Python HW Link

High Level Overview

As Flight Simulators increasingly offer support for API interfaces (eg DCS, P3D), it now is feasible to move away from using Joystick interfaces (along with fun of dealing with USB issues) to a distributed system using Ethernet. Such as approach also simplifies output interfacing (e.g. indicators and gauges) as tasks and processes can be divided up across a number of devices and technologies.

This project builds on the Python/Raspberry Pi project used in the Huey Warning panel.

Definitions

Primary Flight Simulator – the PC running an instance of the flight simulator

Primary Node – the Raspberry Pi node that communicates to the Primary Flight Simulator

Distributed Node – either a Raspberry Pi or Arduino (with Ethernet shield) that communicates to the Primary node.

Principles

* Use UDP for all communications where possible. This removes any possible performance issues associated with Nagle and TCP slow start. It also means components are loosely coupled, enabling them to be restarted without impacted other modules.
* Accept inputs from push, toggle, rotary, and rotary encoders.
* Outputs – analog and digital, text. All outputs are normalised before being send to output card/block. Should consider the format used by DCS-BIOS
* Multiple Sim support. As outputs are normalised the and loosely coupled now simulator support can be added without negatively impacting existing sim support. The receive interface from the Sim listens on unique ports, allowing the code to be running at all times.
* A shallow native shim is used to link the simulator to this hardware modules. As an example, LUA is used with DCS, for P3D Sim Connect used.
* Remote shutdown of all Pi nodes is provided through the master, once this has been invoked the nodes will shut down the OS, requiring either a hardware reset, and a power cycle to resume. Ideally outputs will display a checkerboard to reflect a shutdown command has been received.
* Indicator test, a single command will be supported to light all indicators, and perhaps cycle gauges.
* Downstream nodes should accept a request to report input switch position. On receipt of such a request the node will send a report of switch positions, probably at a rate of 20 per second. Need to consider reporting toggle switches in off position and three position switches.
* Will initially develop using DCS, and the variable names currently used by the A10
* The IP addressing of the ‘internal’ network (i.e. between the primary Flight Simulator computer) as well as between nodes will use the 172.16.1.X network. This enabled multiple flight simulators to share a common 192.168.X.X network. If there is only a single simulator the 172.16.1.X network can exist as a secondary address on the Primary network interface, if multiple Simulators share a network, then the internal network should use a different network interface on the Primary Computer.
* Mapping of physical inputs to simulator functions is performed on the Primary Pi node. This keeps the distributed nodes independent of flight sim, and relatively simple, enabling the use Arduino nodes as needed without adding unneeded complexity.
* Each distributed node will have a Unique identifier, which largely is used to uniquely identify different input modules.
* Each distributed node will maintain a state machine for its interfaces, sending only deltas to the Primary Node
* Distributed nodes receiving non-string values receive data as A=V:A1=V1:A2=V2

# Choice of platform for the hardware interface.

Whilst the Raspberry Pi offers the nicest development and troubleshooting environment, it lacks the high pincount found on an Arduino. The Arduino will require an Ethernet shield (and not all Ethernet shields are created equal, have run into issues with an IOT shield that had a poorly cooled chip which caused lock ups).

As the Arduino codes runs without an intermediate operating system, it offers the highest performance for IO related tasks such as driving stepper motors.

Have ran into issues with lockups in the 737 overhead display, possibly due to incorrectly terminated strings or invalid characters included in string. Strongly bounds checking will be performed on the Pi before strings are send to Arduino displays.

As we are not bound by the 128 input limit associated with either windows DirectX or the 32 input limit associated with FSUIPC, OverPro’s Arduino Joystick interface can be used with 256 inputs reducing the number of controllers needed for the pit. Hopefully a single controller for Port, Starboard and Forward zones can be used. OverPro’s code will be modified removing the usb interface, instead storing switch state and reporting deltas after completing a full scan of 256 inputs. Unsure if rotaries will be supported, no logical reason why not.

# Program tasks

The workload is divided into two programs – one dealing with inputs, the other with outputs. Both are fundamentally loops which briefly block awaiting receipt of a UDP packet either from the simulator/shim or from the input devices.

Command line parameters are used to get a debugging level as well as enable a configuration mode.

Configurations will be held in two separate files, the format of these files is yet to be determined, but JSON is mostly likely. The configuration files include:

1. IP Address code listens on (optional - if not explicitly configured 127.0.0.1)
2. Port Code listens on (optional – if not explicitly configured 7784 for input module and 7785 for output module. (still considering whether to leave export.lua sending direct to existing Arduino units)
3. Mapping of input to aircraft commands and aircraft outputs to physical displays/gauges.

# Protocols

Where possibly output values will be carried directly mapping to value to be displayed, and may be integer, float, or string. Indicators/Solenoids will be represented as a 0 or 1.

Data will be carried as AV pairs A1:V2, A2:V2. The data packet will be preceded with a D.

Operational tasks such as shutdown, reboot, refresh switch state will be preceded with a C (command), and a single operational task. Eg C9999

Operational Task

|  |  |  |
| --- | --- | --- |
| Task Id | Task |  |
|  |  |  |
|  |  |  |
|  |  |  |

Ikarus uses the following format

# If Sending Commands to DCS with Ikarus installed

# The values to be sent can be found

# C:\Program Files\Eagle Dynamics\DCS World\Mods\aircraft\Uh-1H\Input\UH-1H\joystick

# Structure is

# C - Command

# 15 - Cockpit Device Id

# 3003 - Unknown but seen in multiple places

# Switch Position

# Send to Port UDP\_PORT = 26027

# MESSAGE = "C15,3003,-1" - Turns Test Switch on - All warning lights

# MESSAGE = "C15,3003,0" - Turns Test Switch to centre

# MESSAGE = "C15,3003,1" - Turns Test Switch to Reset - clears caution on front panel

# Bright/Dim 15,3004

# Initiating Scripts on Pi Nodes

Instead of trying to start python directly from crontab, use shell script (usually my\_server) to

start things

To get the script to autostart

sudo crontab -e

And add line

@reboot sh /home/pi/Documents/Flightsim/Huey\ Caution\ Panel/my\_server 2>&1

Which results in the crontab file looking like

# Edit this file to introduce tasks to be run by cron.

..

..

#

@reboot sh /home/pi/Documents/Flightsim/Huey\ Caution\ Panel/my\_server 2>&1

Originally a separate shutdown script was operated, need to work out why it was commented out, possibly as the script below only ever reaches remoteshut.py after the receiver code exits, which is never…

### BEGIN INIT INFO

# Provides: my\_server

# Required-Start: $remote\_fs $syslog $network

# Required-Stop: $remote\_fs $syslog $network

# Default-Start: 2 3 4 5

# Default-Stop: 0 1 6

# Short-Description: Simple Web Server

# Description: Simple Web Server

### END INIT INFO

#! /bin/sh

# /etc/init.d/my\_server

export HOME

echo "Starting My Server"

cd /home/pi/Documents/Flightsim/Huey\ Caution\ Panel

sudo /usr/bin/python receiver\_004.py 2>&1 &

#sudo /usr/bin/python remoteshut.py 2>&1 &

exit 0

# Receive UDP Port utilisation

A number of these values are referenced from soic\_conv\_ExportStart.lua from DCS

|  |  |  |
| --- | --- | --- |
| Port | IP Address | Description |
|  |  |  |
| 26027 |  | Ikarus Port data is sent to |
|  |  |  |
| 7777 | 127.0.0.1 | SOIC on Primary Sim PC[[1]](#footnote-1) |
| 7784 |  | Input Codes listens on |
|  |  |  |
| 13135 | 192.168.1.105 | Fuel Hands on A10 |
| 13135 | 192.168.1.106 | Fuel Display (OLED on A10) |
| 13135 | 192.168.1.107 | Compass and Clock Analog hands |
| 13135 | 192.168.1.108 | Clock Digits |
| 13135 | 192.168.1.109 | General Stepper |

Notes from the Huey readme

Git Commands

Watch for untracked files

$ git add .

$ git commit .

$ git push

1. The SOIC port points to another Shim which maintains a TCP connection to the SOIC processes, converting the UDP payload into a TCP stream. Currently error handling does not address a restart of SOIC processes. [↑](#footnote-ref-1)