8 MATLAB INTERFACE

8.1 Introduction

The LFS includes an integrated interface to Mathworks MATLAB. The interface enables students to design and develop prototype aircraft flight control code, written as M-FILEs and executed in MATLAB, where the output of the flight control code is transmitted to the flight simulator to overriding manual or automatic flight control inputs (elevator, aileron, rudder and throttle). This *external interface* between the flight simulator and MATLAB provides a convenient method of rapid prototyping flight control code in the MATLAB environment without the need to access, develop or recompile the flight simulator software. Additionally, the interface provides an accessible way for students to learn and test automated flight control concepts in an environment with which they may already be familiar.

This section comprises two parts, which are intended for two different groups of users:

1. Users developing and testing flight control code using MATLAB (e.g. students, educators, researchers), and
2. Developers needing to understand, modify or extend the interface code that connects the flight simulator to MATLAB (e.g. simulator support staff).

8.2 User Guide

Use this guide if you intend to use MATLAB scripts to develop flight control laws and test the output using the flight simulator.

8.2.1 Overview of the Interface and Components

Connecting MATLAB to the flight simulator is achieved by use of a custom *MEX* file that passively listens for flight simulator UDP packets and returns a UDP packet to the flight simulator at the appropriate time. A precompiled MEX file is provided which requires the 64-bit version of MATLAB running on Microsoft Windows (7, 8, 10) and is connected to the dedicated flight simulator LAN - see Section 2.3. No additional MATLAB toolboxes are required.

Required files:

fslink.mexw64	Precompiled MEX file.
simctrl-template.m	Template script.
simctrl-constants.m	Script containing constants used within **simctrl-template.m**

These files should be copied from the flight simulator external interface resources directory into a working directory. The working directory should be added to the MATLAB path. There is no need to alter or recompile the fslink.mexw64 file. The copy of **simctrl-template.m** script file in the working directory should be renamed, for example **simctrl-speedcontrol.m**.

8.2.2 MATLAB Interface M-File Script

A template M-FILE script is presented as an example to assist with familiarisation of the Flight Simulator – MATLAB interface. This template should be copied and renamed, rather than being overwritten. This walk-through of the script describes the basic structure of the M-FILE and the necessary MEX file calls required to establish a link to the flight simulator.

At the start of the file, there is a call to another M-FILE script. The purpose of this script is to define constants that are used throughout this template script. The variable **done** is used as a condition to control the main loop. It is set to 1 when MATLAB mode is disabled in the IOS, causing the loop to exit cleanly. The variable **matlabMode** is read from the flight simulator data every frame and indicates that the IOS has activated the flight simulator MATLAB interface. The variable **matlabRunning** is set to 1 if **matlabMode** is ever true. If **matlabMode** subsequently becomes 0 because the IOS has disabled the MATLAB interface, **matlabRunning** remains 1. The Boolean state of **matlabMode = 0** and **matlabRunning = 1** provides the necessary condition to determine that MATLAB mode has been disabled at the IOS.

```
% Define the constants used in the MEX file interface
simctrl-constants;

% Main loop control
done = 0;

% Gets set to 1 when the IOS activates MATLAB mode
matlabMode = 0;

% Set to 1 when matlabMode 1 detected. Used to detect IOS has deactivated MATLAB mode
matlabRunning = 0;
```

The following line is the first call to the MEX file interface. Constants are used to inform the MEX file which operation to execute and to pass a parameter specific to that operation.

```
% Open the UDP port for the flight simulator connection
fslink(fslink_open,fslink_defaultport);
```

Here, the script enters the main loop of waiting for the data from the current frame of the flight simulator, executing any control calculation, and returning data back to the flight simulator in the form of control inputs.

```
%
% Main Loop - It is advised to not execute a Ctrl-C at the MATLAB prompt.
% The script will exit in an orderly way via the IOS. Only in the event that
% the flight simulator has unexpectedly stopped, should Ctrl-C be used.
%
while done == 0
```

Several calls to the MEX file interface are made before any script code is used in the computation of a control law.

```
% Wait for the current flight simulation iteration packets to arrive.
fslink(fslink_recv);

% Extract the received data from the MEX file into a Matlab array - DIN (Data IN)
DIN = fslink(fslink_dataget);
matlabMode = DIN(fslink_Active); % Flight simulator MATLAB mode state.
```

If the data received from the IOS indicates that MATLAB mode is enabled, the script executes the block containing the flight control law code, otherwise it waits for MATLAB mode to be enabled.

```
% Only compute and send data to the flight simulator if mode is active
if matlabMode == 1

    % Flag that MATLAB mode on IOS has been set. Once set, it remains 1
  % If the IOS switches MATLAB mode off, matlabMode = 0 and matlabRunning = 1
  % signals that the script exit.
  matlabRunning = 1;

  %
  % CUSTOM FLIGHT CONTROL CODE STARTS HERE
  %
```

This code marks the beginning of the region in the script where user code can be written.

The output from a prototype control law needs to be formed as a small packet to be sent to the flight simulator. The MEX file interface requires a four element array containing values for aileron, elevator, rudder and throttle position, in that order. If the control law requires that any of the flight control inputs are unaltered, then those values can be returned using the data in the DIN array. See the examples in sections 8.2.4 and 8.2.5.

```
% Finally, prepare the control data array for the MEX file - DOUT (Data OUT)
% This should be the last line in the custom code

DOUT = [0.0, 0.0, 0.0, 1.0];

%
% CUSTOM FLIGHT CONTROL CODE ENDS HERE
%
```

Flight controls are now passed back to the MEX file to be sent to the flight simulator. The values returned to the flight simulator will override the usual flight control inputs used by the flight model in the next frame of the flight simulator.

```
% Pass the control data to the MEX file
fslink(fslink_dataset,DOUT);

% Send the control data to the flight simulator
fslink(fslink_send);
```

As soon as the script detects that MATLAB mode has been disabled from the IOS, the main loop of the script exits. The remaining lines in the script close the network connection and clear the memory used by the MEX file.

```
% Close UDP connection
fslink(fslink_close);

% Clean up the memory used by the MEX file
clear fslink;
```

8.2.3 Executing a Script

In order to execute a flight control script (assuming no M-FILE control script bugs are present), the following sequence should be followed to ensure that the flight simulator is able to receive data from MATLAB:

1. Start the flight simulator as described in **Section 4.4 Start Up Procedure**
2. Restore the flight simulator to a position and state suitable for testing the control script. The restore operation is found in **Section 7.2.1 Reposition-Restore**. Place the simulator in the HOLD state.
3. Start the MATLAB script, either via typing the script name at the MATLAB command prompt, or by clicking the Run button. If the script has started and initialised the MEX file component correctly, the following output will be seen in MATLAB. These lines show that the interface to the flight simulator has been established and that the script is now waiting for the flight simulator to been switched into MATLAB MODE. If your output does not show this, then the network connection to the flight simulator has probably failed to have been established. In this case, follow the procedure outlined in **Section 8.2.6, Restarting the Flight Simulator - MATLAB Interface Component.**

```
Calling Function MEX_UDP_START
Port : 54321
UDP_Start : Matlab UDP connection ready for datagrams 54321
rv = 1
```

4. At the IOS, activate the flight simulator MATLAB mode as described in **Section 7.1.7 Master-Matlab.** Selecting the ON option (followed by OK) notifies the simulator that the MATLAB PC should be included in the protocol, receiving data packets from MATLAB to provide the flight control inputs.
5. The flight simulator will still be in the HELD state and the MATLAB script will be executing, but receiving unchanging data from the simulator. In turn, the MATLAB script will now be executing the custom control code and sending data back to the flight simulator. Remove the flight simulator from the HELD state. The flight control script will now be providing the flight control inputs for the simulator.
6. If the simulation is complete (for example, modifications to the script are required) **DO NOT type Ctrl-C** at the MATLAB command prompt, as this will violate the protocol with the flight simulator and cause the other simulator nodes to 'hang'. Rather, switch the flight simulator to the HELD state, and then deselect the flight simulator MATLAB mode as described in **Section 7.1.7 Master-Matlab.** This will take the simulator out of MATLAB mode cleanly and also inform the MATLAB script to stop and disconnect cleanly.
7. The simulator should still be operating, but in the HELD state. Modifications can now be made to the flight control script, or a new script developed.
8. To test the modified script, restart this process from step (2) above.

8.2.4 Example 1. Echoing the Flight Controls

A portion of an M-FILE script is presented (**simctrl-echo.m**) which receives data from the flight simulator and returns the same flight control inputs back to the flight simulator. This simple script demonstrates an important concept - reading and writing data to and from the flight simulator.

As described in the template script, a MATLAB array, DIN, is created and populated with the data received from the flight simulator. Variables are then created which store a local copy of the current flight controls by accessing the DIN array, with the specific indices defined in **simctrlconstants.m**.

```
% Extract the received data from the MEX file into a Matlab array - DIN (Data IN)
DIN = fslink(fslink_dataget);

% Access simulator data
de = DIN(fslink_Elevator);
da = DIN(fslink_Aileron);
dr = DIN(fslink_Rudder);
dt = IN(fslink_Throttle);
```

Because this example is only intended to demonstrate how to access simulator data, the retrieved data is returned back to the flight simulator by filling the array DOUT with the values that were previously read. If this script is executed as described in **Section 8.2.3 Executing a Script**, the flight simulator should operate under manual flight control for elevator, aileron, rudder and throttle inputs.

```
DOUT = [da, de, dr, dt];
```

8.2.5 Example 2. Altitude Hold Control Law

A portion of an M-FILE script is presented (**simctrl-altitudehold.m**) which implements a basic altitude hold control law. When executing this code, it is advisable to position the flight simulator model at a different altitude (e.g. ±500 feet) from the reference altitude defined in the altitude control law M-FILE script.

Near the start of the script, global variables defining a reference altitude and elevator limits are declared and initialised;

```
Href   = -2500.0 * 0.3048;   % 2500 ft -> m
de     = 0.0;
demin  = -0.4;
demax  = 0.4;
```

As in the previous example 8.2.4, data needed for the altitude hold flight control law is extracted from the array DIN.

```
U     = DIN(fslink_U);  % U
Udot  = DIN(fslink_Udot);  % Udot
H     = DIN(fslink_Altitude);  % Altitude
pitch = DIN(fslink_Pitch);  % Pitch
alpha = DIN(fslink_Alpha);  % Alpha
q     = DIN(fslink_Q);  % Q
Vd    = DIN(fslink_Vd);  % Vd
if ( U < 0.1 )
    U = 0.1;
End
```

The script lines that follow implement an altitude hold control law. At the end of this block of code, the elevator command (in the range -1.0 to 1.0) is includes in the array DOUT, to be transmitted to the flight simulator. Note that this control law only acts as an elevator command and sets the aileron and rudder commands to zero. These zeros values represent a neutral flight control input, in other words the flight control column is centered. Also note that the throttle input is set to echo the current throttle setting.

```
VSref = -0.08333*(Href - H);
               if VSref > 5.08
    VSref = 5.08;
elseif VSref < -5.08
    VSref = -5.08;
end

% Vertical speed controller
FPAngC = VSref / U;

% Flight path angle controller
pitchC = FPAngC + alpha;

% Pitch angle controller (converts pitch angle error into pitch rate demand)
qC = 0.2*(pitchC - pitch); % Pitch rate commanded

% Pitch rate controller (converts pitch rate error into elevator command)
dedot = -2.0*(qC - q); % Elevator change from pitch rate error
de = de + 0.02*dedot; % Forward Euler integration

% Apply the elevator input saturation limits
% Limit: demin <= de <= demax
               if de > demax
    de = demax;
elseif de < demin
    de = demin;
end

% Finally, prepare the control data array for the MEX file - DOUT (Data OUT)
               % This should be the last line in the custom code

DOUT = [0.0, de, 0.0, DIN(fslink_Throttle)];
```

8.2.6 Restarting the Flight Simulator - MATLAB Interface Component

If during step 3, executing a Script, MATLAB fails to display the shown output, this situation indicates a network connectivity problem. In this case, check the LAN configuration. Additional steps that may resolve connectivity problems include:

1. Manually cleaning up the network connection held by the MEX component by typing the following commands at the MATLAB command prompt:

```
fslink(fslink_close);
```

2. Removing the MEX external interface component from MATLAB's memory by typing the following commands:

```
clear fslink;
```

The fslink MEX component is automatically reloaded when the flight control script is next executed.

8.3 Developer Guide

Use this guide if you intend to further develop or understand the implementation of the MEX interface between the flight simulator and MATLAB.

8.3.1 Technical Description of the Simulator – MATLAB Interface

This interface has been developed using MATLAB MEX files. MEX files extend MATLAB by providing access to functions developed in C/C++ or FORTRAN to act as if they are built-in MATLAB functions. As outlined in **Section 2, System Architecture** the flight simulator subsystems are distributed over a dedicated local network, which includes the MATLAB PC. The MATLAB component has been network enabled by developing a custom MEX function in C to support UDP networking primitives, including the necessary connection control and synchronisation required by the flight simulator networking protocol. This approach simplifies the M-FILE scripts and also avoids the need to use additional MATLAB toolboxes.

8.3.2 MEX File Interface Implementation

Developers interested in extending the functionality found within the **fslink.mexw64** MEX file should read this section. A source code *walk-through* follows, to describe the functionality provided in the MEX file. The interface implementation is found in the C file **fslink.c**, along with the associated header file **fslink.h**.

The entry point to a MEX file function is via the **mexFunction()** function. MEX files usually act as a single function within MATLAB, called in a script by the name of the MEX file without the extension. Refer to the MATLAB documentation for detailed description of the mexFunction() parameters. These parameters provide a means of setting the number of inputs and outputs for a function. The input and output parameters ***plhs** and ***prhs** are MATLAB mxArrays in which function inputs and outputs are populated. MEX files can only pass data to and from a MATLAB script using these arrays.

```
void mexFunction (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
```

Within the mexFunction, a *switch yard* provides a multi-function capability in comparison with the usual single function MEX file. The first element of the input array, **prhs**, contains an identifier defining the sub-function to call. These identifiers are C macros, defined at the top of the file, corresponding to the function identifiers defined in **simctrl-constants.m**.

```
#define MEX_UDP_START 1
#define MEX_UDP_STOP 2
#define MEX_UDP_SEND 3
#define MEX_UDP_RECV 4
#define MEX_UDP_DATASET 5
#define MEX_UDP_DATAGET 6
```

The switch yard, shown below, calls the sub-function corresponding to the **prhs[0]** element. These sub-functions are regular C functions. In this case, these sub-functions deal with the UDP networking protocol used to connect to the flight simulator. Each sub-function can take different parameters, but it is important to remember which parameters map to each element in the input array **prhs[]**.

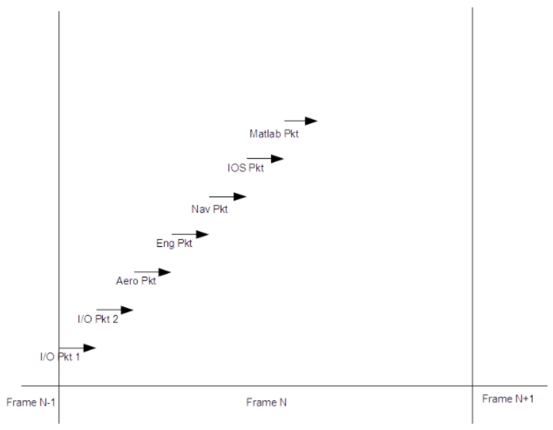
```
functionId = (int)mxGetScalar(prhs[0]);
switch(functionId) {
        case MEX_UDP_START:
                mexPrintf("Calling Function MEX_UDP_START\n");
                port = (unsigned short)mxGetScalar(prhs[1]);
                mexPrintf("Port : %d\n", (int)port);
                rv = UDP_Start(port);
                mexPrintf("rv = %d\n", rv);
                break;
        case MEX_UDP_STOP:
                mexPrintf("Calling Function MEX_UDP_STOP\n");
                UDP_Stop();
                break;
        case MEX_UDP_SEND:
                rv = UDP_Send();
                break;
        case MEX_UDP_RECV:
                rv = UDP_Recv();
                break;
        case MEX_UDP_DATASET:
                /* create a pointer to the real data in the input matrix  */
                inMatrix = mxGetPr(prhs[1]);
                /* get dimensions of the input matrix */
                ncols = mxGetN(prhs[1]);
                UDP_Data_Set(inMatrix);
                break;
        case MEX_UDP_DATAGET:
                /* create the output matrix */
                plhs[0] = mxCreateDoubleMatrix(1,(mwSize)MEX_OUTPUT_DATANUM,mxREAL);
                /* get a pointer to the real data in the output matrix */
                outMatrix = mxGetPr(plhs[0]);
                UDP_Data_Get(outMatrix);
                break;
}
```

To add further sub-functions, additional function identifiers can be created (C macros at the top of **fslink.c** and the equivalent constants in **simctrl-constants.m**) and their respective case blocks can be added to the switch yard.

The following functions deal with setting and acquiring data in a form that is compatible with MATLAB. The first function (UDP_Data_Set) populates the data in the return packet sent to the flight simulator and the second function (UDP_Data_Get) extracts data from the packets received from the flight simulator.

```
void UDP_Data_Set(double *data)
void UDP_Data_Get(double *data)
```

The remainder of the interface is composed of functions that handle the networking aspect of the interface. UDP_Start and UDP_Stop create and close the sockets required for receiving and transmitting UDP packets. The UDP_Send function transmits the ProtoPkt structure object that was populated via the UDP_Data_Set function. The most complex function, UDP_Recv, contains loops to ensure that, when receiving packets from the simulator, the MATLAB interface receives each of the flight simulator packets in the necessary order to maintain packet consistency across a single frame, or iteration, of the flight simulator. The first loop causes this function to block until the packet from I/O system 1 has been received. Once received, the MEX interface is now synchronised to the current simulation frame and then waits for packet from I/O system 2, followed by packets from the Aero, Eng, Nav and IOS computers. Note that this function will block until each of these packets has arrived. The following diagram (not to scale) shows the sequence of packets arriving at the MATLAB node in a single frame and also the transmission of the MATLAB packet.



In the event that the flight simulator unexpectedly stops with MATLAB mode enabled, it is likely that MATLAB will become blocked and will not respond to a Ctrl-C at the prompt. In such an event, MATLAB will require restarting.

```
int  UDP_Start(unsigned short);
void UDP_Stop(void);
int  UDP_Send(void);
int  UDP_Recv(void);
```

Developers needing to access additional variables from the flight simulator can expand the function UDP_Data_Get with the necessary assignments from the data available from the AeroPkt, NavPkt, IOSPkt and IOPkt and alter the macro MEX_OUTPUT_DATANUM to reflect the number of data elements being retrieved. This macro is used when allocating the MATLAB array used to provide the data for the calling M-FILE script.

8.3.3 Modifying and Recompiling the Mex File Interface

In the event that a change to the MEX file is required, for example, the MEX file requires new functionality, the modified C source files will need recompiling. A compiler compatible with MATLAB is required (refer to the MATLAB documentation for a list of compatible compilers). The supplied MEX file, **fslink.mexw64**, was compiled with a Windows version of the GCC (Gnu

Compiler Collection) C compiler. GCC is available for Windows through suites such as MinGW. The Matlab PC has a 64 bit version of MinGW installed.

To configure MATLAB to recognise your compiler, please refer to the MATLAB documentation. Comprehensive instructions exist for MS Visual Studio. However, there is minimal formal documentation for using GCC. A short description of this process now follows (which is only applicable to users of GCC on Windows). Note, these instructions only have to be executed once. Subsequent recompilations of the MEX file will not require these steps to be followed.

i. Locate the XML configuration file required to enable MATLAB to use the installed compiler. This file is found in the currently logged-in users profile data. This file has been configured to work with the current installation of MinGW.
E.g. C\Users\SIM8\AppData\Roaming\Mathworks\MATLAB\R2015a\mex_C.xml
Note, on the LFS, the user profile may be referred to as Cranfield and not SIM8.
ii. At the MATLAB command prompt, type:

```
mex –setup :C\Users\SIM8\AppData\Roaming\Mathworks\MATLAB\R2015a\mex_C++_mingw-w64.xml
```

Follow the instructions output at the MATLAB command prompt. You will be asked to choose which compiler MATLAB will use when building MEX files. Choose GCC. MATLAB is now configured to use GCC to compile MEX files.

To compile the flight simulator MATLAB interface MEX file (fslink.c), in MATLAB, change to the directory where the C source files are stored and type the following at the MATLAB command prompt.

```
mex '-IC:\mingw-w64\mingw64\x86_64-w64-mingw32\include' fslink.c C:\mingw-w64\mingw64\x86_64-
w64mingw32\lib\libwsock32.a
```

If your installation of GCC resides elsewhere, you will have to alter the above line to refer to the correct *include* directory and the correct Winsock library. If no errors are encountered, a new fslink.mexw64 file will have been created. This module can now be used as outlined in Section 8.2.