## Overview

The following sections are an overview of my setup of a MATLAB GUI to do TCPIP communication to the robot cars used in GE423: Mechatronics at the University of Illinois during the Spring of 2014. This document is meant as a relatively high-level overview of important aspects of the project. Sources for additional information are provided through hyperlinks wherever possible.

## Using MATLAB GUIDE

MATLAB GUIDE is a useful tool for quickly laying out and programming graphical user interfaces. It is **not** **necessary** to create graphical user interfaces in GUIDE (many examples given online create GUIs through a single function file with no supporting “.fig” file). At the very least, the GUIDE graphical layout tool is much simpler way to layout graphical components. The downside to using GUIDE is that there are some processes that happen behind the scenes when MATLAB uses your GUI layout to produce a code file. (For example, notice that the first function in the generated code file has the comment tells the user not to edit it.)

The GUIDE tool can be started by typing ‘guide’ at the MATLAB command prompt. MATLAB prompts the user to either create a new layout, or open an existing one. The workflow that I have found works when using guide is the following:

1. Create GUI layout in graphical tool. Name all objects on the layout something intuitive now. (renaming/adding objects should always be done in the GUI layout tool. This is because the names of objects in the .fig file need to match the names used in code. Since the ‘.fig’ file generates the editable code, (and the code does not change the ‘.fig file’) changing names of components needs to be done at this level.

TIP: ‘.fig’ files are just ‘.mat’ files which contain a single special struct. The struct contains all child objects of the figure and their properties. To see this, take any ‘.fig’ file, copy it (just to be safe), and change the copied file extension to be ‘.mat’. Load this ‘.mat’ file into your workspace like any other. Now you can view (or change) any subfields in the .mat file. Of course, if you want these changes to take effect in a ‘.fig’ file, you need to save your workspace and rename the generated ‘.mat’ file a ‘.fig’ file.

1. Save the GUI layout (just a ‘.fig’ file) through GUIDE. This will automatically open an accompanying ‘.m’ file where all the code for the GUI resides. If it is the first time you saved the ‘.fig file’, the ‘.m’ file will be only a skeleton of code. MATLAB will automatically generate the callback functions for each object you created on the original figure. If you develop code in this ‘.m’ file and decide later to add objects to the GUI, you can use the GUIDE tool to reopen the ‘.fig’ file and make necessary changes. When you save this changed ‘.fig’ file through GUIDE, the code you have already written in the associated ‘.m’ file will be altered to reflect the changes in the GUI. (You won’t change any code you have written there either. Additional callbacks will be added to the bottom of the ‘.m’ file.)

*Variable Scope and the ‘handles’ structure*

Passing variables between callback functions in MATLAB meant to be done using the ‘handles’ structure. This is as opposed to the often simpler method of using global variables (a technique we used in mechatronics for visual basic GUIs and DSP code). By default, every callback function has 3 input arguments which are accessible during its execution. These are (in their usual order): (1) hObject, (2) eventData, and (3) handles. hObject contains the handle of the calling object, eventData contains information about when the call was made, and the handles structure contains all variables being passed among different callback functions of the GUI.

The usual procedure for creating a variable to be used by multiple subroutines is written in the following example code snippet:

function TCPIP\_V6\_wImg\_OpeningFcn(hObject, eventdata, handles, varargin)

.

.

.

handles.TCPIP.numFloats=10; %Initialize a new field of the handles struct

.  
 .  
 .  
 guidata(hObject, handles); %Make update persist after function exits

(Note from the function name that the above code will run when the GUI first opens) These statements initialize a new field of the handles struct, and make the update persist beyond the scope of this function by using the command: guidata(hObject, handles).

Variables used with a function which are not part of the handles structure die when the function exits. Further, during GUI execution, none of the variables used in the GUI (even those in the handles structure) are visible in the user’s workspace by default. Using breakpoints during a callback is a simple way to get at their values.

## Use of the InputStreamByteWrapper class

***Why?***

I have not found a simple way to use TCPIP in MATLAB without having to deal with the underlying Java objects. MATLAB does have a toolbox ([Instrument control toolbox](http://www.mathworks.com/products/instrument/)) that allows users to use TCPIP (which I assume uses the underlying Java libraries) but as UofI does not have this toolbox, I needed to figure out how to interact with the underlying Java objects through MATLAB.

Java classes/methods can be used through MATLAB, but as you might expect, some compatibility between data types needs to be enforced behind the scenes ([more info here](http://www.mathworks.com/help/matlab/matlab_external/passing-data-to-a-java-method.html)). When using TCPIP through MATLAB, I use Java’s TCPIP classes. The classes used include a [Socket](http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html), an [InputStream](http://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html), and an [OutputStream](http://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html). In order to read n bytes from the Java input stream, I use a Java [read method](http://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html#read(byte[], int, int)) of the InputStream class. The method takes as an input argument a buffer in which to store the bytes which are read from the stream and relies on the buffer being passed by reference (it has no output argument.) When MATLAB data types are passed into java methods, the MATLAB type is converted to the closest possible java type (conversion rules are in the “more info” link above). After this conversion, the converted value is passed by **value**, and not by reference. Thus, it is not possible to pass MATLAB types by reference to Java functions. One solution to this problem is to create a java “wrapper” class. This is what the “InputStreamByteWrapper” class file is doing.

***Initializing an Instance of InputStreamByteWrapper In MATLAB***

In MATLAB, we initialize instances of the InputStreamByteWrapper with the following statement:

*myReader=InputStreamByteWrapper;*

After initializing this instance of the InputStreamByteWrapper class we can call any methods defined in the .java file. For example, we might write:

myReader.read(mySocket,numchars);

If you are having a problem initializing this class, make sure that you have compiled the ‘.java’ file into a ‘.class’ file and make sure that the location of the ‘.class’ file is currently on your java path. (Paths can be added to your java path using the statement ‘javaaddpath(<path>)’.)

***Compiling java classes for use in MATLAB***

A ‘.java’ file may contain information defining a class. In order to use this class in MATLAB, the ‘.java’ file must be compiled using a java compiler which MATLAB supports. (This compiler may not be the most recent java version. At this time of this writing, MATLAB 2013a does not support the latest Java 1.7 compiler, so I had to use Java 1.6.)

## Timer object in MATLAB

(More information can be found in [Mathworks documentation](http://www.mathworks.com/help/matlab/ref/timerclass.html))

Creating a MATLAB timer object can be accomplished with the following syntax:

myTimer=timer(*PropertyName*,*PropertyValue*,…);

Valid propertyname/propertyvalue pairs can be viewed in the [MathWorks documentation](http://www.mathworks.com/help/matlab/ref/timerclass.html).

Upon creation of the TCPIP GUI, a timer object is initialized using the following syntax:

handles.TCPIP.writeOutTimer=timer('ErrorFcn','disp(''Timer object error'')',…

'ExecutionMode','fixedRate');

Note that later in the code, (in the connect button callback function) the timer object is updated using the “set” function:

set(handles.TCPIP.writeOutTimer,'TimerFcn',{'writeOutData',hObject,handles},…

'Period',0.1)

The above statement gives the timer a function to call, and two input arguments. Note that the input arguments passed to the timer function on each execution are the **values** of ‘hobject’ and ‘handles’ when the ‘set’ statement is executed. E.g. if the ‘handles’ structure is modified after this ‘set’ statement, those changes will not be reflected in the ‘handles’ structure that is seen by the timer function. Thus is important that any items that are to be modified within the timer function are initialized before the timer ‘TimerFcn’ property is set.

After all properties of the timer are set, the timer can be started with the following command:

start(handles.TCPIP.writeOutTimer);

This statement starts the timer object running in the background at the predefined execution rate.

***Deleting Timers***

Creating a Timer object creates an underlying java object which is pointed to by the MATLAB timer variable, and deleting the MATLAB object will not delete the underlying Java object. The underlying object can be deleted using the ‘delete’ method of the timer class as follows:

delete(handles.TCPIP.writeOutTimer)

TIP: To view all existing timer objects in memory (even the ones for which you cleared the MATLAB pointer, type ‘timerfindall’ at the command prompt. To delete these timers, type ‘delete(timerfindall)’.

***Rules of Writing a Timer Function***

Timer functions have two standard input arguments by convention. Typically they are given names like: (1) ‘object’, and (2) ‘eventdata.’ Any user-written timer function needs to have these two input arguments appear before user-defined inputs (assuming there are any user-defined input arguments). The first input argument ‘object’ holds the handle of the object that started the timer. The second input argument ‘eventdata’ contains information about when the object was executed. This information can be very useful if you want to print out exactly when the timer function is being executed.