 Run the installer program with this command.

```
user.install '/bin/pwd'
```

3 Run the application with this command.

```
user.app
```

`user.app` requires that `DISPLAY` be set. Alternatively, pass the `DISPLAY` value as an argument.

The sample files `user.install` and `user.app` are in `toolbox/runtime/oem`. For more details, see the `README` file in `toolbox/runtime/oem`.

Manually Building an Installer on PC

If you copied and organized your files manually, then you need to create an installer for your application. The key consideration for the installation process is that the Runtime Server should be able to locate the application files when it is launched. To be certain of this:

- Make sure that your application directories are on the runtime path, as discussed in the section "Creating the Path Definition Function" on page 2-4.
- Make sure that the Runtime Server can find `matlabrt.p` and `pathdefrt.p`. These startup files need to be in the `toolbox\local` directory or the current directory.

Associating Files with the MATLAB Runtime Server on PC

In addition to installing the files, the installer software can associate the application files with the Runtime Server executable. This allows the user to launch the application (without explicitly launching the Runtime Server) by clicking on an application file icon or typing the application name.

To create custom icons for the application files, the installer should register the file extensions of all related files to open with the Runtime Server. Clicking on any file will then launch the Runtime Server and start the application. You might also want the installer to create a shortcut to launch the application.

Registering MATLAB as an Automation Server on PC

If you are shipping a MATLAB runtime engine application for PC platforms, then the Runtime Server needs to be registered in the Microsoft Windows Registry on the user's computer as an *Automation Server*. The Runtime Server

will register itself with this designation each time it is run with the `/Automation` flag.

To register the Runtime Server, your installer software should run the Runtime Server executable once as the last step of the installation. Because the Runtime Server cannot be launched through the Engine API commands until it is registered, the installer needs to launch the Runtime Server executable explicitly with the `/Automation` flag.

During this *initial run,* the application's `matlabrt.p` file needs to provide a mechanism for the Runtime Server to quit. You can build in this facility by making one change to the `matlabrt` file, and by installing an extra file in the `toolbox\local` directory.

Follow these steps:

1. Add the following lines at the beginning of the `matlabrt.m` file, right after the function header.

   ```
   if exist('register.txt') == 2
     delete('register.txt');
     quit;
   end
   ```

   This causes the Runtime Server to look on the path for a file called `register.txt`. If the file exists, the Runtime Server deletes the file and quits. This brief execution is enough for the Runtime Server to register itself as an Automation server.

2. Create the file `register.txt`, or instruct the installer software to create it. It should be installed in the `toolbox\local` directory (along with `matlabrt.p`). It will be deleted when the installer initially runs the Runtime Server, so the contents of the file are not important.

Final Testing

Finally, test the installed application with the Runtime Server. For complete accuracy, you should test the application, including installation, with the same platform/installation that the end user has. You should test the application on machines that do not have any copies of MATLAB installed, as well as those that do. Launch the application while running a commercial copy of MATLAB

on the same system, as well as before and after, to make sure that the application runs with the Runtime Server under all circumstances.

5

Function Reference

Functions — By Category

The tables below list the functions commonly used in the development of runtime applications. Some functions, such as `buildp`, are part of the Runtime Server and reside in the `toolbox\runtime` directory. Others, such as `inmem`, are standard MATLAB functions.

General Tools

`isruntime`	True if MATLAB is the Runtime Server or is emulating the Runtime Server
`runtime`	Emulate the runtime environment in MATLAB and set the global error mode

P-Code Generation Tools

`buildp`	Generate runtime P-code for application
`cleanp`	Delete all P-files on the path and in the current directory
`depdir`	List the dependent directories of an M-file or P-file
`depfun`	List the dependent functions of an M-file or P-file
`inmem`	List functions in memory
`pcode`	Create a preparsed pseudocode file (P-file)
`pcodeall`	Compile all M-files on the path and current directory into P-files

Utilities

`dirlist`	List all files in directories on the current path
`makeconfig`	Generates a configuration file for application packaging

Functions — Alphabetical List

buildp

Purpose

Generate runtime P-code for application

Syntax

```
[log,depfunout,pcodeout] = buildp(files);
[log,depfunout,pcodeout] = buildp(files,'develStruct','',verbose);
```

Description

The `buildp` function generates runtime P-code for a runtime application based on the filenames listed in `files`. The outputs report status and possible problems.

`[log,depfunout,pcodeout] = buildp(files)` determines the direct and indirect dependencies of the filenames listed in the cell array `files`. For each subordinate M-file, this command generates a runtime P-file and places it in the same directory with the M-file.

`[log,depfunout,pcodeout] = buildp(files,'develStruct','',verbose)` is the same as the first syntax, except that if `verbose` is 1, then `buildp` sends output to the command window.

Inputs

`files` is a cell array of strings. Each string is the name of a file that is part of your runtime application. If one function in your application depends on another, then you do not need to list the subordinate function when invoking `buildp`.

If `verbose` is 1, then `buildp` sends output to the command window; if `verbose` is 0 or absent, then `buildp` suppresses such output.

Outputs

The output `log` is a string containing the name of a file that details the various phases of execution of `buildp`.

The outputs `depfunout` and `pcodeout` are cell arrays that give information about what happens when `buildp` invokes `depfun` and `pcode` while executing. `depfunout` contains the output from `depfun`, which indicates possible problems finding or parsing the runtime application's files. `pcodeout` indicates whether `buildp` had problems creating P-files for the application.

These outputs are useful for troubleshooting, and for checking whether you need to consider functions that your application invokes via an `eval` command.

Note If your application uses toolbar items from the MATLAB default figure window, then you must include `'FigureToolBar.fig'` in your input to `buildp`. If your application uses menu items from the MATLAB default figure window, then you must include `'FigureMenuBar.fig'` in your input to `buildp`.

Example

The command

```
[log,depfunout,pcodeout] = buildp({'matlabrt'});
```

creates P-code for the runtime application whose top-level file is `matlabrt`.

Suppose the log file from the command above indicates that some file in the runtime application includes an `evalc` command. If you examine the instance and determine that the command might invoke either of the files `comp1` and `comp2`, then you can issue this second `buildp` command to complete the building of the runtime application.

```
[log,depfunout,pcodeout] = buildp({'comp1','comp2'});
```

Example

The command

```
[log,depfunout,pcodeout] = buildp({'startup\matlabrt',...
'mainfiles\myapp'});
```

creates P-code for a runtime application whose top-level files are the two independent files `startup\matlabrt` and `mainfiles\myapp`. (If `matlabrt` invokes `myapp`, then you do not need to list `myapp` in the call to `buildp`. Similarly, if `myapp` invokes `matlabrt`, then you do not need to list `matlabrt` in the call to `buildp`.)

See Also

`depfun`, `pcode`

cleanp

Purpose

Delete all P-files on the path and in the current directory

Syntax

```
cleanp
```

Description

`cleanp` deletes all files whose names end with `.p` on the current MATLAB path and in the current directory. This *includes* the `toolbox/runtime` directory if it is the current directory or on the path. This also includes private and class directories whose parent directories are either the current directory or on the MATLAB path. Use this command to remove P-files generated with the `pcodeall` or `pcode` command.

Caution `cleanp` looks only at names, not contents, of files. Use caution if you have files other than MATLAB P-files that use a `.p` filename extension.

See Also

`pcode`, `pcodeall`, `rehash`

Purpose

List the dependent directories of an M-file or P-file

Syntax

```
list = depdir('file_name');
[list,prob_files,prob_sym,prob_strings] = depdir('file_name');
[...] = depdir('file_name1','file_name2',...);
```

Description

The `depdir` function lists the directories of all of the functions that a specified M-file or P-file needs to operate. This function is useful for finding all of the directories that need to be included with a runtime application and for determining the runtime path.

`list = depdir('file_name')` creates a cell array of strings containing the directories of all the M-files and P-files that `file_name.m` or `file_name.p` uses. This includes the second-level files that are called directly by `file_name`, as well as the third-level files that are called by the second-level files, and so on.

`[list,prob_files,prob_sym,prob_strings] = depdir('file_name')` creates three additional cell arrays containing information about any problems with the `depdir` search. `prob_files` contains filenames that `depdir` was unable to parse. `prob_sym` contains symbols that `depdir` was unable to find. `prob_strings` contains callback strings that `depdir` was unable to parse.

`[...] = depdir('file_name1','file_name2',...)` performs the same operation for multiple files. The dependent directories of all files are listed together in the output cell arrays.

Example

```
list = depdir('mesh')
```

See Also

`depfun`

depfun

Purpose

List the dependent functions of an M-file or P-file

Syntax

```
list = depfun('file_name');
[list,builtins,classes] = depfun('file_name');
[list,builtins,classes,prob_files,prob_sym,eval_strings,...
   called_from,java_classes] = depfun('file_name');
[...] = depfun('file_name1','file_name2',...);
[...] = depfun('fig_file_name');
[...] = depfun(...,'-toponly');
```

Description

The `depfun` function lists all of the functions and scripts, as well as built-in functions, that a specified M-file needs to operate. This is useful for finding all of the M-files that you need to compile for a MATLAB runtime application.

`list = depfun('file_name')` creates a cell array of strings containing the paths of all the files that `file_name.m` uses. This includes the second-level files that are called directly by `file_name.m`, as well as the third-level files that are called by the second-level files, and so on.

Note If `depfun` reports that "`These files could not be parsed:`" or if the `prob_files` output below is nonempty, then the rest of the output of `depfun` might be incomplete. You should correct the problematic files and invoke `depfun` again.

`[list,builtins,classes] = depfun('file_name')` creates three cell arrays containing information about dependent functions. `list` contains the paths of all the files that `file_name` and its subordinates use. `builtins` contains the built-in functions that `file_name` and its subordinates use. `classes` contains the MATLAB classes that `file_name` and its subordinates use.

`[list,builtins,classes,prob_files,prob_sym,eval_strings,... called_from,java_classes] = depfun('file_name')` creates additional cell arrays or structure arrays containing information about any problems with the `depfun` search and about where the functions in `list` are invoked. The additional outputs are:

- `prob_files`, which indicates which files `depfun` was unable to parse, find, or access. Parsing problems can arise from MATLAB syntax errors. `prob_files` is a structure array whose fields are:
  - `name`, which gives the names of the files
  - `listindex`, which tells where the files appeared in `list`
  - `errmsg`, which describes the problems
- `prob_sym`, which indicates which symbols `depfun` was unable to resolve as functions or variables. It is a structure array whose fields are:
  - `fcn_id`, which tells where the files appeared in `list`
  - `name`, which gives the names of the problematic symbols
- `eval_strings`, which indicates usage of these evaluation functions: `eval`, `evalc`, `evalin`, `feval`. When preparing a runtime application, you should examine this output to determine whether an evaluation function invokes a function that does not appear in `list`. The output `eval_strings` is a structure array whose fields are:
  - `fcn_name`, which give the names of the files that use evaluation functions
  - `lineno`, which gives the line numbers in the files where the evaluation functions appear
- `called_from`, a cell array of the same length as `list`. This cell array is arranged so that `list(called_from{`*`i`*`})` returns all functions in `file_name` that invoke the function `list{`*`i`*`}`.
- `java_classes`, a cell array of Java class names that `file_name` and its subordinates use

`[...] = depfun('file_name1','file_name2',...)` performs the same operation for multiple files. The dependent functions of all files are listed together in the output arrays.

`[...] = depfun('fig_file_name')` looks for dependent functions among the callback strings of the GUI elements that are defined in the `.fig` or `.mat` file named `fig_file_name`.

`[...] = depfun(...,'-toponly')` differs from the other syntaxes of `depfun` in that it examines *only* the files listed explicitly as input arguments. It does not examine the files on which they depend. In this syntax, the flag `'-toponly'` must be the last input argument.

Notes

1. If `depfun` does not find a file called `hginfo.mat` on the path, then it creates one. This file contains information about Handle Graphics callbacks.
2. If your application uses toolbar items from the MATLAB default figure window, then you must include `'FigureToolBar.fig'` in your input to `depfun`.
3. If your application uses menu items from the MATLAB default figure window, then you must include `'FigureMenuBar.fig'` in your input to `depfun`.
4. Because many built-in Handle Graphics functions invoke `newplot`, the list produced by `depfun` always includes the functions on which `newplot` is dependent:
   - `'matlabroot\toolbox\matlab\graphics\newplot.m'`
   - `'matlabroot\toolbox\matlab\graphics\closereq.m'`
   - `'matlabroot\toolbox\matlab\graphics\gcf.m'`
   - `'matlabroot\toolbox\matlab\graphics\gca.m'`
   - `'matlabroot\toolbox\matlab\graphics\private\clo.m'`
   - `'matlabroot\toolbox\matlab\general\@char\delete.m'`
   - `'matlabroot\toolbox\matlab\lang\nargchk.m'`
   - `'matlabroot\toolbox\matlab\uitools\allchild.m'`
   - `'matlabroot\toolbox\matlab\ops\setdiff.m'`
   - `'matlabroot\toolbox\matlab\ops\@cell\setdiff.m'`
   - `'matlabroot\toolbox\matlab\iofun\filesep.m'`
   - `'matlabroot\toolbox\matlab\ops\unique.m'`
   - `'matlabroot\toolbox\matlab\elmat\repmat.m'`
   - `'matlabroot\toolbox\matlab\datafun\sortrows.m'`
   - `'matlabroot\toolbox\matlab\strfun\deblank.m'`
   - `'matlabroot\toolbox\matlab\ops\@cell\unique.m'`
   - `'matlabroot\toolbox\matlab\strfun\@cell\deblank.m'`
   - `'matlabroot\toolbox\matlab\datafun\@cell\sort.m'`
   - `'matlabroot\toolbox\matlab\strfun\cellstr.m'`
   - `'matlabroot\toolbox\matlab\datatypes\iscell.m'`
   - `'matlabroot\toolbox\matlab\strfun\iscellstr.m'`
   - `'matlabroot\toolbox\matlab\datatypes\cellfun.dll'`

Example

```
list = depfun('mesh'); % Files mesh.m depends on
```

```
list = depfun('mesh','-toponly') % Files mesh.m invokes directly
[list,builtins,classes] = depfun('gca');
```

See Also

depdir, dirlist

dirlist

Purpose

List all files in directories on the current path

Syntax

```
[w,wc,wp] = dirlist;
```

Description

`[w,wc,wp] = dirlist` returns three structure arrays containing information about all of the directories on the path and in the current directory. Each structure array (`w`, `wc`, and `wp`) has the same fields as the structure array produced by the MATLAB `what` command.

Field	Description
`path`	The directory path (a string)
`m`	The names of the M-files in the directory (a cell array of strings)
`mat`	The names of the MAT-files in the directory (a cell array of strings)
`mex`	The names of the MEX-files in the directory (a cell array of strings)
`mdl`	The names of the Simulink model files in the directory (a cell array of strings)
`p`	The names of the P-files in the directory (a cell array of strings)
`classes`	The names of the class directories in the directory (a cell array of strings)

`w` provides this information for every *standard* directory on the path and in the current directory. `wc` provides this information for every *class* directory on the path and in the current directory. `wp` provides this information for every *private* directory on the path and in the current directory.

The length of `w` is the same as the number of standard directories, the length of `wc` is the same as the number of class directories, and the length of `wp` is the same as the number of private directories.

Example

```
[w,wc,wp] = dirlist;
```

To see the `dirlist` information for private directories on the path, look at `wp`. The output shown below is for a typical MATLAB installation.

```
wp =
14x1 struct array with fields:
    path
    m
    mat
    mex
    mdl
    p
    classes
```

This means that there are 14 private directories on the path and in the current directory. To see what the third one is, type

```
wp(3)
ans =
       path: 'c:\matlab\toolbox\matlab\uitools\private'
          m: {12x1  cell}
        mat: { 2x1  cell}
        mex: { 1x1  cell}
        mdl: {}
          p: {12x1  cell}
    classes: {}
```

The directory is `toolbox\matlab\uitools\private`, and it contains 12 M-files, 2 MAT-files, 1 MEX-file, and 12 P-files. There are no Simulink model files or class directories in the `uitools\private` directory. To see the MAT-files in `uitools\private`, look at `wp(3).mat`.

```
wp(3).mat
ans =
    'algnbtn.mat'
    'toolbtn.mat'
```

See Also

`depdir`, `depfun`

inmem

Purpose

List functions in memory

Syntax

```
M = inmem;
[M,mexfiles] = inmem;
[M,mexfiles,jclasses] = inmem;
```

Description

`M = inmem` returns a cell array of strings containing the names of the M-files that are in the P-code buffer.

`[M,mexfiles] = inmem` returns also a cell array, `mexfiles`, containing the names of the MEX-files that are currently loaded.

`[M,mexfiles,jclasses] = inmem` returns also a cell array, `jclasses`, containing the names of the Java classes that are currently loaded.

Example

```
clear functions
sqrtm([1 0;0 1]);
M = inmem

M =
    'rsf2csf'
    'sqrtm'
```

These are the M-files that were required to run `sqrtm`.

See Also

`depdir`, `depfun`, `rehash`

Purpose

True if MATLAB is the Runtime Server or is emulating the Runtime Server

Syntax

```
isruntime
```

Description

`isruntime` returns logical true (`1`) if MATLAB is either commercial MATLAB currently emulating the Runtime Server, or the Runtime Server variant. `isruntime` returns logical false (`0`) otherwise.

Example

```
runtime on
isruntime

ans =
     1

runtime off
isruntime

ans =
     0
```

See Also

`runtime`

makeconfig

Purpose

Generate a configuration file for packaging an application

Syntax

```
makeconfig(list)
makeconfig(list,filename)
makeconfig(list,filename,list2)
```

Description

`makeconfig(list)` creates the configuration file `user.config_pc` on a PC or `user.files` on a UNIX machine. This file is used by the packaging utility. `list` is single column, cell array where each cell contains a string with the complete filename to be included in the runtime application. Each filename must begin with the equivalent of your `matlabroot` directory, similar to the output of `depfun`. Any duplicate filenames are ignored.

`makeconfig(list,filename)` writes the configuration file to the specified `filename`.

`makeconfig(list,filename,list2)` writes the configuration files to the specified filename and appends `list2` to `list`. `list2` contains any additional files that you want to ship that are not included in `list`, such as an optional splash screen bitmap.

Note If your files contain calls to Java classes, you must package the files needed for these classes to run. See Automatically Packaging Files for Shipping on page 4-3 for more information.

Example

Generate a configuration file.

1. Create the `matlabroot` pathname and change to that directory.
2. Use buildp to create the application and capture the dependencies in `depfunout`
3. Create `list2` which contains paths to `matlabrt.p` and `pathdefrt.p` files.
4. Run `makeconfig`.

The code to execute these steps is

```
cd(fullfile(matlabroot,'toolbox','runtime','examples','gui'));
[log, depfunout] = buildp({'matlabrt'});
```

```
list2 = {fullfile(matlabroot,'toolbox','local','matlabrt.p'),...
   fullfile(matlabroot,'toolbox','local','pathdefrt.p')};
makeconfig(depfunout{1},'configfile',list2);
```

See Also

depfun

pcode

Purpose

Create a preparsed pseudocode file (P-file)

Syntax

```
pcode fun
pcode *.m
pcode fun1 fun2 ...
pcode ... -inplace
pcode ... -runtime
pcode ... -inplace -runtime
```

Description

`pcode fun` parses the M-file `fun.m` into the P-file `fun.p`. If `fun.m` is not in a class or private directory, then `fun.p` is placed in the current directory. Otherwise, `fun.p` is placed in a class or private subdirectory of the current directory, which `pcode` automatically creates if necessary. The original M-file can be anywhere on the search path.

`pcode *.m` creates P-files for all the M-files that are in the current directory. The location of the new P-files is as described above.

`pcode fun1 fun2 ...` creates P-files for the listed functions. The location of the new P-files is as described above.

`pcode ... -inplace` creates P-files in the same directory as the corresponding M-files.

`pcode ... -runtime` creates *runtime* P-files. If `fun.m` is not in a class or private directory, then `fun.p` is placed in the current directory. Otherwise, `fun.p` is placed in a class or private subdirectory of the current directory, which `pcode` automatically creates if necessary. The `pcode.m` file itself cannot be compiled into a runtime P-file.

`pcode ... -inplace -runtime` creates *runtime* P-files in the same directory as the corresponding M-files. The `pcode.m` file itself cannot be compiled into a runtime P-file.

Note In order for runtime P-files to work with a runtime application, the shipping version of MATLAB must have the same password stamp as the development version of MATLAB that created the runtime P-file. See "Password Consistency Rules" on page xi for more details.

See Also

`cleanp`, `pcodeall`

pcodeall

Purpose

Compile all M-files on the path and current directory into P-files

Syntax

```
pcodeall
pcodeall -runtime
```

Description

`pcodeall` compiles all M-files on the current MATLAB path and in the current directory into standard P-files (preparsed MATLAB code), including those in private and class directories.

`pcodeall -runtime` compiles all M-files on the current MATLAB path and in the current directory into *runtime* P-files, including those in private and class directories. MEX-files and runtime P-files are the only executable files that the MATLAB Runtime Server recognizes.

Each generated P-file resides in the same directory as its corresponding M-file.

Note In order for runtime P-files to work with a runtime application, the shipping version of MATLAB must have the same password stamp as the development version of MATLAB that created the runtime P-file. See "Password Consistency Rules" on page xi for more details.

See Also

`cleanp`, `pcode`

Purpose

Emulate the runtime environment in MATLAB and set the global error mode

Syntax

```
runtime on
runtime off
runtime status
runtime errormode mode
```

Description

The `runtime` command lets you emulate the Runtime Server environment in commercial MATLAB. On PC platforms, it also sets the global error mode for a runtime application. Because the Runtime Server disables the command window, it is generally much more convenient to test and debug with MATLAB emulating the Runtime Server than with the Runtime Server variant itself.

`runtime on` tells commercial MATLAB to begin emulating the Runtime Server. This means that MATLAB executes neither M-files nor standard P-files. The command line remains accessible.

`runtime off` returns MATLAB to its ordinary state.

`runtime status` indicates whether MATLAB is emulating the Runtime Server or not.

`runtime errormode` *`mode`* sets the global error mode to *`mode`* on PC platforms. The value of *`mode`* can be either `continue`, `quit`, or `dialog`. However, `dialog` is both the default error mode and the recommended one.

The error mode setting is only effective when the application runs with the Runtime Server; when the application runs with commercial MATLAB emulating the Runtime Server, untrapped errors are always displayed in the command window. For details about error modes, see “Setting the Global Error Behavior on a PC” on page 1-9.

Index

D

E

F

G

I

J

L

M

O

P

Q

R

S

T

U

V

W