FLIGHTLAB Real Time Application Tutorial

16 June 2006

Contents

1 Standalone Model

Generating a standalone model allows a FLIGHTLAB model to be used in end-user applications without requiring the Scope run-time system. The standalone model is object code that can be loaded into a C application. Current FLIGHTLAB standalone models support execution in Windows (32 bit), Linux, and Irix operating environments. There are three aspects involved in generating a standalone model: generating the FCM (FLIGHTLAB Code-Gen) standalone model, creating the C header file linking the API, and creating applications.

1.1 Generating an FCM Model

The FCM model is created through the FLIGHTLAB development system. The first step in creating an FCM model is to define and create the interface. The interface allows the model to communicate with the application by sending and receiving data through varlists. The syntax for creating a varlist is **varlist @*varlistname* =** followed by the list of variables desired for that varlist, separated by commas. Once the desired varlist has been created, it must be dumped to the FCM model using the following syntax: **fcmdump(@*varlistname*)**. When creating the desired interface, the user should keep in mind that there is a default interface that should always be used when building a standalone model for PilotStation applications. This interface is located in the PilotStation directory under the scripts directory in a file called **pws-configure.exc**.

Once the script for generating the interface has been created, the standalone model needs to be generated. To do this, the FLIGHTLAB model should be loaded into the development system. The interface script should then be executed. Once this is done, the initial conditions should be set and the model should be trimmed. This will put the model in a good condition for loading into the application. The final step is to dump the FLIGHTLAB model into an FCM model. The syntax for this is **fcmdump("*modelname*.fcm");**. This will generate a standalone model data file in the current working directory. An example script for dumping a model into an FCM file follows:

```
//File:   fl-fcm.exc
//Desc:   Generation of an FCM model

//Load a model
 exec("$FL_DIR/flme/models/articulated/arti-rgd-3iv-qs.def",1);

//Trim the model in hover default condition
 exec("xamodeltrim.exc",1);
```

```
//Configure a model for use with FCMDRIVER
path("$FL_DIR/fcmdriver/scripts")
exec("$FL_DIR/fcmdriver/scripts/fcmdriver.configure",1)

//Configure a model for use with PilotStation
exec("$PWS_DIR/scripts/pws-configure.exc",1)

//Initialization
init;

//Dump the model
fcmdump("arti.fcm")

//EOF
```

This script loads a model called **arti-rgd-3iv-qs.def**, trims it, configures the FCM driver, sets the default PilotStation interface, and then dumps the model to a standalone FCM model called **arti.fcm**.

1.2 Creating the C Header File and Linking the API

Once the FCM model has been created, the interface between the standalone model and the application needs to be built. This consists of two pieces: the C language header file, which defines the relationship between the varlists and the C data structures, and the Application Programming Interface (API), which is the programming interface between the model and the user application.

For the header file, there should be a one to one correspondence between the Scope varlists and the C data structures. For example, if a varlist is constructed as follows:

```
pushg(world_model_airframe_cpg_xaout)
  varlist @STATES "Aircraft states" = posxi,posyi,poszi, vxb,vyb,vzb, ..
                              phi,theta,psi, p,q,r;
  fcmdump(@STATES);
popg;
```

a corresponding C data structure must exist in the header file as follows:

```
struct STATES /* Aircraft states */
{
    double posxi; /* Inertial X [ft] */
    double posyi; /* Inertial Y [ft] */
    double poszi; /* Inertial Z [ft] */
    double vxb;  /* Vx of body [ft/sec] */
    double vyb;  /* Vy of body [ft/sec] */
    double vzb;  /* Vz of body [ft/sec] */
    double phi;  /* Euler phi [rad] */
    double theta; /* Euler theta [rad] */
    double psi;  /* Euler psi [rad] */
    double p;    /* Body roll rate, p [rad/sec] */
    double q;    /* Body pitch rate, q [rad/sec] */
    double r;    /* Body yaw rate, r [rad/sec] */
};
```

Note that the FLIGHTLAB data are all double precision and, therefore, the C data structures must be double precision as well.

When working with ART to develop an FCM model, FLIGHTLAB developers will often generate a “magic number” in the header file. This number is then checked against the header file when the model is loaded in order to ensure that the correct header file is being used and that the interface will work correctly. To disable this check, users can set the “magic number” to 0000.

FLIGHTLAB provides an API utility with which to interface the standalone model with user applications. The API handles the communication between the standalone model and the user-created driver for their application. When linking the API, there are two important things to keep in mind:

- The system header file, **fcm.h**, which is located in **$FL_DIR/include/**, must be included in all C code.
- Link with **-lfcm** (**FL_DIR/libFL_MACHTYPE/libfcm.a**).

There are several data types used in the API.

- **FCM_FILE**: This is the master handle.
- **FCM_MODEL**: This is the model instance handle. It contains the complete state of a model instance.
- **FCM_OPERATION**: This is an operation handle. It is used to invoke simulation operations, such as stepping the model.
- **FCM_VARLIST**: This is a variable list handle. It is used to access structured data
- **FCM_FIELD**: This is a field handle;

Using these data types, the model can be loaded, operations can be run, and data operations can be performed.

- Call **fcm_open** to obtain an **FCM_FILE** handle.
- Obtain other handles from the **FCM_FILE**, such as variable lists, operations, and fields.

- Call **fcm_create** to create an **FCM_MODEL** instance.
- Call **fcm_destroy()** to destroy an **FCM_MODEL** instance.
- Call **fcm_close()** to free all resources associated with an **FCM_FILE**.
- To obtain operation handle:

  FCM_OPERATION *op_handle* = fcm_lookup_operation(*fcm*, "*op-name*")

- To call an operation:

  fcm_invoke(*model_handle*,*op_handle*)

  where *model_handle* is the model instance handle obtained from **fcm_create()**.
- To obtain a variable list handle:

  FCM_VARLIST vl_handle = fcm_lookup_varlist(*fcm*, "@*name*")

- To read model data:

  fcm_read(*model*, *vl_handle*, &*structvar*, *size*)

- To write model data:

  fcm_write(*model*, *vl_handle*,&*structvar*, *size*)

- To save model test conditions at a checkpoint:

  fcm_checkpoint(*model_handle*, *filename*)

- To restore model test conditions from a checkpoint:

  fcm_restore(*model_handle*, *filename*)

- To reload a model:

  fcm_reload(*model_handle*)

The following is an example C API:

```
#include "fcm.h"
... /* Initialization: */
FCM_FILE       fcm       = fcm_open("articulated.fcm");
FCM_OPERATION  op_step   = fcm_lookup_operation(fcm,"STEP");
FCM_VARLIST    vl_states = fcm_lookup_varlist(fcm,"@STATES");
FCM_MODEL      model     = fcm_create(fcm);
double         state_buf[N_STATES];
... /* Run-time, once per frame: */
fcm_invoke(model, op_step);
fcm_fetch(model, vl_states, state_buf, N_STATES);
... /* Cleanup: */
fcm_close(fcm);
```

1.3 Creating Applications

There are four other items that must be addressed in order to fully integrate the standalone model with the user application.

- Set the environment variable for the system component library.
- Develop a driver to interface between the user application and the API.
- Include the system header file in the driver.
- Create a makefile for compiling the standalone model with the appropriate header file and interface.

In order to run the standalone model, the environment variable **FL_FCM_COMPONENTS** must be pointed to the system component library file. The command to do this is

```
setenv FL_FCM_COMPONENTS $FL_DIR/lib/linux/flcomp.so
```

The **$FL_DIR/lib/linux/** directory is the Linux glibc-2.3 library, which is the default library for FLIGHTLAB 3.1. FLIGHTLAB also supports three other operating systems: Linux glibc-2.2 **$FL_DIR/lib/linux-glibc22**, Unix **$FL_DIR/lib/irix6**, and Microsoft Windows **$FL_DIR/lib/win32**. The user should set their environment variable to the appropriate directory.

A driver must be written in order to develop the user's application. An example driver, **driver.c**, can be found in **$FL_DIR/examples/fcm/**. Note that the system header file, **fcm.h**, must be included in the driver. This file can be found in **$FL_DIR/include/**.

Finally, a makefile can be prepared to facilitate the generation of the standalone model and compilation of the driver program. An example, **Makefile**, can be found in **$FL_DIR/examples/fcm/**.

2 FCMConsole

The **FCMConsole** is a graphical interface for running **fcm** dynamic models. It runs the models only; any visual system to be used with the simulation must be started separately. To start the interface, type **fcmconsole &** at the command prompt. This will bring up the graphical interface window for the **FCMConsole**.

2.1 Options

By default, the graphical interface starts in the **Options** section of the interface. Once the interface window appears, the desired **fcm** model and configuration files need to be selected. To do this, click on the "..." button to the right of either the **Model** or **Config** fields. A new window will appear that will allow the user to browse through the directory structure to find the desired model or configuration file. Highlight the desired file by clicking it and then click the "Open" button.

Once the appropriate files have been selected, there are five options that the user may select from:

- **Verbose?** Selecting this option will generate more detailed messages in the log screen when the "Start" button is pressed.
- **Enable network?** Selecting this option will enable the model to utilize **netflc** for data communication over a local area network.
- **Enable PWS?** Selecting this option will enable the model to communicate with PilotStation.
- **Pilot inputs:**
  - **None** Selecting this option means that there is no input device for the model.
  - **Use control box?** Selecting this option will allow the user to fly the model using the control box.
  - **Use joystick?** Selecting this option will allow the user to fly the model using a joystick.
  - **Pilot inputs from remote host?** Selecting this option will allow the user to fly the model from a remote host.

Once the desired options have been selected, click the "Start" button to launch the model. As soon as the button is pressed, the interface will switch to the **Log** area and a series of messages will be displayed. Any error messages will be shown in red. Note that the **Config** and **Monitor** sections of the interface are now accessible.

2.2 Running the Model

To run the model and begin flying, click the "Run" button. To pause or reset the model to the last trim point, click the "Pause" or "Reset" buttons, respectively.

2.3 Log

The log screen displays status messages when the simulation is started using the "Start" button in the **Options** screen as well as when the model is stopped by pressing the "Stop" button.

2.4 Diagnostics

The **Diagnostics** screen can be utilized to check the interface for the model. After clicking on the **Diagnostics** tab, the different available diagnostic checks are available through the **Diagnostics** menu.

- Trace
  - **commands**
  - **model.controls**
  - **model.trim**
  - **off** This option turns the tracing function off.
  - **pilotin**
  - **simtime**
  - **udpcmd**
- **List FLCOMMS** This performs the **flcomms -l** command and lists the output in the **Diagnostics** screen.
- **FCM Information** This performs the **fcminfo -icLlo** command and lists the output in the **Diagnostics** screen.
- **Shared memory status** This performs the **ipcs** command and lists the output in the **Diagnostics** screen.
- **Clear FLCOMMS** This performs the **flcomms -stc** command, which clears the shared memory.
- **Remove shared memory** This performs the **ipcs -m** command, which removes the shared memory blocks.

2.5 Configuration

It is possible to configure some initial conditions for the simulation. To do this, click on the **Config** tab in the interface screen. This will bring up the configuration screen, which consists of three parts:

- Aircraft
  - X Inertial position
  - Y Inertial position
  - Z Inertial position
  - Heading
  - Airspeed
  - Gross weight
- Environment
  - Wind magnitude
  - Wind azimuth
  - Sea level pressure
  - Sea level temperature (degF)

For **fcm** models with no ship modeled, all the configuration entries will be zero. Once the configuration has been set as desired, press the “Apply” button at the bottom of the screen to apply the settings to the simulation. The model can then be trimmed, if desired, by pressing the “Trim” button.

2.6 IO

This screen presents the user with several tabs that contain fields which present the values of pre-selected model variables. To change an input value, enter the desired value in the field and press the “Apply” button. To get the current value of an output value, press the “Refresh” button. These can be done in real time.

2.7 Monitor

A real time monitor has been implemented in the **FCMConsole** interface. It can be configured to display in real time any variable contained within a FLIGHTLAB CPG group. Note that it is not necessary to use the monitor in order to run the model. To access the monitor, click on the **Monitor** tab in the interface screen. By default, there are no variables in the monitor when it is first opened. A blank screen is displayed with two buttons labeled with a right facing triangle followed by three dots, one in the top right hand corner and one in the bottom left hand corner. The top button enables the user to add multiple panes to the monitor as well as to add variables to each screen. The bottom button is for adding switch monitors, which appear as buttons along the bottom of the monitor screen. To add a pane or to add/delete variables to/from an existing pane, click on the top button. This will open a new window labeled “Configure slider...” At the top of this window is a field labeled “Pane.” This field tells the user which pane they are configuring. To the right of this field is a button labeled “New.” This allows the user to add a new pane to the monitor. Below the “Pane” field is another field labeled “Find.” This allows the user to search in the variable list for a specific variable. Below this is a window containing a list of all the variables available for the real time monitor. To the right are two buttons: “Add” and “Delete.” To the right of these buttons is another window which contains all the variables that have been added to the monitor. By default, this window is empty. To add a variable to this window, click on the desired variable in the left hand window and click the “Add” button. A variable can be removed from the monitor list by clicking it in the right hand window and clicking “Delete.” Below these two windows are four data entry boxes:

- **Field:** The name of the currently selected variable is listed in this box.
- **Min:** The user may specify the desired minimum value for the selected variable to be displayed in the real time monitor.
- **Max:** The user may specify the desired maximum value for the selected variable to be displayed in the real time monitor.
- **Current:** This box displays the current value of the selected variable.

When the real time monitor has been configured as desired, click the “Apply” button. This will close the “Configure slider...” window and apply the changes to the real time monitor screen in the **FCMConsole** interface.

The switch configuration screen works in much the same way, except that there are no panes to deal with and variables are only available from FLIGHTLAB CONFIGPAR groups. At the top of the screen is a “Find” field that works the same way as for the pane configuration. Below that are the same two windows separated by the “Add” and “Delete” buttons. Below this are two fields, rather than four.

- **Field:** The name of the currently selected variable is listed in this box.
- **Label:** The user can specify a label for the switch button. The default setting is the variable name.

When the switches have been configured as desired, click the “Apply” button. This will close the “Switch configuration...” window and apply the changes to the real time monitor screen in the **FCMConsole** interface. When the switch is off, the respective button will be a gray color. When the switch is turned on, it will change to green. A red color indicates that the specified variable could not be found in the **fcm** model.

3 Communication with External Processes

There are two communication protocols that can be used either with the FLIGHTLAB development system or a standalone model. The first protocol is FLCOMMS, a shared memory based interprocess communications facility used to transfer data between a FLIGHTLAB model and external applications. The other is NetFLC, which uses UDP to broadcast over the local area network to synchronize the FLCOMMS blocks on all participating hosts. Figures 1 and 2 show a graphical representation of how FLCOMMS and NetFLC manage the data communication, respectively.

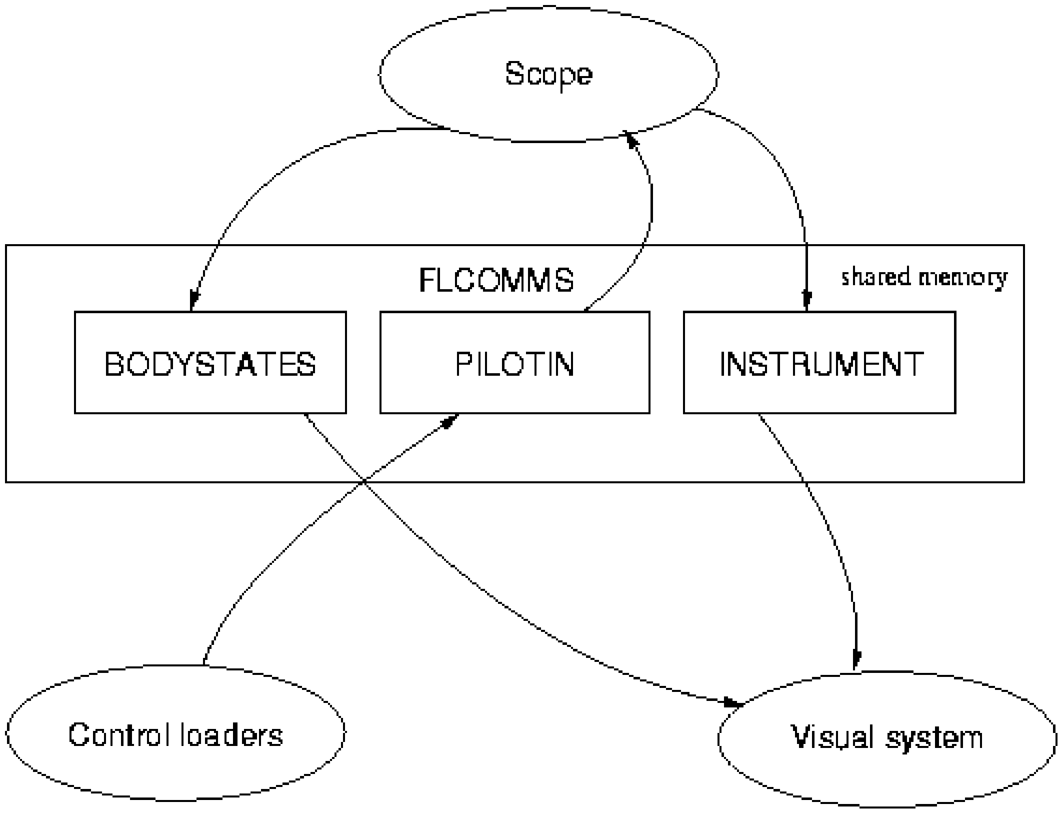

Figure 1: Graphical Representation of FLCOMMS Functionality

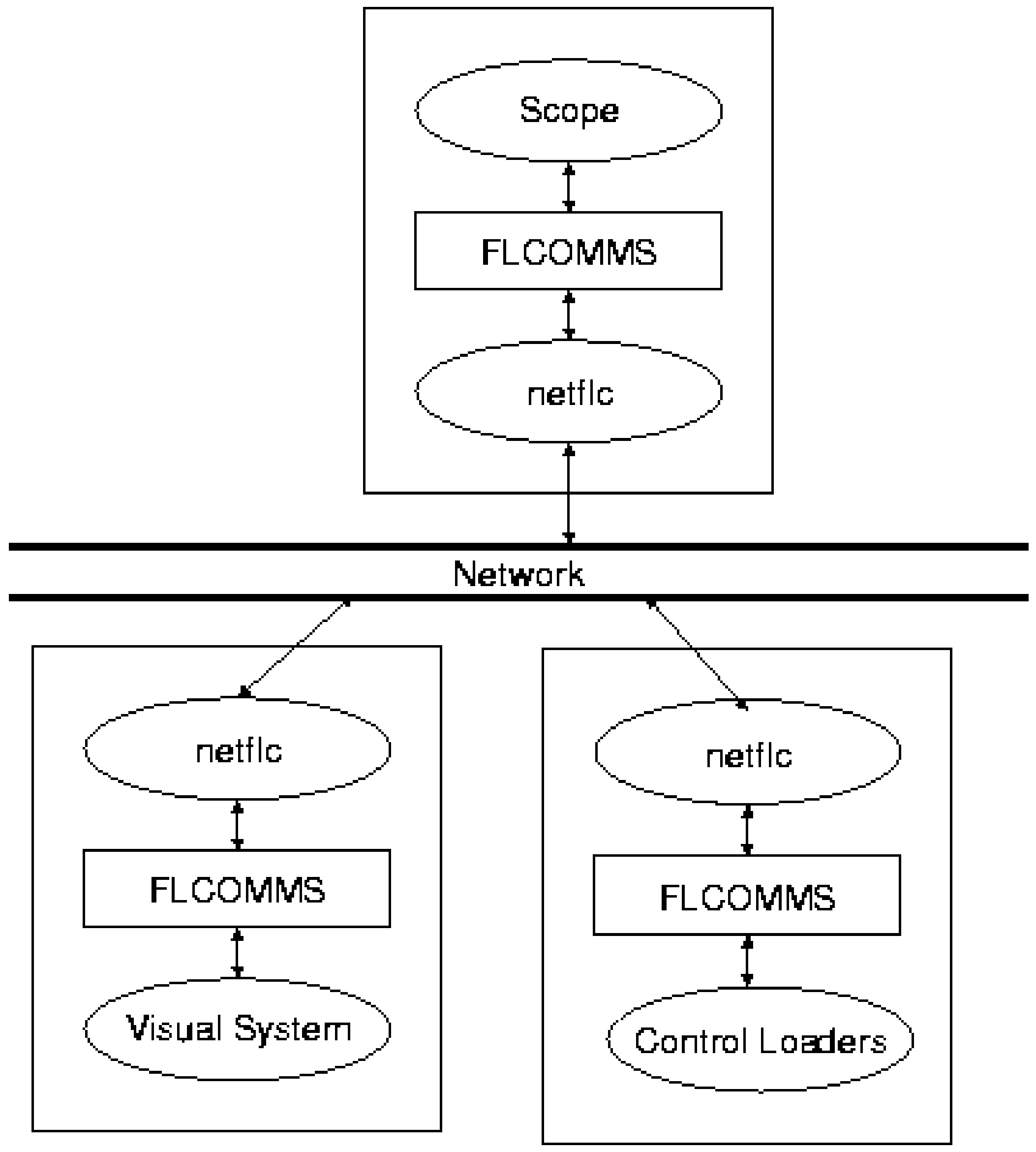

Figure 2: Graphical Representation of NETFLC Functionality

3.1 FLCOMMS

FLCOMMS was designed specifically for real time applications. It consists of a collection of data blocks, each of which is defined by name, size, type, and "magic number." Each data block corresponds to a varlist that was created as discussed in Section 1. For various options that can be used with FLCOMMS, type *flcomms -h* at the command prompt. When running a model through the development system, each FLCOMMS data block must be attached using the following commands:

- **attach(@*varlistname*,"w")** for outputs (the "w" tells the model to write data to the FLCOMMS block)
- **attach(@*varlistname*,"r")** for inputs (the "r" tells the model to read data from the FLCOMMS block)

When running a standalone model, the **fcmdump** command is used as described in Section 1.

As with the standalone model, an API is required to interface FLCOMMS with the user's application. FLIGHTLAB provides the utility to facilitate the application. When linking the API, there are two important things to keep in mind:

- The system header file, **flcomms.h**, which is located in **$FL_DIR/include/**, must be included in all C code.
- Link with **-lflcomms** (**FL_DIR/libFL_MACHTYPE/libflcomms.a**).

There are several operations provided by the API utility.

- Call **flc_attach(NULL,0);** at program startup.
- Call **flc_open(...)** to acquire data block handles.
- Use **flc_read(...)** and **flc_write(...)** to transfer data.
- Call **flc_close(...)** to release a handle (optional).
- Call **flc_detach();** before exiting the program (**not** optional).

To connect the user application to the FLCOMMS data blocks, use the following syntax:

```
FLC_HANDLE h = flc_open("name", size, magic, mode_flag | type_flag);
```

where

- *name* is the data block name.
- *size* is the size in bytes of data block.
- *magic* is the magic number.
- *flags* specifies the mode and type as follows:
  - **FLC_MODE_READ** opens data block for reading.
  - **FLC_MODE_WRITE** opens the block for writing.
  - **FLC_TYPE_DOUBLE**, **FLC_TYPE_INT**, etc. specify the data type.
  - **FLC_TYPE_MIXED**: anything else

  - Flags are combined with the bitwise-OR operator (|).

Note that Scope always uses **FLC_TYPE_DOUBLE** and that *name* is the variable list name without a leading @ sign (all uppercase).

Reading and writing operations are performed as follows:

- **flc_read(*handle*, *dst*, *nbytes*);**
- **flc_write(*handle*, *src*, *nbytes*);**
  - *handle* is the **FLC_HANDLE** returned from **flc_open(...)**
  - *src* and *dst* may be an array or pointer to a structure.
  - *nbytes* is the size of the data block in bytes.
- **flc_ready(*read_handle*)** returns 1 if the data block has been written to since the last time the handle was read.

An example FLCOMMS driver can be found in **$FL_DIR/examples/flcomms/** with a filename of **flcomms-example.c**. As with the standalone model, a makefile needs to be written in order to compile the API. An example, **Makefile**, can be found in **$FL_DIR/examples/flcomms/**.

3.2 NetFLC

NetFLC uses broadcast UDP over the local area network to synchronize FLCOMMS blocks on all participating hosts. It periodically broadcasts current values of all data blocks being written to on the host and listens for updates from other hosts. It also handles floating point and integer format conversions between different architectures. For a list of NetFLC options, type **netflc -h** at the command prompt.