Robot Raconteur® using MATLAB

Version 0.8 Beta

http://robotraconteur.com

Dr. John Wason

Wason Technology, LLC

PO Box 669

Tuxedo, NY 10987

wason@wasontech.com

http://wasontech.com

May 3, 2016

Contents

1 Introduction

This document serves as the reference for using Robot Raconteur® with MATLAB. This is a supplement that only focuses on issues related to the MATLAB interface. It is assumed the user has reviewed "Introduction to Robot Raconteur® using Python" which serves as the primary reference for the Robot Raconteur® library itself. This document refers to the iRobot Create example software. Because of the verbosity of the code, it will not be reproduced in this document. The reader should download the software examples for reference. Tha MATLAB interface is client only; it does not support writing services.

2 Installation

The MATLAB Robot Raconteur interface are distributed in compressed archives corresponding to the target architecture. Simply unpack the archive and add the directory that contains "RobotRaconteurMex.*" to the path of MATLAB.

3 MATLAB $\leftrightarrow$ Robot Raconteur Data Type Mapping

Each valid Robot Raconteur type has a corresponding MATLAB data type. Scalars, strings, numeric arrays, structures, and multi-dimensional arrays map directly to MATLAB data types. Multi-dimensional arrays do not have special data types in MATLAB due to the native support within MATLAB. Table 1 shows the mapping between Robot Raconteur and MATLAB data types.

In MATLAB, the default is for numbers to be type `double`. When passing numeric arguments to Robot Raconteur it is necessary to match the expected type. This is easily done by using the type name as a function, for example:

```
bytevalue=uint8([10; 20]);
```

All arrays are *column* vectors. Multi-dimensional arrays behave like normal arrays.

Maps in MATLAB use the built in `containers.Map` type. The key must be type `char` or `int32`. As an example map of type `double[]{int32}`:

```
mymap=containers.Map(int32({1; 2}), {[10; 20]; [30; 40]});
```

Robot Raconteur structures map directly to MATLAB structures without any extra work. The structures must have matching fields and types. A structure can be created dynamically. For instance, to create a `SensorPacket` structure:

```
s=struct;
s.ID=uint8(19);
s.Data=uint8([1; 2; 3]);
```

`s` can now be passed where `SensorPacket` is expected. If a `varvalue` is expected, an extra field `RobotRaconteurStructureType` that contains the fully qualified structure type is expected as a string.

4 Service Objects

Robot Raconteur service objects are returned as `RobotRaconteurObject` objects that dynamically map from MATLAB to Robot Raconteur. This is a dynamic class and retrieves the type information on connection. Two helper commands `type` and `members` returns the Robot Raconteur type of the object and prints out the definition of the object type, respectively.

```
obj=RobotRaconteur.ConnectService('rr+tcp://localhost/?service=Create');
type(obj) %Return the object type
members(obj) %Print out the object definition
```

Note that exceptions are passed transparently to and from MATLAB using standard `error`, `try/catch` notation.

Table 1: Robot Raconteur ↔ MATLAB Type Map

Robot Raconteur Type	MATLAB Type	Notes
double	`double`	
single	`float`	
int8	`int8`	
uint8	`uint8`	
int16	`int16`	
uint16	`uint16`	
int32	`int32`	
uint32	`uint32`	
int64	`int64`	
uint64	`uint64`	
double[]	`double`	
single[]	`float`	
int8[]	`int8`	
uint8[]	`uint8`	
int16[]	`int16`	
uint16[]	`uint16`	
int32[]	`int32`	
uint32[]	`uint32`	
int64[]	`int64`	
uint64[]	`uint64`	
string	`char[]`	Strings are transmitted as UTF-8 but converted to normal MATLAB strings
$T\{$int32$\}$	`containers.Map`	See text for more details
$T\{$string$\}$	`contairs.Map`	See text for more details
$T\{$list$\}$	`cell` column vector	Cell vector containing the entries of the list
structure	*struct*	Standard MATLAB struct
N[*]	`N`	Normal MATLAB array of type N
varvalue	`*`	Mapped directly to appropriate type
varobject	`RobotRaconteurObject`	

The following list describes the member types and gives a short example of the usage. In all the examples, `obj` is the connected object. The `obj` is the "Create_interface.Create" object type and `webobj` is the "Webcam_interface.WebcamHost" type.

property

Properties are implemented as standard MATLAB properties. They can be accessed using "dot" notation.

```
dist=obj.DistanceTraveled;
obj.DistanceTraveled=uint16(100);
```

function

Functions are implemented as standard MATLAB functions. They can be accessed using "dot" notation.

```
obj.Drive(int16(200),int16(1000));
```

event

In MATLAB, a callback function handle set to be called when an event occurs by using the "addlistener" command. Events are disabled by default. To enable events on the object, run `RobotRaconteur.EnableEvents(obj)` where `obj` is the object. The callback function must have the correct number of arguments matching the event definition. Because MATLAB is single threaded it is necessary to call the `RobotRaconteur.ProcessRequests()` command to execute the event pump.

```
RobotRaconteur.EnableEvents(obj)
addlistener(c,'Bump',@Bump) % Bump is a function
RobotRaconteur.ProcessRequests(); %Run repeatedly to process events
```

objref

The objref members are implemented through a function that is named "get_" pre-pended to the member name of the objref. The index is the argument to the function if there is an index. This can be accessed using the "dot" notation.

```
obj2=webobj.get_Webcams(0);
```

pipe

Pipes are have a special syntax that allows connecting `PipeEndpoint` objects. Replace `FrameStream` with the name of the pipe being connected. `pe` is the connected `PipeEndpoint`. The -1 can be replaced with the desired endpoint index. See Section A.3 for more details.

```
pe=obj2.FrameStream.connect(-1);
count=pe.Available
datin=pe.ReceivePacket();
pe.SendPacket(datout);
```

callback

Callbacks in MATLAB are function handles that have matching parameters and return value (or empty if void). "Dot" notation is used to specify the function handle similar to properties. Because MATLAB is single threaded, it is necessary to pump the callbacks using the `RobotRaconteur.ProcessRequests()` command.

```
obj.play_callback=@my_play_callback; % my_play_callback is a function
RobotRaconteur.ProcessRequests(); %Run repeatedly to process callbacks and events
```

wire

Pipes are have a special syntax that allows connecting `WireConnection` objects. Replace `packets` with the name of the pipe being connected. `wc` is the connected `WireConnection`. See Section A.4 for more details.

```
wc=obj.frame.connect();
wc.OutValue=myoutdat;
myindat=wc.InValue;
```

memory

Memories in MATLAB work by returning a special object of type `RobotRaconteurMemoryClient`. This object type has much of the same functionality as a normal MATLAB array. The main difference is to read and write the full array use the `(:)` notation. The memory must be stored in a variable before use.

```
buffer=obj2.buffer;
size(buffer);
buffer(:)=mydat1;
mydat2=buffer(:);
```

See the reference section (Section A) and examples for more details for how to use the MATLAB interface.

5 Asynchronous Operations

Robot Raconteur has limited support for asynchronous operations. Property get, property set, and functions can be called asynchronously. The asynchronous version is called with the normal parameters for the call, a handler function, a user defined callback parameter, and a timeout in seconds. The invocation returns immediately, and the handler function is called once the operation has completed. Because of the single threaded nature of MATLAB the handler will only be called when the function `RobotRaconteur.ProcessRequests()` is called.

async property get

The asynchronous property gets are accessed by prepending "async_get_" to the member

name and passing the asynchronous parameters. The form is:

`obj.async_get_`*`propertyname`*`(handler,param,timeout)`

where `propertyname` is the name of the property to get, `handler` is a function handle of the form `function myhandler(param, value, error)`, `param` is any value that you want to pass from the call to the handler, and `timeout` is the timeout for the call in seconds. `error` will be empty if no error occurs or will be a string containing the error if an error occurs. An example of getting `DistancTraveled`:

```
obj.async_get_DistanceTraveled(@myhandler,1,1);
```

async property set

The asynchronous property sets are accessed by prepending "async_set_" to the member name and passing the asynchronous parameters. The form is:

`obj.async_set_`*`propertyname`*`(value,handler,param,timeout)`

where `propertyname` is the name of the property to set, `value` is the value to set, `handler` is a function handle of the form `function myhandler(param, error)`, `param` is any value that you want to pass from the call to the handler, and `timeout` is the timeout for the call in seconds. `error` will be empty if no error occurs or will be a string containing the error if an error occurs. An example of setting `DistanceTraveled`:

```
obj.async_set_DistanceTraveled(10,@myhandler,1,0.5);
```

In this case error will be set because the property is read only.

async function

Functions are implemented by prepending "async_" to the function name. The three extra asynchronous parameters are added to the parameter list. The form is:

`obj.async_`*`functionname`*`(`*`parameters`*`,handler,param,timeout)`

where `functionname` is the name of the function, `parameters` are the zero or more parameters that are part of the original function call, `handler` a function handle of the form `function myhandler(param, error)` for void return and `function myhandler(param, ret, error)` for non-void return, `param` is any value that you want to pass from the call to the handler, and `timeout` is the timeout for the call in seconds. `error` will be empty if no error occurs or will be a string containing the error if an error occurs. An example of calling `Drive` is:

```
obj.async_Drive(int16(200),int16(1000),@myhandler,1,0.5);
```

A common use of asynchronous operations is to read multiple sensors simultaneously. For instance, it may be necessary to read a few dozen light sensors for an advanced lighting control system. Asynchronous operations are very efficient because they do not use any resources between the call and the handler. Example 1 shows an example of reading many sensors. The variable `c` contains a cell array of sensor connections to read.

Example 1 Example of simultaneously reading of sensors using asynchronous operations

```
function [res, err]=readsensors(c,timeout)

        N=length(c);
        res=cell(N,1);
        err=cell(N,1);
        activekeys=[];

        for i=1:N
                c1=connections{i};
                c1.async_ReadSensor(@myhandler,i,timeout);
                activekeys=[activekeys;i];
        end

        while (~isempty(activekeys))
                RobotRaconteur.ProcessRequests();
        end



function myhandler(key1, value1, err1)
        res{key1}=value1
        err{key1}=err1
        activekeys(activekeys==key1)=[];

end
end
```

6 Using MATLAB as a Robot Raconteur server

Occasionally it is useful to expose a MATLAB object as a service using Robot Raconteur so that it can be accessed by other computers or programs using a standard Robot Raconteur client. As of version 0.5-testing-2014-10-17 Robot Raconteur supports exposing MATLAB object as services. Using MATLAB a a server is difficult because MATLAB has very limited threading capabilities. Because of this limitation MATLAB must call `RobotRaconteur.ProcessServerRequests` repeatedly to process incoming requests. The object registered as a service must be a new style "classdef" class that extends "handle". It can use the standard built in properties, functions, and events using any `varvalue` type. Note that objrefs cannot be used. The service definition for the object may not contain anything but functions, properties, and events but has full support for structures and other data types. See Appendix A.5 and the examples (separate download) for more information.

7 Conclusion

This document serves as a reference for the MATLAB bindings of Robot Raconteur. More information can be found on the project website.

A Robot Raconteur Reference

A.1 RobotRaconteur Object Global Commands

obj = RobotRaconteur.ConnectService(*url*, *username*, *credentials*)

Connects to a service. It retrieves the service definition on connection to determine type information. The *username* and *credentials* are optional use with services requiring authentication.

Parameters:

- *url* (`string` or `cell`) - The url of the service or multiple url in a cell array
- *username* (`string`) - (optional) The username for authentication
- *credentials* (`containers.Map`) - (optional) A `containers.Map` instance with string key type containing the credentials to use for authentication.

Return Value:

(`RobotRaconteurObject`) - The connected service object

Example:

```
obj=RobotRaconteur.ConnectService('rr+tcp://localhost/?service=Create');
```

With authentication:

```
credentials=containers.Map({'password'},{'mypassword'});
obj=RobotRaconteur.ConnectService('rr+tcp://localhost/?service=Create',
    'myusername',credentials);
```

RobotRaconteur.DisconnectService(*obj*)

Disconnects the service. `obj` must have been returned by `RobotRaconteur.ConnectService`.

Parameters:

- *obj* (`RobotRaconteurObject`) - The service to disconnect

Return Value:

None

Example:

```
RobotRaconteur.DisconnectService(obj);
```

RobotRaconteur.EnableEvents(*obj*)

By default objects will not process events and they will simply be ignored. By running this command, events will be queued and will be processed during calls to `RobotRaconteur.ProcessRequest`

Parameters:

- *obj* (`RobotRaconteurObject`) - The service object on which to enable events

Return Value:

None

Example:

```
RobotRaconteur.EnableEvents(obj);
```

RobotRaconteur.DisableEvents(*obj*)

Disables events for *obj*.

Parameters:

- *obj* (`RobotRaconteurObject`) - The service object on which to disable events

Return Value:

None

Example:

```
RobotRaconteur.EnableEvents(obj);
```

RobotRaconteur.ProcessRequests()

Processes events and requests. Because MATLAB is single threaded this must be called or events and callbacks will not be processed.

Parameters:

None

Return Value:

None

Example:

```
RobotRaconteur.ProcessRequests();
```

RobotRaconteur.RequestObjectLock(*obj*, *type*)

Requests a lock of the supplied service object.

Parameters:

- *obj* (`RobotRaconteurObject`) - The service object to lock
- *type* (`string`) - The type of lock. Can be 'User' or 'Client'

Return Value:

None

Example:

```
RobotRaconteur.RequestObjectLock(obj,'User');
```

RobotRaconteur.ReleaseObjectLock(*obj*)

Releases a lock of the supplied service object.

Parameters:

- *obj* (`RobotRaconteurObject`) - The service object to unlock

Return Value:

None

Example:

```
RobotRaconteur.ReleaseObjectLock(obj);
```

RobotRaconteur.MonitorEnter(*obj*)

Enters a monitor lock.

Parameters:

- *obj* (`RobotRaconteurObject`) - The service object to lock

Return Value:

None

Example:

```
RobotRaconteur.MonitorEnter(obj);
```

RobotRaconteur.MonitorExit(*obj*)

Enters a monitor lock.

Parameters:

- *obj* (`RobotRaconteurObject`) - The service object to unlock

Return Value:

None

Example:

```
RobotRaconteur.MonitorExit(obj);
```

`ret`=**RobotRaconteur.GetPulledServiceType**(*obj*, *servicename*)

Retrieves a service definition in string form that has been pulled from a Robot Raconteur service.

Parameters:

- *obj* (`RobotRaconteurObject`) - The object returned by `ConnectService`.
- *servicename* (`string`) - The name of the service definition to retrieve.

Return Value:

(`string`) - The service definition as a string.

`ret`=**RobotRaconteur.GetPulledServiceTypes**(*obj*)

Retrieves a cell list of service definition names pulled by the client object *obj*.

Parameters:

- *obj* (`RobotRaconteurObject`) - The object returned by `ConnectService`.

Return Value:

(`string`) - The cell list of service definitions.

res = RobotRaconteur.FindServiceByType(*type*)

Finds a service by type using node auto-discovery.

Parameters:

- *type* (`string`) - The fully qualified type of the object to search for.

Return Value:

(`cell`) - The found services in a cell vector. The returned structures have the same format as the Python ServiceInfo2 structure.

Example:

```
res=RobotRaconteur.FindServiceByType(type);
```

res = RobotRaconteur.FindNodeByID(*id*)

Finds a node by "NodeID" using node auto-discovery.

Parameters:

- *type* (`string`) - The "NodeID" to search for in string form.

Return Value:

(`cell`) - The found nodes in a cell vector. The returned structures have the same format as the Python NodeInfo2 structure.

Example:

```
res=RobotRaconteur.FindNodeByID(id);
```

res = RobotRaconteur.FindNodeByName(*name*)

Finds a node by "NodeName" using node auto-discovery.

Parameters:

- *type* (`string`) - The "NodeName" to search for.

Return Value:

(`cell`) - The found nodes in a cell vector. The returned structures have the same format as the Python NodeInfo2 structure.

Example:

```
res=RobotRaconteur.FindNodeByName(name);
```

RobotRaconteur.UpdateDetectedNodes()

Updates detected nodes. Must be called before `GetDetectedNodes`

Parameters:

None

Return Value:

None

Example:

```
RobotRaconteur.UpdateDetectedNodes()
```

res = RobotRaconteur.GetDetectedNodes()

Returns a cell array vector of detected NodeID as strings. `UpdateDetectedNodes` must be called before calling this function.

Parameters:

None

Return Value:

(`cell`) - The detected nodes as a cell vector of strings.

Example:

```
res = RobotRaconteur.GetDetectedNodes
```

t = RobotRaconteur.nowUTC()

Returns the node time using the same format as the built in MATLAB `now` command. This can be useful if the node is running in a simulation timebase.

Parameters:

None

Return Value:

(`double`) - The node time as a serial date number.

c = RobotRaconteur.clock()

Returns the node time using the same format as the built in MATLAB `clock` command. This can be useful if the node is running in a simulation timebase.

Parameters:

None

Return Value:

(`double[]`) - The node date and time as date vector.

A.2 RobotRaconteurObject

`RobotRaconteurObject` is used for client service object references. Section 4 demonstrates how to access the different member types. The object also has a number of ”commands” that are not members but control the object. These ”commands” do not use ”dot” notation but instead pass the object as the first parameter.

`objtype` **= type**(*obj*)

Returns the fully qualified type as a string.

Parameters:

- *obj* (`RobotRaconteurObject`) - The object.

Return Value:

(`string`) - The fully qualified type of `obj`

`members` **= type**(*obj*)

Prints or returns the object definition.

Parameters:

- *obj* (`RobotRaconteurObject`) - The object.

Return Value:

(`string`) - The object definition `obj`

`const` **= constants**(*obj*)

Returns a structure with the constants in the service definition.

Parameters:

- *obj* (`RobotRaconteurObject`) - The object.

Return Value:

(`string`) - (struct) - The constants stored in a structure.

`eventhandle` **= addlistener**(*obj*, *eventname*, *func*)

Adds a function handle to listen for event *eventname*. The event will not be triggered unless events are enabled for this object and `RobotRaconteur.ProcessRequests()` is called. The return value *eventhandle* can be used to delete the connection using the `delete(eventhandle)`.

Parameters:

- *obj* (`RobotRaconteurObject`) - The object.
- *eventname* (`string`) - The member name of the event.
- *func* (`Function Handle`) - Handle to the function to call when event is triggered.

Return Value:

(RobotRaconteurEventListener) - Handle to the event used to delete the event handler connection. Call `delete(eventhandle)` to clear the handler.

A.3 PipeEndpoint

The `PipeEndpoint` object is used to represent the pipe endpoint connection. The pipe endpoint type is implemented using `RobotRaconteurObject` but has the members that match the pipe endpoint. A pipe connection is created by connecting through a Robot Raconteur object, for example:

```
p=c1.FrameStream.Connect(-1);
```

`a`=**p.Available**

Returns the number of packets that can be received.

`packet`=**p.ReceivePacket**()

Receives a packet. The number of packets available can be determined through the `Available` property.

Parameters:

None

Return Value:

(*) - The received packet

`packet`=**p.PeekPacket**()

Peeks a packet. The number of packets available can be determined through the `Available` property. This is the same as `ReceivePacket` but does not remove the packet from the queue.

Parameters:

None

Return Value:

(*) - The received packet

p.SendPacket(`packet`)

Sends a packet.

Parameters:

- *packet* (*) - The packet to send.

Return Value:

None

p.Close()

Closes the PipeEndpoint connection.

Parameters:

None

Return Value:

None

A.4 WireConnection

The `WireConnection` object is used to represent the wire connection. The wire connection type is implemented using `RobotRaconteurObject` but has the members that match the wire connection. A wire connection is created by connecting through a Robot Raconteur object, for example:

```
w=obj.packets.Connect();
```

`value=`**w.InValue**

The InValue for the pipe. This may throw an error if the value is not set.

`value=`**w.OutValue**
w.OutValue`=value`

The OutValue for the pipe. This is a read/write property. This may throw an error if the value is not set.

`valid=`**w.InValueValid**

true is InValue is valid, otherwise false.

`valid=`**w.OutValueValid**

true is OutValue is valid, otherwise false.

`time=`**w.LastValueReceivedTime**

The time that the last InValue was received in the sender's clock. This will return a structure with two fields, `seconds` and `nanoseconds`. This will normally be relative to the epoch January 1, 1970.

`time=`**w.LastValueSentTime**

The time that the last OutValue was set in the local clock. This will return a structure with two fields, `seconds` and `nanoseconds`. This will normally be relative to the epoch January 1, 1970.

A.5 Server Reference

These global commands are used for exposing MATLAB objects as services. For most users they can be ignored.

`ret`=**RobotRaconteur.GetServiceType**(*servicename*)

Retrieves a service definition in string form.

Parameters:

- *servicename* (`string`) - The name of the service definition to retrieve.

Return Value:

(`string`) - The service definition as a string.

`ret`=**RobotRaconteur.GetRegisteredServiceTypes**()

Retrieves a cell list of registered service definition names.

Parameters:

None

Return Value:

(`string`) - The cell list of service definitions.

RobotRaconteur.RegisterServiceType(*servicedef*)

Registered a service definition.

Parameters:

- *servicedef* (`string`) - The service definition in string form.

Return Value:

None

Example:

RobotRaconteur.StartLocalServer(*nodeid*)

Starts the LocalTransport server listening for connections.

Parameters:

- *nodeid* (`string`) - The nodeid to listen to. Can either be a NodeID in string form or a valid NodeName.

Return Value:

None

Example:

```
RobotRaconteur.StartLocalServer('141141a2-a67c-46a2-81ed-d8a12c0e694a');
```

or

```
RobotRaconteur.StartLocalServer('example.MatlabTestServer');
```

RobotRaconteur.StartLocalClient(*nodeid*)

This function is optional for clients. Starts the LocalTransport client listening for connections and pulls a "NodeID" from the registry based on the "NodeName".

Parameters:

- *nodename* (`string`) - The nodename to use to retrieve the nodeid.

Return Value:

None

Example:

```
RobotRaconteur.StartLocalClient('example.MatlabTestClient');
```

RobotRaconteur.StartTcpServer(*port*)

Starts the TcpTransport server listening for connections.

Parameters:

- *port* (`int32`) - The port to listen on. Set to 0 to let the system select the port. Use `GetListenPort()` to get the port that is being listened on.

Return Value:

None

Example:

```
RobotRaconteur.StartTcpServer(3452);
```

RobotRaconteur.StartTcpServerUsingPortSharer()

Starts the TcpTransport server listening for connections using the port sharer service.

Parameters:

None

Return Value:

None

Example:

```
RobotRaconteur.StartTcpServerUsingPortSharer();
```

RobotRaconteur.IsTcpPortSharerRunning()

Returns if the port sharer is running

Parameters:

None

Return Value:

(`int32`) - 1 if the port sharer is running, 0 otherwise.

Example:

```
running=RobotRaconteur.IsTcpPortSharerRunning
```

`port`=**RobotRaconteur.GetTcpListenPort**()

Returns the port that the TcpTransport is listening on.

Parameters:

None

Return Value:

(`int32`) - The port.

Example:

```
port=RobotRaconteur.GetListenPort();
```

RobotRaconteur.LoadTlsNodeCertificate()

Loads the TLS node certificate based on the NodeID of the MATLAB session. Set using `RobotRaconteur.StartLocalServer`.

Parameters:

None

Return Value:

None

Example:

```
RobotRaconteur.LoadTlsNodeCertificate();
```

RobotRaconteur.IsTlsNodeCertificateLoaded()

Returns if the port sharer is running

Parameters:

None

Return Value:

(`int32`) - 1 if the certificate is loaded, 0 otherwise.

Example:

```
loaded=RobotRaconteur.IsTlsNodeCertificateLoaded
```

RobotRaconteur.StartCloudTransportAsClient()

Connects to the *Robot Raconteur Cloud Client* as a client. The Cloud Client allows the node to connect to the Robot Raconteur Cloud.

Parameters:

None

Return Value:

```
RobotRaconteur.StartCloudTransportAsClient()
```

Example:

RobotRaconteur.StartCloudTransportAsServer()

Connects to the *Robot Raconteur Cloud Client* as a server. The Cloud Client allows the node to connect to the Robot Raconteur Cloud.

Parameters:

None

Return Value:

```
RobotRaconteur.StartCloudTransportAsServer()
```

Example:

RobotRaconteur.IsCloudClientRunning()

Returns if the cloud client is running.

Parameters:

None

Return Value:

(`int32`) - 1 if the cloud client is running, 0 otherwise.

Example:

```
running=RobotRaconteur.IsCloudClientRunning()
```

RobotRaconteur.RegisterService(*name*, *type*, *obj*, *security*)

Registers a MATLAB classdef object as a service. Optionally uses a string of authentication data to provide password security.

Parameters:

- *name* (`string`) - The name of the service.
- *type* (`string`) - The fully qualified type of the object
- *obj* (`*`) - The classdef object to register as a service
- *security* (`string`) - (optional) The authentication data. Uses the same format as PasswordFileUserAuthenticator.

Return Value:

None

Example:

```
RobotRaconteur.RegisterService('MatlabTestService',
    'example.MatlabTestService.MatlabTestObject',o);
```

RobotRaconteur.CloseService(*servicename*)

Closes a service.

Parameters:

- *servicename* (`string`) - The name of the service to close.

Return Value:

None

RobotRaconteur.ProcessServerRequests(*timeout*)

Executes pending requests from clients. This must be called manually because of the threading in MATLAB. The timeout is how long the command will wait for requests.

Parameters:

- *timeout* (`double`) - The timeout in seconds. Maximum is 10 seconds.

Return Value:

None

Example:

```
RobotRaconteur.ProcessServerRequests(1);
```