mobile communications series

SOFTWARE-DEFINED RADIO for ENGINEERS

TRAVIS F. COLLINS
ROBIN GETZ
DI PU
ALEXANDER M. WYGLINSKI

Software-Defined Radio for Engineers

For a listing of recent titles in the *Artech House Mobile Communications*, turn to the back of this book.

Software-Defined Radio for Engineers

Travis F. Collins
Robin Getz
Di Pu
Alexander M. Wyglinski

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalog record for this book is available from the British Library.

ISBN-13: 978-1-63081-457-1

Cover design by John Gomes

10 9 8 7 6 5 4 3 2 1

Dedication

To my wife Lauren
—Travis Collins

To my wonderful children, Matthew, Lauren, and Isaac, and my patient wife, Michelle—sorry I have been hiding in the basement working on this book. To all my fantastic colleagues at Analog Devices: Dave, Michael, Lars-Peter, Andrei, Mihai, Travis, Wyatt and many more, without whom Pluto SDR and IIO would not exist.
—Robin Getz

To my lovely son Aidi, my husband Di, and my parents Lingzhen and Xuexun
—Di Pu

To my wife Jen
—Alexander Wyglinski

Contents

APPENDIX B

Getting Started with MATLAB and Simulink

You will be using MATLAB and Simulink for the experiments and for the open-ended design projects in this book. This appendix serves as a brief refresher of MATLAB, since you should have used it before. However, if you don't have extensive experience with Simulink, then this appendix shows you how to get started with the tool. Please note the MATLAB portion of this appendix is mainly based on the MATLAB documentation presented in [1] and the Simulink portion is based on the Simulink Basics Tutorial presented in [2], but here we extract the most important and fundamental concept so that you can quickly get started after reading this appendix. For more information about these two products, you are encouraged to refer to [1] and [2].

B.1 MATLAB Introduction

MATLAB is widely used in all areas of applied mathematics, in education and research at universities, and in industry. MATLAB stands for Matrix Laboratory and the software is built up around vectors and matrices. Consequently, this makes the software particularly useful for solving problems in linear algebra, but also for solving algebraic and differential equations as well as numerical integration. MATLAB possesses a collection of graphic tools capable of producing advanced GUI and data plots in both 2-D and 3-D. MATLAB also has several toolboxes useful for performing communications, signal processing, image processing, optimization, and other specialized operations.

MathWorks has created an excellent online tutorial to review basic and advanced concepts, as well as provide instructor lead tutorials to show off the various capabilities of MATLAB. It can be found at https://matlabacademy.mathworks.com

B.2 Useful MATLAB Tools

This section introduces general techniques for finding errors, as well as using automatic code analysis functions in order to detect possible areas for improvement within the MATLAB code. In particular, the MATLAB debugger features located within the Editor, as well as equivalent Command Window debugging functions, will be covered.

Debugging is the process by which you isolate and fix problems with your code. Debugging helps to correct two kinds of errors:

- Syntax errors: For example, misspelling a function name or omitting a parenthesis.
- Run-time errors: These errors are usually algorithmic in nature. For example, you might modify the wrong variable or code a calculation incorrectly. Run-time errors are usually apparent when an M-file produces unexpected results. Run-time errors are difficult to track down because the function's local workspace is lost when the error forces a return to the MATLAB base workspace.

B.2.1 Code Analysis and M-Lint Messages

MATLAB can check your code for problems and recommend modifications to maximize the performance and maintainability through messages, sometimes referred to as M-Lint messages. The term lint is the name given to similar tools used with other programming languages such as C. In MATLAB, the M-Lint tool displays a message for each line of an M-file it determines possesses the potential to be improved. For example, a common M-Lint message is that a variable is defined but never used in the M-file.

You can check for coding problems using three different ways, all of which report the same messages:

- Continuously check code in the Editor while you work. View M-Lint messages and modify your file based on the messages. The messages update automatically and continuously so you can see if your changes addressed the issues noted in the messages. Some messages offer extended information, automatic code correction, or both.
- Run a report for an existing MATLAB code file: From a file in the Editor, select **Tools > Code Analyzer > Show Code Analyzer Report**.
- Run a report for all files in a folder: In the Current Folder browser, click the Actions button, then select **Reports > Code Analyzer Report**.

For each message, review the message and the associated code in order to make changes to the code itself based on the message via the following process:

- Click the line number to open the M-file in the Editor/Debugger at that line.
- Review the M-Lint message in the report and change the code in the M-file based on the message.
- Note that in some cases, you should not make any changes based on the M-Lint messages because the M-Lint messages do not apply to that specific situation. M-Lint does not provide perfect information about every situation.
- Save the M-file. Consider saving the file to a different name if you made significant changes that might introduce errors. Then you can refer to the original file as you resolve problems with the updated file.
- If you are not sure what a message means or what to change in the code as a result, use the Help browser to look for related topics.

You can also get M-Lint messages using the `mlint` function. For more information about this function, you can type `help mlint` in the Command Window. Read the online documentation [3] for more information about this tool.

B.2.2 Debugger

The MATLAB Editor, graphical debugger, and MATLAB debugging functions are useful for correcting run-time problems. They enable access to function workspaces and examine or change the values they contain. You can also set and clear *breakpoints*, which are indicators that temporarily halt execution in a file. While stopped at a breakpoint, you can change the workspace contexts, view the function call stack, and execute the lines in a file one by one.

There are two important techniques in debugging: one is the *breakpoint* while the other is the *step*. Setting breakpoints to pause the execution of a function enables you to examine values where you think the problem might be located. While debugging, you can also step through an M-file, pausing at points where you want to examine values.

There are three basic types of breakpoints that you can set in the M-files:

- A standard breakpoint, which stops at a specified line in an M-file.
- A conditional breakpoint, which stops at a specified line in an M-file only under specified conditions.
- An error breakpoint that stops in any M-file when it produces the specified type of warning, error, or NaN or infinite value.

You cannot set breakpoints while MATLAB is busy (e.g., running an M-file, unless that M-file is paused at a breakpoint). While the program is paused, you can view the value of any variable currently in the workspace, thus allowing you to examine values when you want to see whether a line of code has produced the expected result or not. If the result is as expected, continue running or step to the next line. If the result is not as expected, then that line, or a previous line, contains an error.

While debugging, you can change the value of a variable in the current workspace to see if the new value produces expected results. While the program is paused, assign a new value to the variable in the Command Window, Workspace browser, or Array Editor. Then continue running or stepping through the program. If the new value does not produce the expected results, the program has a different or another problem.

Besides using the Editor, which is a graphical user interface, you can also debug MATLAB files by using debugging functions from the Command Window, or you can use both methods interchangeably. Read the online documentation [4] for more information about this tool.

B.2.3 Profiler

Profiling is a way to measure the amount of time a program spends on performing various functions. Using the MATLAB Profiler, you can identify which functions in your code consume the most time. You can then determine why you are calling them and look for ways to minimize their use. It is often helpful to decide whether the number of times a particular function is called is reasonable. Since

programs often have several layers, your code may not explicitly call the most time-consuming functions. Rather, functions within your code might be calling other time-consuming functions that can be several layers down into the code. In this case, it is important to determine which of your functions are responsible for such calls.

Profiling helps to uncover performance problems that you can solve by

- Avoiding unnecessary computation, which can arise from oversight.
- Changing your algorithm to avoid costly functions.
- Avoiding recomputation by storing results for future use.

When you reach the point where most of the time is spent on calls to a small number of built-in functions, you have probably optimized the code as much as you can expect. You can use any of the following methods to open the Profiler:

- Select **Desktop** → **Profiler** from the MATLAB desktop.
- Select **Tools** → **Open Profiler** from the menu in the MATLAB Editor/Debugger.
- Select one or more statements in the Command History window, right-click to view the context menu, and choose **Profile Code**.
- Enter the following function in the Command Window: `profile viewer`.

To profile an M-file or a line of code, follow these steps after you open the Profiler, as shown in Figure B.1:

1. In the **Run this code** field in the Profiler, type the statement you want to run.
2. Click **Start Profiling** (or press **Enter** after typing the statement).
3. When profiling is complete, the **Profile Summary** report appears in the Profiler window.

Read the online documentation [5] for more information about this tool.

B.3 System Objects

System objects are a specialization of a class in MATLAB, which define a specific set of methods that make initialization, runtime operation, and tear-down simple.

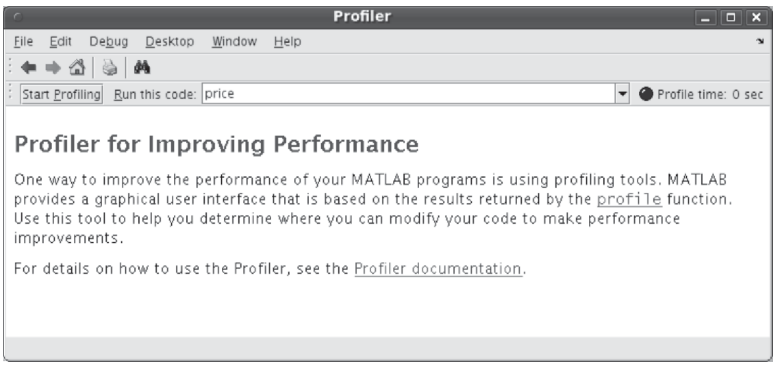

Figure B.1 The profiler window.

A class itself is basically a set of functions that share a set of variables called parameters. These parameters are defined within the class and have defined scopes. Although many methods are implemented by system objects, the three main methods a user should understand are `setupImpl`, `stepImpl`, and `releaseImpl`. They will be written as

```
1 methods (Access = protected)
2     function setupImpl(obj)
3         % Set parameter
4         obj.x = 1;
5     end
6 end
```

`setupImpl` is used to initial parameters and perform calculations that are needed for the life of the system object. `stepImpl` is the runtime function (method) that is called generally multiple times and will consume or produce inputs/outputs. Finally, `releaseImpl` is used to tear-down the object that will clear its memory or perform closing actions. For example, if the object is to write data to a file it will close the file when this method is called.

When a system object's operator or step method is called, the initial call will actually first call the `setupImpl` method. At this point the system object is considered locked. Then the `stepImpl` method will be called. Successive operator or step calls will only call the `stepImpl` method. To unlock the object the `releaseImpl` method must be called. Once unlock the `setupImpl` will again be called at the next operator or step call. We outline this process in a script here:

```
 1 % Instantiate object
 2 ss = dsp.SignalSource;
 3 % setupImpl and stepImpl are called
 4 data = ss();
 5 % stepImpl is only called
 6 data = ss();
 7 % Object is unlocked
 8 ss.release();
 9 % setupImpl and stepImpl are called
10 data = ss.step();
```

System objects are extremely useful for encapsulating a large degree of functionality and when state needs to be maintained. For example, filters require state and are an obvious use for system objects. The toolboxes that make up MATLAB utilize extensions for their system objects that are related to their abbreviation. For example, system objects that are from the Communication Systems Toolbox will have the extension `comm`, resulting in objects with names such as `comm.AGC`, `comm.CarrierSynchronizer`, or `comm.EVM`. Examples in the DSP Systems Toolbox are: `dsp.FIRDecimator`, `dsp.SpectrumAnalyzer`, and `dsp.SignalSource`. More information about system objects can be found in the MathWorks documentation with extensive details on their implementation and use.

References

[1] The MathWorks, *MATLAB Documentation*, http://www.mathworks.com/help/techdoc/.

[2] University of Michigan, *Simulink Basics Tutorial*, http://www.engin.umich.edu/group/ctm/working/mac/simulink_basics/.

[3] The MathWorks, *Avoid Mistakes While Editing Code*, http://www.mathworks.com/help/techdoc/matlab_env/brqxeeu-151.html.

[4] The MathWorks, *Debugging Process and Features*, http://www.mathworks.com/help/techdoc/matlab_env/brqxeeu-175.html.

[5] The MathWorks, *Profiling for Improving Performance*, http://www.mathworks.com/help/techdoc/matlab_env/f9-17018.html