MakerPlane

MakerPlane CAN-FIX Avionics Tools

FiX Gateway, PyAvTools, CANArduino

Overview and Implementation

V1.0

John Nicol

11 Feb 2020

Document Changes		
Version	Changes	Author
1.0	Release Document	J. Nicol

<u>Contributors:</u>

I just put the document together, the real work and content comes from these fine folks:

Phil Birkelbach
Neil Domalik
Garrett Hershleb

Table of Contents

Table of Figures

Overview

Document Purpose

The purpose of this document is to provide an overview of the MakerPlane CAN-FiX Avionics environment. It provides a reference for developers that wish to create plug-ins or applications that use the MakerPlane tools and communication protocol.

This document has three main sections:

1. FiX Gateway;
2. PyAvTools; and
3. CANArduino.

Related Documentation

The following shows the related documents within the MakerPlane CAN-FiX Avionics environment.

1. CAN-Fix protocol specification;
2. MakerPlane CAN-FiX Avionics Tools – FiX Gateway, PyAvTools, CANArduino;
3. MakerPlane pyEFIS Application; and
4. MakerPlane pyAvMap Application.

System of Systems

A “system of systems” is defined as a collection of task-oriented or dedicated systems that pool their resources and capabilities together to create a new, more complex system which offers more functionality and performance than simply the sum of the constituent systems.

The MakerPlane solution as a system of systems that can not only be used as a complete avionics solution in an experimental aircraft, it can also be used as an innovation platform and test bed for avionics and aircraft systems. It is already a well-populated ecosystem with flight proven implementations including our Electronic Flight Information System (EFIS) called pyEFIS, pyAvMap moving map, Engine Information System (EIS), trim controllers, annunciators, audio solutions and so on.

The EFIS includes a multitude of configurable and extendable applications such as user instrument clusters, moving map display, engine monitoring and so on. It also has a 3-dimensional display of airports and runways within the primary flight instrument. Underpinning the system of systems is the CAN-FiX protocol that allows the different nodes to share data.

It is our hope that future innovators for Loss of Control in Flight (LOC-I) solutions may be able to quickly implement their solutions without needing to re-invent sensors, displays or gauges if they need them.

The diagram below shows the possibilities of implementing a MakerPlane avionics system.

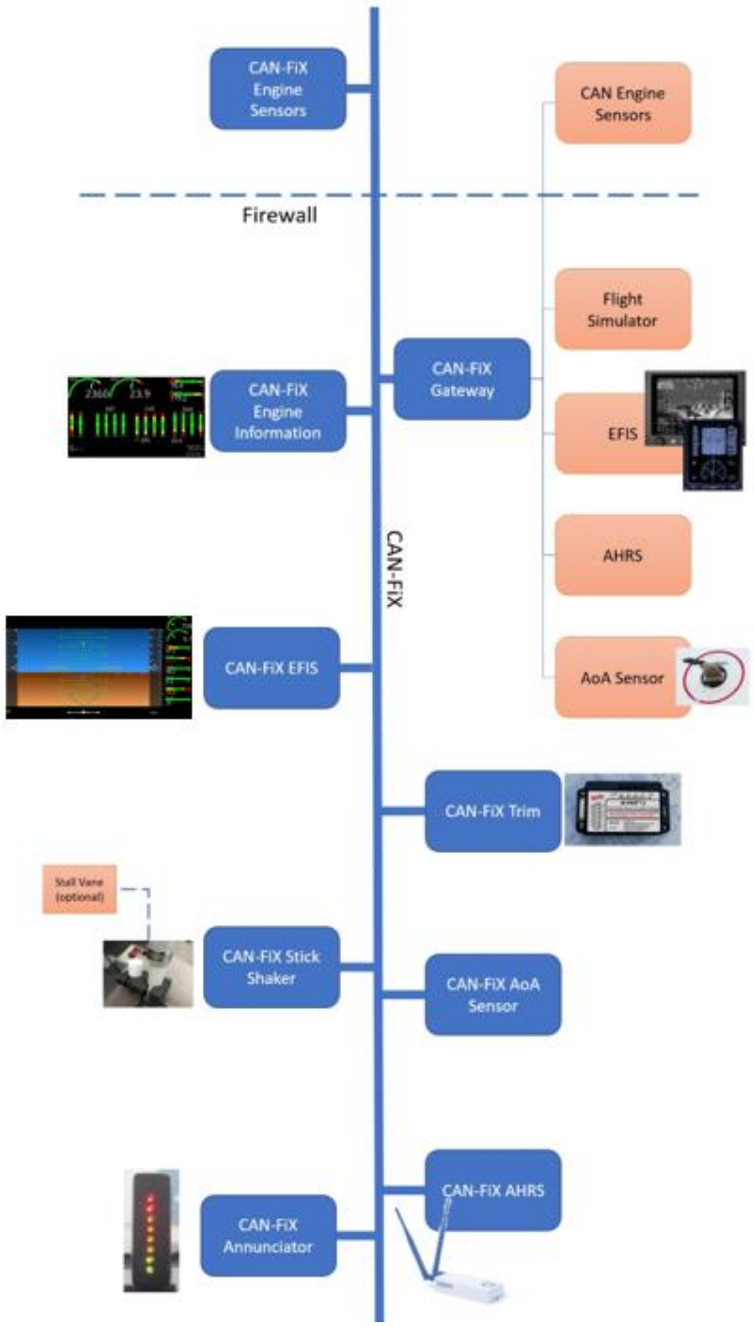

Figure 1. MakerPlane Avionics System of Systems.

CAN-FiX Overview

CAN® stands for Controller Area Network. It was developed originally as a robust message-based protocol for use in automotive applications. It has spread to other uses as well and is now found in medical equipment and industrial controls. FiX is an acronym for Flight information eXchange. It is a set of protocol specifications for exchanging information between aircraft avionics and flight systems. This specification and the protocols themselves are licensed under a Creative Commons license that allows anyone to modify and redistribute these documents without charge. The full specification can be found on the MakerPlane GitHub site here:

https://github.com/makerplane/canfix-spec

This is a community supported endeavor, with the primary goal of providing a standard method for avionics and flight control systems to communicate with one other in a vendor neutral way. The specifications and protocols are primarily geared toward the Experimental Amateur Built (EA-B) aircraft community.

Keeping the specification open and free allows airplane builders to create their own devices and write their own software that will be able to communicate with other devices without need to pay for specifications or licenses. It also encourages collaboration in the development and improvement of the protocols themselves. We have incorporated support for electric propulsion for example and extended ADS-B and weather information.

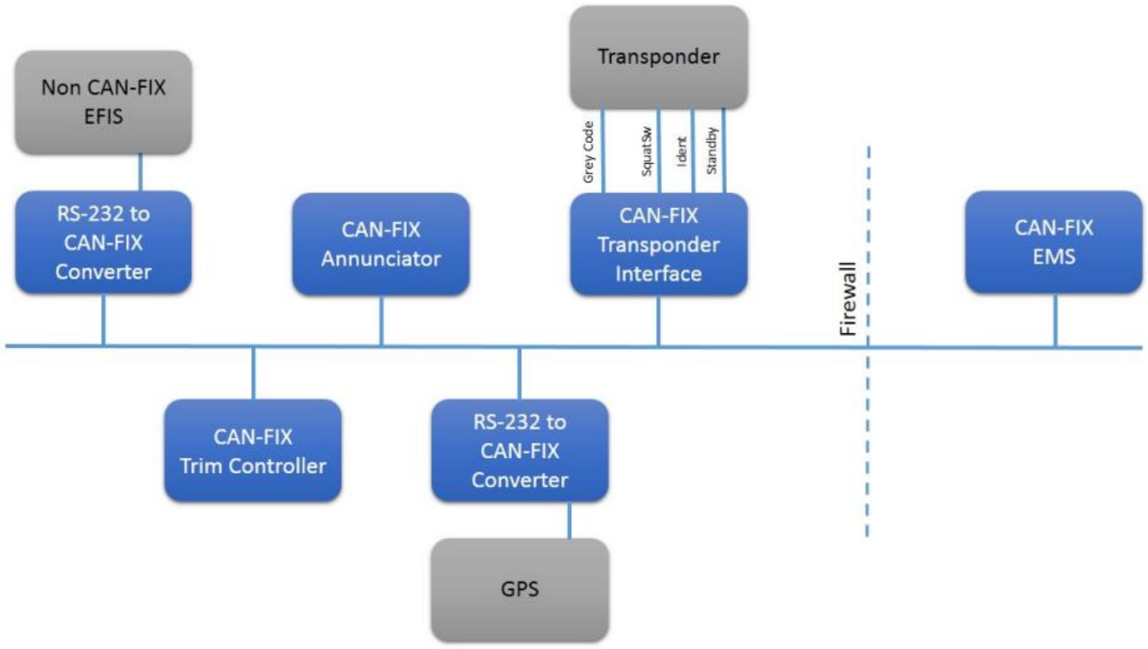

Figure 2. CAN-FiX Concept.

CAN-FiX Gateway

The FiX Gateway application is a free open source program that abstracts flight information from the CAN-FiX protocol and allows communication between different technologies that may wish to implement CAN-FiX without changing their underlying code. The primary use is as the interface to the pyEFIS and Engine Information System (EIS) projects. It can also be used to interface flight simulator software to 'real' hardware such as instrumentation or custom flight controls.

CAN-FiX Bus Connectors

The standard connector for CAN-BUS devices is a DB-9 and the network is terminated at each end with a 120 ohm resistor. The standard connector wiring is shown in the image below. Installers have the option of powering the module through the DB-9 connector or powering directly from the module to the avionics bus. Typical installations would only require two wires (CAN_H and CAN_L) from the module to the CAN-FiX bus plus power.

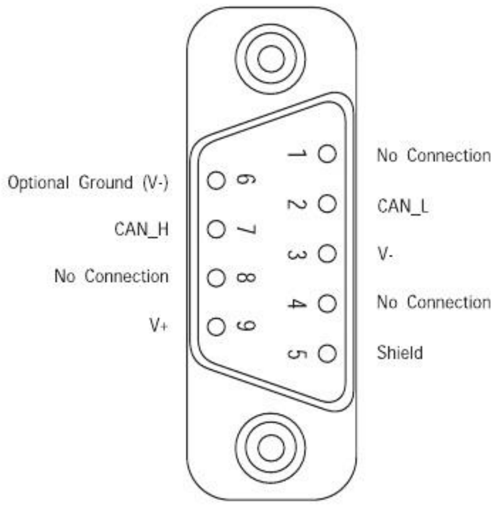

Figure 3. Standard DB-9 CAN connection pin-out.

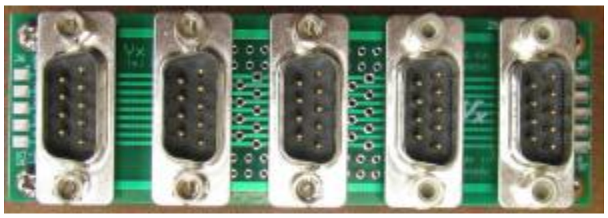

Figure 4. MakerPlane Axis-9 DB-9 CAN hub.

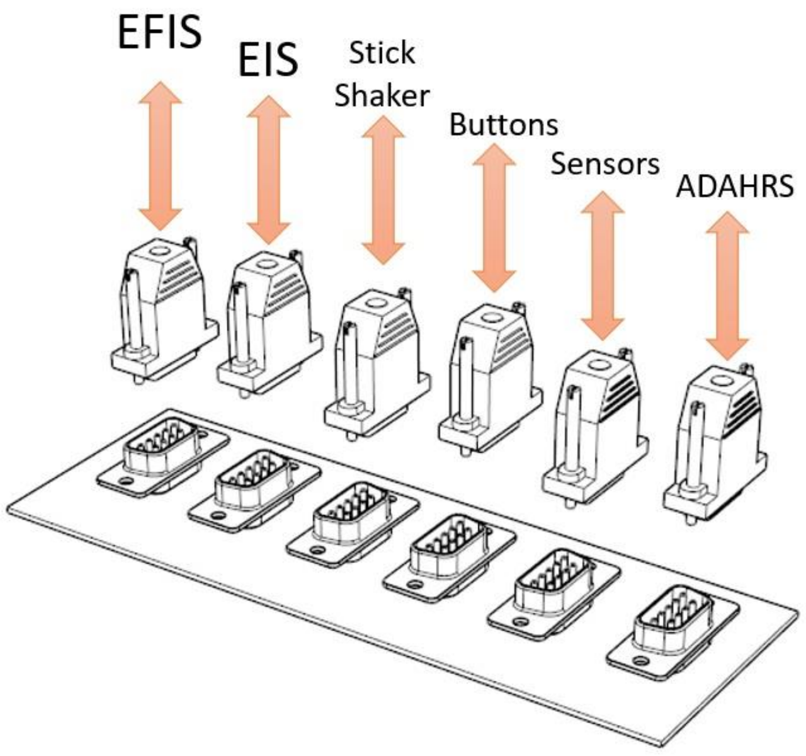

Figure 5. Using the Connector Bus.

FIX Gateway

FIX is an acronym for Flight Information eXchange. It is a set of protocol specifications for exchanging information between aircraft avionics and flight systems. This specification and the protocols themselves are licensed under a Creative Commons license that allows anyone to modify and redistribute these documents without charge.

This is a community supported endeavor, with the primary goal of providing a standard method for avionics and flight control systems to communicate with one other in a vendor neutral way. These specifications and protocols are primarily geared toward the Experimental Amateur Built (E-AB) aircraft community. Keeping the specification open and free allows airplane builders to create their own devices and write their own software that will be able to communicate with other devices without need to pay for specifications or licenses. It also encourages collaboration in the development and improvement of the protocols themselves.

FIX Gateway is a plug-in based program written in Python. There is a central database that is a repository for all of the information in the system. A plug-in would be loaded for each particular type of connection that is needed. Each of these plug-ins communicates with the central database. Each data item represents a distinct aircraft data point. Airspeed, altitude or oil pressure are simple examples.

The database is determined by a database configuration file that is read during start up. The user can modify this file if need be to customize the database for specific needs.
The plug-ins which would be loaded are determined by the main configuration file.
FIX is a protocol family. This manual describes the operation of FIX Gateway. A program designed to pass data between disparate technologies that may or may not communicate with any of the FIX protocols.
Examples of use are:

- Allow an Electronic Flight Information System (EFIS) communicate with a flight simulator.
- EFIS communication to CAN-FIX or other FIX devices within the airplane
- Conversion of data from one format to a standard FIX type.
- Hardware interface to flight simulators.
- Testing

The primary use is as the interface to the pyEfis electronic flight information project.

Installation

Begin by cloning the Git repository

```
git clone git@github.com:makerplane/FIX-Gateway.git fixgw
```

or

```
git clone https://github.com/makerplane/FIX-Gateway.git fixgw
```

Then run one of the two helper scripts.

```
./fixgw.py

./fixgwc.py
```

These will run the client and the server respectively.
The configuration files are in the `fixgw/config` directory.
If you'd like to install the program permanently to your system or into a virtualenv you can issue the command...

```
sudo pip3 install .
```

from the root directory of the source repository. **Caution** This feature is still in development and may not work consistently.
After installing with pip3 two helper scripts will be installed in the user's search path. These are `fixgw` and `fixgwc` for the server and client respectively. Once installed properly these commands can be used from anywhere.

Requirements

The only dependencies for FIX Gateway are Python itself and `pyyaml`. If you used pip3 to install FIX Gateway the dependencies should have been installed automatically. FIX Gateway requires Python 3.7 and should run on versions of Python higher than 3.7.

Many of the plugins will require other dependencies. See the individual plugin documentation for information about those. We'll discuss some of the more common ones.

If you intend to use the gui plugin you will also need PyQt5 installed. Consult the PyQt documentation on how to install PyQt on your system. FIX Gateway does not support PyQt4.
The canfix plugin will require both the python-can package as well as the python-canfix package. Installing the python-canfix package with pip3 should install both.

Possible Applications

The primary use case for FIX Gateway (FGW) is to integrate all the disparate flight information in the aircraft for display on an EFIS. PyEFIS is a Python based EFIS that is being developed in concert with FGW. This allows the EFIS designers to ignore all of the data interfaces that may be necessary and concentrate on development of the EFIS.

FGW supplies a common interface through a socket connection that abstracts the data so the EFIS does not care whether the data comes from a flight simulator, a real airplane or is just being managed manually by the programmer. Once the EFIS program is finished and installed in the airplane it need not be modified to get actual aircraft data. Only the FGW configuration would need to be changed.

Another use for FGW is to make it easy to integrate flight simulator data into other systems. The EFIS could be used as an instrument for the flight simulator. The entire aircraft panel could be duplicated in the EFIS/FGW and removed from the flight simulator itself. Interface to real equipment is simplified. Since FGW is written in Python it's quite easy to write a custom plug-in to read data from any source and get it into the database. Once in the database it's a simple matter of configuring other plug-ins to move that data to/from other sources.
For example, one plug-in could be written to read analog values from an Arduino and then that information could be used as pilot inputs to Microsoft Flight Simulator, Prepar3D, FlightGear, X-Plane or any other flight simulator that has an interface built for FGW.

It could also be used as a real hardware interface for an actual aircraft as well but good engineering practices should be used here for obvious reason. Flight controls should probably be done by other means in an actual aircraft.

It's conceivable that different flight simulators could be tied together using this technology as well. You could be flying a 172 in FlightGear and your neighbor could fly with you in his RV using X-Plane. The plug-ins for the different flight simulators are not quite this sophisticated yet but it's possible.

Client Utility

The client is useful for debugging and testing. The server should generally be run in the background as a service. The client connects to the server using the netfix protocol. The client allows us to remotely read and write database items as well as see database details and server status.

Usage

The Client can be run by executing the `fixgwc` command from the console. Or by executing the `fixgwc.py` python script from the distribution. If run with no arguments the client will try to connect to a server running on the same host with the default port number and start up in interactive mode.

The following command line arguments can be passed to the client to change the behavior.

```
--help, -h                 Show a help message and exit

--debug                    Run in debug mode

--host HOST, -H HOST       IP address or hostname of the FIX-Gateway Server

--port PORT, -P PORT       Port number to use for FIX-Gateway Server connection

--prompt PROMPT, -p PROMPT   Command line prompt

--file FILENAME, -f FILENAME Execute commands within file

--execute EXECUTE [EXECUTE ...], -x EXECUTE [EXECUTE ...] Execute command

--interactive, -i            Keep running after commands are executed
```

Commands

read <KEY>
This command will return the value in the database associated with the KEY.

write <KEY> <VALUE>
This command writes the value into the database entry associated with the key.

list
Lists all of the keys that are available in the database.

report <KEY>
Gives a list of all of the information that is associated with the database entry given by the key. This includes the datatype, the value, quality flags etc.

flag <KEY> <FLAG> <ARG>
Sets or clears a quality flag associated with the database entry given by key. The flag argument can be any one of b,f,a or s
These are for the *bad*, *fail*, *annunciate* and *secondary failure* flags respectively. ARG can be true or false and the flag will be set appropriately. 1 and 0 can also be used for ARG as a shorthand.

poll <KEY>
Subscribes to the given key and prints the value of the item in the database every time that item changes. Pressing a key stops the polling.

status
Prints the status to the screen

quit exit
Exits the Client

Basic Configuration

FIX Gateway is configured through a configuration file named `default.yaml. This file is`located in the fixgw/config/ subdirectory of the distribution, or installed into the proper place on the filesystem. The configuration uses YAML as it's configuration language. Upon startup the server searches a predefined set of directories, looking for the configuration file. If the configuration is not found the program will attempt to install default configuration files into the `.makerplane` directory in the users home folder. If that fails the server will report an error and fail to start.

When the server finds the configuration file it remembers where and writes that path to an internal variable that can be retrieved with `{CONFIG}` in other parts of the configuration. The first place the server will look for configuration is in the `~/.makerplane/fixgw/config` directory. This is where individual user configurations should be stored. For machine wide configurations the system directories such as `/etc/fixgw` or `/usr/local/etc/fixgw` can be used as well.

The **database file** option tells FGW where to find the database definition file. This file tells FGW how to build the internal database. For details on the format of the database definition file see the :doc:`database` section.

database file: "{CONFIG}/default.db"

The **initialization files** is a way to initialize the data in the database before the plugins are loaded. This will override the initial value defined in the database definition file but it's mostly used to set up things like the V speeds (Vfe, Vs, Vx, Vy, etc) or high and low alarm setpoints. It's preferable that the end devices send this data as part of their communication but not all end devices are designed to do this. The initialization file is a way to customize this data easily. Any plugin that writes to the database can override this data. Information in files listed later in the list will override earlier initializations. The database is initialized once on startup with this information. Any connections will be able to overwrite this data at runtime.
Initialization files:

```
  - "{CONFIG}/c170b.ini"

  - "{CONFIG}/fg_172.ini"
```

In both of the above declarations the string {CONFIG} is used. This will be replaced with the location where pip3 installed the configuration files. Relative paths can be used here as well and they will be relative to the current directory from where the server was run. Absolute paths to these files can also be given.

There is a list of connections in the configuration file that determine which connection plug-ins will be loaded. Each item in this connection list represents a specific connection plugin. Here is a short snippet of the connections list:

```
connections:

  # Network FIX Protocol Interface

  netfix:

    load: yes

    module: fixgw.plugins.netfix

    type: server

    host: 0.0.0.0

    port: 3490

    buffer_size: 1024

    timeout: 1.0
```

The above configuration tells FGW to load a connection plugin named *netfix* and use the python module found at fixgw.plugins.netfix. The load and module configuration options are the only two mandatory items. Any other options inside a connection object would be passed 'as is' to the plugin. The included configuration file contains examples of all the plugins that ship with the FIX Gateway distribution. Configuration of the individual plugins are documented elsewhere. The rest of the configuration file contains directives for message logging.

FGW uses the built in Python logging module. This is for message logging of the program itself. Not to be confused with logging flight data which is handled elsewhere. Python's logging system is very sophisticated and can log information in many different ways. It can log to the terminal, a file, the system logger, network sockets even email. A description of all that this system is capable of is beyond the scope of this documentation. See Python's logging module documentation for more details. So far we don't add any logging levels beyond what is included in the logger by default.

Running the server

To run the program simply type the following at the command line.

```
fixgw
```

There are a few command line arguments that can be used to adjust how the program runs. `--debug` is probably the most useful. This forces the logging module to set the loglevel to **debug**. If you are having trouble getting things to work the way you think they should using this argument can give you a lot of information to discover where the problem is. This option will produce a lot of data and probably shouldn't be used in the actual airplane.

Also if `--debug` is set there are some exceptions that will be raised in certain parts of the program that will stop the whole program. Without this flag they may simply cause a particular part of the program to stop functioning. With this flag it will raise the exception all the way to the top so that we can get the traceback information for troubleshooting. Again don't set this flag unless you are troubleshooting.

Other command line options are `--config-file` and `--log-config`. These set alternate files for the main configuration and logging configuration respectively. If the `--log-config` option is not set whatever file is used for the main configuration will be used for logging. The following command will load an alternate configuration file and turn debugging on:

```
fixgw --debug --config-file="test.yaml"
```

FGW will load the `test.yaml` file instead of the `default.yaml` configuration file that ships with the program.

Running the client

FIX Gateway ships with a small client program that allows the user to interact with the server through the netfix protocol. The netfix plugin must be loaded for this to work.

To run the client simply type the following at the command line.

```
fixgwc
```

The Main Database

The central feature of FIX-Gateway is the database. This is located in /fixgw/config/database.yaml. The database is essentially just a table of information stored in memory. All the data from all the plug-ins must pass through the database for the system to become functional. One plugin may be reading the Airspeed from a sensor and would write this data to the database. Another plugin may be communicating the airspeed to an EFIS or other type of indicator.

Each item in the database is identified by a key. The key is a short string that is unique for each entry in the database. An example would be IAS which is for indicated airspeed.
The value of the entry is not the only piece of information stored in the database. There are also descriptions of the entry, the range of the value, etc.

- **Value** - This is the actual value that the entry represents. An airspeed might be 123.4.
- **Units** - This would be the engineering units of the give database entry. For airspeed it is `knots`.
- **Description** - A detailed description of what the point represents. i.e. "Altimeter Setting" or "Exhaust Gas Temp Engine 1 Cylinder 3"
- **Min / Max** - The maximum range the of the value
- **Bad** - A flag indicating that the data might be in doubt. Some indication should be made that this data may be untrustworthy. `True` if data is bad and `False` otherwise.
- **Fail** - A flag indicating that the data is known to be bad. This flag is set when the plugin that is writing the data knows that the data is bad. Typically a zero value will be sent as well. This data should not be displayed at all and should not be used in any calculations.
- **Old** - A flag indicating that the database entry has not been updated within the specified period of time. Each database entry has a *time to live* and if this time is exceeded before another update to the database is made the *Old* flag will be set to `True`. For some data this is not relevant so the *time to live* is set to zero and the entry will never be marked as old.

- **Annunciate** - This is a flag that would tell indicating equipment that this point needs to be annunciated. This might mean that an oil pressure limit has been exceeded. It's more appropriate for the sending devices to decide when to annunciate information than the display equipment. There are a couple of reasons for this. The first is that the equipment that generates the data is better equiped to know when that data has exceeded limits. An example is oil temperature. There are typically low oil temperature limits, but they are useless right after engine startup when the oil temperature will be low anyway. No need to alarm on that. The other reason is, there may be more than one piece of display equipment and configuring each one with alarm limits is redundant.

- **Auxillary Data** - The Auxiliary Data (or Aux Data) is additional data that is associated with the point. It is mostly used for ranging instruments and indicating alarm and warning set points. It could be used for other things like 'V' speeds for Indicated airspeed as well. These aux data values are assumed to be of the same data type and should be within the same range as the item itself. They are simply stored in the database and delivered to the plugins that need them.

Database Definition

The database for FIX Gateway is defined by a YAML text file. The database is initialized based on the contents of this file before any of the plugins are loaded. The database structure is immutable. It cannot be changed by plugins once the program is loaded. Users can modify the database definition for their own use.

File Format

The following is an excerpt from the default database definition file.

```
variables:

  e: 2  # Engines

  c: 6  # Cylinders

  a: 8  # Generic Analogs

  b: 16 # Generic Buttons



entries:

- key: ANLGa

  description: Generic Analog %a

  type: float

  min: 0.0

  max: 1.0

  units: '%/100'

  initial: 0.0

  tol: 2000



- key: BTNb

  description: Generic Button %b

  type: bool

  tol: 0
```

```
- key: IAS

  description: Indicated Airspeed

  type: float

  min: 0.0

  max: 1000.0

  units: knots

  initial: 0.0

  tol: 2000

  aux: [Min,Max,V1,V2,Vne,Vfe,Vmc,Va,Vno,Vs,Vs0,Vx,Vy]


- key: ALT

  description: Indicated Altitude

  type: float

  min: -1000.0

  max: 60000.0

  units: ft

  initial: 0.0

  tol: 2000


- key: VS

  description: Vertical Speed

  type: float

  min: -30000.0

  max: 30000.0

  units: ft/min

  initial: 0.0
  tol: 2000

  aux: [Min,Max]


- key: OAT

  description: Outside Air Temperature

  type: float

  min: -100.0

  max: 100.0

  units: degC

  initial: 0.0

  tol: 2000

  aux: [Min,Max,lowWarn]


- key: ROLL

  description: Roll Angle

  type: float

  min: -180.0

  max: 180.0

  units: deg

  initial: 0.0

  tol: 200


- key: PITCH

  description: Pitch Angle

  type: float

  min: -90.0

  max: 90.0
  units: deg
  initial: 0.0
  tol: 200

- key: OILPe
  description: Oil Pressure Engine %e
  type: float
  min: 0.0
  max: 200.0
  units: psi
  initial: 0.0
  tol: 2000
  aux: [Min,Max,lowWarn,highWarn,lowAlarm,highAlarm]

- key: OILTe
  description: Oil Temperature Engine %e
  type: float
  min: 0.0
  max: 150.0
  units: degC
  initial: 0.0
  tol: 2000
  aux: [Min,Max,lowWarn,highWarn,lowAlarm,highAlarm]

- key: EGTec
  description: Exhaust Gas Temp Engine %e, Cylinder %c
  type: float
  min: 0.0

  max: 1000.0

  units: degC

  initial: 0.0

  tol: 2000

  aux: [Min,Max]


- key: CHTec

  description: Cylinder Head Temp Engine %e, Cylinder %c

  type: float

  min: 0.0

  max: 1000.0

  units: degC

  initial: 0.0

  tol: 2000

  aux: [Min,Max,lowWarn,highWarn,lowAlarm,highAlarm]
```

The YAML file defines two arrays or lists. The first is variables. Variables are a way to eliminate duplication in the database definition file. Comments can be included in the file as well. Following the variables list is the entries list. The entries list defines the individual entries that will make up the database.

Variables

Each variable when found in the definition will cause the initialization routine to duplicate and index that particular datapoint based on the number given. For example the variable e: 2 represents the number of engines that our aircraft will have. Instead of having to write each of the following three points twice (once for each engine) we just use the lower case letter 'e' in the Key definition and %e in the description.

```
- key: OILPe

  description: Oil Pressure Engine %e
  type: float

  min: 0.0

  max: 200.0

  units: psi

  initial: 0.0

  tol: 2000

  aux: [Min,Max,lowWarn,highWarn,lowAlarm,highAlarm]
```

This would cause OILP1 and OILP2 to be created in the database. This doesn't seem like much with just this example but consider the following:

```
- key: EGTec

  description: Exhaust Gas Temp Engine %e, Cylinder %c

  type: float

  min: 0.0

  max: 1000.0

  units: degC

  initial: 0.0

  tol: 2000

  aux: [Min,Max]
```

If e=2 and c=6 that single entry would produce 12 items in the database. It also makes it quite easy for a user to change the quantities of these things without having to search the entire database file to manage the data points. There are a number of items that use the 'e' for number of engine. Each of these would have to be managed individually if we had an entry for each point. As it is the user simply changes the line e: 1 to e: 2 if the aircraft only has two engines, and he'll get two Oil Pressures, Two Manifold Pressures, two Fuel Flows etc.

Database Item Definitions

Key

The Key is the unique identifier of the data point. The key should be in all capitol letters as any lower case letter will be considered to be a variable.

Description

The description is obvious. It is the human readable name of the item. '%x' can be used in the description to cause variable duplication and indexing. The text "Oil Pressure Engine #%e" would become "Oil Pressure Engine #1" during the first item's creation and "Oil Pressure Engine #2" during the second.

Type

There are four datatypes recognized by FIX Gateway, float, int, bool and str. float is the most common and simply represents a real number. int represents a whole number, or a number that has no decimal point. These are good for counters or numbers that would never contain a fractional part. bool is boolean value or a True/False value. Buttons and switches would be the most common booleans. str is a text string. This might be the aircrafts registration number or the time in string format.

Min and Max

These are the absolute limits by which the item's value will be constrained. Regardless of what value is written to the database the database item will never exceed these values. For example the magnetic heading (HEAD) has a Min of 0 and a Max of 360. If 370 is written to the point the actual value stored in the database would be 360.

Units

These are the engineering units applied to the item. "psi", "inHg", "feet", or "knots" are examples of units.

Initial

This is the initial value that the item will contain on start up. Most would be zero but occasionally it makes sense to initialize a datapoint to something else. For example the altimeter setting (BARO) is initialized to 29.92.

TOL

TOL stands for Time Out Lifetime in milliseconds. It's the amount of time that is given for each point to be written to the database. If a value is not written to the database in this amount of time the item is considered to be 'old' and the point will have the 'old' flag set to True when the value is read from the database. It is assumed that for the most part, the TOL is set to double the update rate. For some points a timeout does not make sense. If the TOL is set to zero the item will never be considered to be old.

Auxiliary Data

The Auxiliary Data (or Aux Data) is additional data that is associated with the point. It is mostly used for ranging instruments and indicating alarm and warning set points. It could be used for other things like 'V' speeds for Indicated airspeed as well. These aux data values are assumed to be of the same data type and should be within the same range as the item itself. They are simply stored in the database and delivered to the plugins that need them.

There are six fairly common aux data points, Min, Max, lowWarn, lowAlarm, highWarn and highAlarm. Min and Max here don't override the Min and Max above (probably should change the names to avoid confusion.) they would not affect the value that the database would store but are most often used to change the indicating range of the item. The other four might be used to indicate yellow arcs and/or red lines on gauges. In fact all of the gauges in pyEfis use these six values to determine the range of the gauge and the yellow and red arcs that are on the gauge. All a pyEfis gauge widget needs to know to do it's job is the key of the point you want to display. The aux data tells it everything else that it needs to know to do it's job.

The reason that the Auxiliary Data is stored in the FIX Gateway database instead of being handed off to the displaying device, is to make integration simpler. There may be several EFIS screens in the aircraft and each one would have to be configured with all of the low / warning setpoints for each point. Centralizing this information in the gateway makes it easier. It could also mean that a flap controller could have access to the Vfe data from the IAS point and then could protest in some way or indicate an alarm if the pilot tried to lower the flaps above this threshold. The flap controller would not have to be configured with this information it would simply be available and it would always match what is indicated on the Airspeed Indicator(s).

Net-FIX ASCII Protocol Description

The Net-FIX ASCII Protocol is a TCP/IP based FIX protocol that uses simple ASCII sentences.

Net-FIX ASCII is a sentence based ASCII protocol. There are two types of sentences. The first type is a simple data update. This is how the server would communicate the actual values of each data point. The second type of sentence is a command/response type of sentence.

Data Sentence Description

Data points are transmitted in colon ';' delimited strings that begin with the FIX identifier and are followed by the data.

The data sentence from the server to the client is formed like this:

```
ID;xxxx.x;aobfs\n
```

Where the ID is the Net-FIX ASCII identifier for the data point. For example TAS = True Airspeed. The Net-FIX ASCII identifier is the same identifier that is setup in the FIX-Gateway database. At some point these ID's may be formailzed separately but since we are still in development of this system at this stage we are using the FIX-Gateway :doc:`database` as a common place to configure this information.

The `xxxx.x` represents the value. The value can be a float, int, bool or string (The string cannot contain a ;). Floats will contain the decimal point, integers will not. Booleans will be 'T' or 'F' and strings will begin with an '&'. the 'aobfs' represent the quality flags. They will be either 1 or 0. a=annunciate, o=Old, b=bad, f=failed, s=secondary fail. The old flag is set if the data has not been written within the configured time to live for that point. The bad is set if there is reason to doubt the data but it hasn't actually failed. If the failed flag is set then the data cannot be trusted and should not be displayed or used in a calculation. The secondary failed flag may not always exist and if it is there it means that the secondary source of the data (for redundant systems) is failed and is not available. The sentence is terminated with a newline ('n' or 0x0A) character. The sentence from the client to the server is similar:

```
ID;xxxx.x;abfs\n
```

The difference is that the old flag is removed. If the client determines that the data is old it should simply set the bad flag. The secondary failed flag is optional. If flags are not sent they are assumed to be false '0'.

Command Sentence Description

Commands from the client to the server should begin with an '@'. What follows the '@' depend on the individual command.

Commands are single letter commands that are followed by any parameters that are needed. Responses to the command from the server will begin with the '@' and the command letter.

@cxxxxx...\n
c = the command letter xxxx... represents the data required by the individual command

Typically arguments to command are separated by semicolons ';'
@cxxxxx;arg1;arg2;arg3...\n

If the command expects a response then the response will follow with a message that starts with the @ symbol followed by the command letter then followed by the response. The individual command messages will document the actual syntax of a response.
If the command only expects an acknowledgement of success the server will simply respond with the message that was sent.

If there is a problem with the command the server would respond with the error symbol '!' followed by the error code. The following is an exmple of an error returned by the read command.

@rIAS!001 where 001 is the error code.
Some common error codes are given below:

- 001 - ID Not Found
- 002 - Bad Argument
- 003 - Bad Value
- 004 - Unknown Command

Read Command

`r` = Read Data - pass the ID or the ID + aux value that you want to read.
The response from the read command is the ID, folowed by the value, followed by the quality flags. The flags are exactly like the quality flags sent in a data update sentence

```
@rIAS;105.2;00000
```

`@rIAS.Vs` would cause the server to report the Vs auxilliary data if it exists.
Error Codes:

- 001 - ID Not Found

Write Command

`w` = Write the value. This command is similar to sending a normal data sentence except that it does not affect the quality flags and it gives the client a return value with errors if something fails.

```
@wIAS;105.2
```

Error Codes:

- 001 - ID Not Found
- 003- Bad Argument

Subscribe Command

`s` = subscribe - subscribe to an ID to have the server send this data each time it's written.

`@sTAS` would cause the server to send the True Airspeed each time it's written to the database. The server would respond with the identical message, or the ! followed by an error code.

Error Codes:

- 001 - ID Not Found

Unsubscribe Command

`u` = unsubscribe - unsubscribe from the data point.

`@uTAS` would undo the above subscription. The server would respond with the identical message, or the ! followed by an error code.

Error Codes:
- 001 - ID Not Found
- 002 - Duplicate Subscription

List Command

`l` = List - used to list the Identifiers that the server is handling.

`@l` would cause the server to send the entire list of IDs that are configured. The list may be huge and as such may be returned in more than one response. The client should be prepared for multiple responses. The response will include the total number of Identifiers to expect as well as the current index. The Identifiers will not be in any kind of order. Identifiers would be separated with commas ',' The response might look like this...

```
@l234;12;ID1,ID2,ID3,ID4...
```

Where 234 is the total and 12 is the starting index.

Query Command

`q` = Item Report - Used to cause the server to report all the data associated with a given database key. Data such as the min and max values the units the time to live etc.

`@qAOA` would cause the server to respond with all the parameters associated with this data point.

Server response.

```
@qAOA;desc;type;min;max;units;tol;aux
```

desc = the description of the data ("Indicated Airspeed") *type* = data type and will be one of [float, int, bool, str] *min* = the minimum value the point will ever be *max* = the maximum value the point will ever be *units* = string denoting the units ("knots") *tol* = an integer indicating the time to live of the point in milliseconds. *aux* = a comma separated list of the auxillary data points. ("min,max,lowWarn,lowAlarm")

Error Codes:

- 001 - ID Not Found

Flags Command

`f` = Set or Clear quality flags on a database item atomically

`@fID;flag;setting` where ID is the ID of the data point to modify. Flag is a single letter that represents the quality flag. It can be one of the following [aobfs]. Setting is either a '1' or a '0'.

On success the server will respond with the same command that it received.

```
``@fID;flag;bit``
```

Error Codes:

- 001 - ID Not Found
- 002 - Invalid Flag

- 003 - Invalid setting

Server Specific Command

`x` = Server Specific Command - This is used to send specific commands to a particular server.

`@x<cmd>` sends the <cmd> command to a server.

`@x<cmd>;<arguments>;...` sends the <cmd> command to a server with some number of arguments separated by ';'.

Server response.

```
@x<cmd>;<response>
```

Currently FIX-Gateway uses this command for retreiving the status. The command is:

`@xstatus` and the server will respond with a JSON string representing the status of the server.

The client/server is asynchronous so the client does not have to wait for a response from the server before sending another command. Data updates from subscriptions may also come in between the client command and the response. The client should pay attention to the structure of the message to make sure that it is a response to the command. This is why the arguments to the command are returned with the response. So the client can differentiate.

Min and Max that might show up in auxillary data is different than the min and max that show up as items in the report. The report items are the protocols limit on the data. If they show up in the aux data they are to be used for setting the range of indicators for display units. The datapoint will never exceed the min/max that are set in the database definition but the min and max that may be in the aux data are arbitrary and the server does nothing except type check that information.

Plugin Development

The FIX Gateway architecture involves multiple plugins reading and/or writing to a central database of flight information. Each plugin is a Python class that inherits from a base class that is supplied with the FIX Gateway program. The plugin files need only be in the Python search path and their associated information given in the main configuration file.

FIX Gateway will read the configuration file, determine which plugins to load, load them, start them and stop them when the time comes.

The best place to start with writing a plugin is with the `skel.py` file that is included in the distribution under the `plugins/` directory. This is a skeleton file that will give you a start on writing your own plugin.

```
.. literalinclude:: ../fixgw/plugins/skel.py
```

The basic idea is that you need to create a module that contains a class named `Plugin`. That class should extend the `plugin.PluginBase` class.

At a minimum implement two methods `run()` and `stop()`. The `run()` method will be called by FGW to start your plugin. Likewise, `stop()` will be called to tell your plugin to shutdown cleanly.

The __`init`__ method of the parent class will be passed an argument called *config*. This is stored in the created object as *self.config*. This is simply a dictionary of the configuration key/value pairs that were in your plugin's `conn_` section of the main configuration file, minus the few directives that FIX-Gateway needs to do it's work.

`run()` needs to return quickly. This is not the place for continuously running code. All the parts of your plugin that need to run continuously should be done in another thread and that thread started from `run()`. The rest of the system will wait for your `run()` method to exit so if it takes a long time it will slow the start of the rest of the system.

At anytime during the execution of your plugin it stops running, set the property `self.running = False`. This is a status point that is automatically generated for you plugin so the user can see if you are actually running or not. If the `run()` method raises an exception the flag will never be set to `True` by FGW.

The `stop()` method should also return fairly quickly. A common way of getting out of threads that are blocked on I/O is to set a timeout for the I/O and check a flag and exit if the flag is set. If the `stop()` method raises the exception `plugin.PluginFail` the FGW system will recognize that some threads may have not shutdown cleanly and will do a hard exit with an error code. This allows the program to go ahead and exit cleanly (without having to be killed) so that it can be restarted.

If you override the `__init__` method, your plugin should call the `__init__` method of the parent class.

The `get_status()` method can be overriden in your plugin to give information back to the status subsystem. The method should simply return a dictionary of key/value pairs of status information. The `skel.py` example simply returns the number of times we've been through the loop. Other plugins might send information about number of connections active, message counters, error counters etc. An `OrderedDict` is prefered as well since the user will get your information in the same order that you sent it. If this is not important then a normal python dictionary will work fine as well.

The dictionary that you return will be added to an internal dictionary along with all the ones returned by the other plugins. User interface plugins can use this information to display status to the user.

The following sections describe the current plug-ins available.

CAN-FIX Plugin

CAN-FIX is the Fix protocol implementation for CAN Bus.

Requirements

The `python-canfix` package is required and can be installed from PyPi with...
```
pip install python-canfix
```

This will also install the `python-can` package onto your system. `python-can` is used as the interface to the CAN Bus. There may be other requirements that need to be installed depending on which CAN Interface you want to use.

See https://github.com/hardbyte/python-can for more details.

Configuration

```
# CAN-FIX
canfix:
  load: yes
  module: plugins.canfix
  # See the python-can documentation for the meaning of these options
  interface: socketcan
  channel: vcan0
  #interface: serial
  #channel: /dev/ttyUSB0

  # This file controls the
  mapfile: config/canfix/default.map
  # The following is our Node Identification Information
  # See the CAN-FIX Protocol Specification for more information
  node: 145     # CAN-FIX Node ID
  device: 145   # CAN-FIX Device Type
  revision: 0   # Software Revision Number
  model: 0      # Model Number
```

Command Line Plugin

The command line plugin is useful for debugging and testing. It gives us the ability to view and manipulate the database from an interactive interface.

Configuration

```
command:

  load: yes

  module: plugins.command

  prompt: "FIX:"

  # If set quiting the command interpreter plugin

  # will end execution of the program

  quit: yes
```

Commands

read <KEY>
This command will return the value in the database associated with the KEY.

write <KEY> <VALUE>
This command writes the value into the database entry associated with the key.

list
Lists all of the keys that are available in the database.

report <KEY>
Gives a list of all of the information that is associated with the database entry given by the key. This includes the datatype, the value, quality flags etc.

flag <KEY> <FLAG> <ARG>
Sets or clears a quality flag associated with the database entry given by key. The flag argument can be any one of b,f,a or s These are for the *bad*, *fail*, *annunciate* and *secondary failure* flags respectively. ARG can be true or false and the flag will be set appropriately. 1 and 0 can also be used for ARG as a shorthand.

sub <KEY>
Subscribes to the given key. Once this command is run the value for the database item given by the key will be returned each time the database entry is changed.

unsub <KEY>

Unsubscribes from the given key. Stops the update of the given key.

status
Prints the status to the screen.

quit exit
Exits the Command Line Plugin. If quit = yes is set in the configuration file this will also cause the entire FIX-Gateway process to exit. Otherwise the command line plugin will stop and the rest of the system will function as it was.

Demo Plugin

The demo plugin is used to send simulated air and engine data through FIX Gateway to devices for testing if sensors are not available. Particularly useful to demonstrate pyEFIS functionality.

Configuration

```
demo:

  load: yes

  module: fixgw.plugins.demo
```

FlightGear Flight Simulator Plugin

FlightGear is the name of a very popular open source flight simulator. This plugin is used to allow FIX-Gateway to communicate with this simulator.

http://www.flightgear.org/

The plugin uses FlightGear's *Generic* protocol. This protocol is customizable by the user by way of an XML file. We've included an XML file for this purpose. The file included is called `fix_fgfs.xml` The fix_fgfs.xml file should be linked or copied to the FG_ROOT/Protocols directory which should be in one of the following locations:

- LINUX: `/usr/share/games/flightgear/Protocol/`
- OSX: `/Applications/FlightGear.app/Contents/Resources/data/`
- WINDOWS: `c:\Program Files\FlightGear\data\`

This is the location where FlightGear will search for the XML file.

To launch FlightGear using this protocol file use the following command:

```
>fgfs --generic=socket,out,10,localhost,5500,udp,fix_fgfs --
generic=socket,in,1,,5501,udp,fix_fgfs
```

The first --generic argument defines the output connection from FlightGear to FIX-Gateway. This corresponds to the <output> group in the XML file. The 10 is the update rate in updates / second, localhost and 5500 are the ip address and port that FlightGear will send to and FIX-Gateway will listen on. udp is the protocol to be used and this should not be changed. FIX-Gateway only usese udp at this point. The last argument in the list is the name of the XML file that would have the <output> group to tell both programs how the output sentence will be formed. This can change but it must match for both programs.

The second --generic argument contains similar data. The update rate has no meaning and if the host address is left off FlightGear will listen on all available interfaces.

This plugin also needs to know the location of this file so there is a directive in the configuration file for setting this location. It is very important that both FlightGear and this plugin are looking to the exact same file otherwise FIX-Gateway is going to be very confused.

It is also very important that the host and port information given in the above command line match what is in the main configuration file.

For more information on FlightGear's Generic Protocol see http://wiki.flightgear.org/Generic_protocol

Configuration

```
fgfs:

  load: yes

  module: fixgw.plugins.fgfs

  # This should be the same as $FG_ROOT on your system.  It is used to help

  # fixgw find the xml configuration files

  fg_root: /usr/share/games/flightgear/

  # fg_root: /Applications/FlightGear.app/Contents/Resources/data/

  # fg_root: c:\Program Files\FlightGear\data\


  # This is the name of the protocol config file that fixgw and fgfs will use to

  # define the protocol. It is very important that both programs are looking at

  # the same file.

  xml_file: fix_fgfs.xml

  # UDP Address and Ports used for communication to FlightGear.

  # Host address to listen on

  recv_host: localhost

  # port used for the -out argument to FlightGear

  recv_port: 5500

  # host address where FlightGear is running

  send_host: localhost

  # port used for the -in argument to FlightGear

  send_port: 5501

  # Update rate for sending data to FlightGear

  rate: 10  # updates / second
```

Both FIX-Gateway and FlightGear should be looking at the same XML file to determine how the data will be formatted and sent. There are two main sections in the FlightGear XML file. One is

<output> and the other <input>. The <output> section defines the data that is output from FlightGear and sent to FIX-Gateway. The <input> section is what FIX-Gateway sends to FlightGear. This seems backwards in FIX-Gateway but the XML file design is driven by FlightGear.

Within each <output> or <input> group each piece of data that we want to send or receive is contained in a <chunk>. Within each chunk are elements that define the protocol. The following is an example chunk:

```
<chunk>

     <name>OAT: Outside Air Temperature</name>

     <type>double</type>

     <format>%.1f</format>

     <node>/environment/temperature-degf</node>

     <offset>-32</offset>

     <factor>0.5555555555555555</factor>

</chunk>
```

FlightGear ignores the <name> element so we use it to define the database key within the FIX-Gateway database that we are wanting to read or write. The first characters up to the “:” are the key. Everything after the “:” is ignored and is there for clarity. The <format> element defines how the data will be represented in the sentence. These are C language style formatting directives. Both FIX-Gateway and FlightGear use these to encode and decode the data. FIX-Gateway expects a format that can be directly converted into the data type of the individual parameter.

The <node> element defines the property within FlightGear that we are going to read or write. You can see all the properties that are available with the Property Browser of FlightGear. (See the Browse Internal Properties menu item within the Debug menu)

The <offset> and <factor> elements define a way to do unit conversions in FlightGear. The <factor> element is a number that FlightGear will multiply the value by before sending it and <offset> is a number that will be added to the value. This example is because FlightGear sends this property in degrees Farenheit and FIX-Gateway expects the value in Degrees Centigrade. This is a bit of a contrived example because there is also a property that's in degrees C but we wanted something to show how this works.

Graphical User Interface Plugin

The GUI plugin is a way to graphically manage and view information in FIX-Gateway.

Requirements

The `PyQt5` package will be required for the gui module to work.

See the PyQt website for details.

Configuration

```
gui:
  load: yes
  module: plugins.gui
```

Net-FIX Protocol Plugin

Net-FIX is the TCP/IP implementation of the FIX protocol suite. There will be two versions of this protocol set ASCII and Binary. Currently this plugin only deals with the ASCII version of the protocol. Note that the binary version hasn't been invented yet.

Currently only the server side is implemented.

Net-FIX is the main way in which we connect FIX-Gateway to the pyEFIS program. Net-FIX/ASCII is currently the only communication mechanism that pyEFIS understands. In fact FIX-Gateway was written specifically to remove the complexities of multiple communications and data gathering mechanisms from pyEFIS.

Configuration

```
netfix:
  load: yes
  module: plugins.netfix
  type: server
  host: 0.0.0.0
  port: 3490
  buffer_size: 1024
  timeout: 1.0
```

RAIS (Redundant Array of Inexpensive Sensors) plugin

This FIX gateway plugin takes the output of a RAIS flight sensor pipeline and updates the database with the following keys:

- ROLL
- PITCH
- ALT
- IAS
- VS
- HEAD
- GS
- TRACK
- TRACKM
- YAW
- LAT
- LONG
- TIMEZ

The project with RAIS functionality is found at:

https://github.com/Maker42/openEFIS

Configuration

There are 3 configuration keys needed by the RAIS plugin:

1. rais_directory -- The directory where the openEFIS code is installed
2. rais_server_module = RAIS Don't change this unless you're sure you know what you're doing
3. rais_config_path (Optional) The path to the .yml file representing the pubsub configuration for RAIS.

rpi_bmp085 Plugin

This precision sensor from Bosch is the best low-cost sensing solution for measuring barometric pressure and temperature. Because pressure changes with altitude you can also use it as an altimeter! The sensor is soldered onto a PCB with a 3.3V regulator, I2C level shifter and pull-up resistors on the I2C pins.

- Pressure sensing range: 300-1100 hPa (9000m to -500m above sea level)
- Up to 0.03hPa / 0.25m resolution
- -40 to +85°C operational range, +-2°C temperature accuracy

Installation

The Adafruit_BMP library is required to use this plugin:

https://learn.adafruit.com/using-the-bmp085-with-raspberry-pi/using-the-adafruit-bmp085-python-library?view=all

```
sudo apt-get update

sudo apt-get install git build-essential python-dev python-smbus

git clone https://github.com/adafruit/Adafruit_Python_BMP.git

cd Adafruit_Python_BMP

sudo python setup.py install
```

Configuration

```
bmp085:
  load: yes
  module: plugins.rpi_bmp085
  tkey: CAT
  pkey: AIRPRESS

# Altitude result send directly to ALT KEY in feet
```

rpi_bno055 Plugin

If you've ever ordered and wire up a 9-DOF sensor, chances are you've also realized the challenge of turning the sensor data from an accelerometer, gyroscope and magnetometer into actual "3D space orientation"! Orientation is a hard problem to solve. The sensor fusion algorithms (the secret sauce that blends accelerometer, magnetometer and gyroscope data into stable three-axis orientation output) can be mind-numbingly difficult to get right and implement on low cost real time systems.

Bosch is the first company to get this right by taking a MEMS accelerometer, magnetometer and gyroscope and putting them on a single die with a high speed ARM Cortex-M0 based processor to digest all the sensor data, abstract the sensor fusion and real time requirements away, and spit out data you can use in quaternions, Euler angles or vectors.

Rather than spending weeks or months fiddling with algorithms of varying accuracy and complexity, you can have meaningful sensor data in minutes thanks to the BNO055 - a smart 9-DOF sensor that does the sensor fusion all on its own!

The BNO055 can output the following sensor data:

Absolute Orientation (Euler Vector, 100Hz)

- Three axis orientation data based on a 360° sphere

Absolute Orientation (Quaterion, 100Hz)

- Four point quaternion output for more accurate data manipulation

Angular Velocity Vector (100Hz)

- Three axis of 'rotation speed' in rad/s

Acceleration Vector (100Hz)

- Three axis of acceleration (gravity + linear motion) in m/s^2

Magnetic Field Strength Vector (20Hz)

- Three axis of magnetic field sensing in micro Tesla (uT)

Linear Acceleration Vector (100Hz)

- Three axis of linear acceleration data (acceleration minus gravity) in m/s^2

Gravity Vector (100Hz)

- Three axis of gravitational acceleration (minus any movement) in m/s^2

Temperature (1Hz)

- Ambient temperature in degrees celsius

This plugin automatically puts absolute orientation and acceleration data to database at +/-60Hz.

Configuration

```
bno055:

  load: yes

  module: plugins.rpi_bno055
```

Dependency Installation

https://learn.adafruit.com/bno055-absolute-orientation-sensor-with-raspberry-pi-and-beaglebone-black/hardware?view=all

Disable the kernel serial port

```
sudo apt-get update
sudo apt-get install -y build-essential python-dev python-smbus python-pip git
cd ~
git clone https://github.com/adafruit/Adafruit_Python_BNO055.git
cd Adafruit_Python_BNO055
sudo python setup.py install
```

rpi_button plugins

This plugins is a small python script to handle a momentary button. You can adjust the repeat delay or simply use the debouncing feature

Requirements

RPi.GPIO python package

Configuration

```
button1:
  load: yes
  module: plugins.rpi_button
  btnkey: BTN1
  btnpin: 4
  rdelay: 0 # 0 for debouncing or time in seconde to determine the repeat delay
```

rpi_mcp3008 Plugin

The MCP3008 is a low cost 8-channel 10-bit analog to digital converter. The precision of this ADC is similar to that of an Arduino Uno, and with 8 channels you can read quite a few analog signals from the Pi. This chip is a great option if you just need to read simple analog signals, like from a temperature or light sensor.

Requirement Installation

- https://learn.adafruit.com/raspberry-pi-analog-to-digital-converters/mcp3008

```
sudo apt-get update

sudo apt-get install build-essential python-dev python-smbus git

cd ~

git clone https://github.com/adafruit/Adafruit_Python_MCP3008.git

cd Adafruit_Python_MCP3008

sudo python setup.py install
```

Configuration

```
mcp3008:
  load: yes
  module: plugins.rpi_mcp3008
  vkey1: VOLT
  vkey2: ANLG2
  vkey3: ANLG3
  vkey4: ANLG4
  vkey5: ANLG5
  vkey6: ANLG6
  vkey7: ANLG7
  vkey8: ANLG8
  clk: 18
  miso: 23
  mosi: 24
  cs: 25
```

rpi_rotary_encoder Plugin

This plugins is a small python script to handle a rotary encoder with or without push button. You can use push button to handle 2 setting. You can adjust start setting point and the increment for bolt KEY

- TODO : Create a menu navigator like Garmin GNC or GNS series.

Requirements

Python's RPi.GPIO Library

Configuration

```
rotary_encoder:
  load: yes
  module: plugins.rpi_rotary_encoder
  btn: True
  btnkey: BARO
  btnstcounter: 29.92
  btnincr: 0.01
  btnpin: 4
  pina: 26
  pinb: 19
  stcount: 0
  rkey: PITCHSET
  incr: 1
```

rpi_virtualwire Plugin

Class to send and receive radio messages compatible with the Virtual Wire library for Arduinos. This library is commonly used with 313MHz and 434MHz radio tranceivers.

This plugins is a complex hardware and software configuration. See in Arduino folder to see the code.

- TODO : Create a <How to> to explain how to reproduce this little device

Requirement Installation

- http://abyz.co.uk/rpi/pigpio/

```
rm pigpio.zip

sudo rm -rf PIGPIO

wget abyz.co.uk/rpi/pigpio/pigpio.zip

unzip pigpio.zip

cd PIGPIO

make -j4

sudo make install
```

Configuration

```
virtualwire:
  load: yes
  module: plugins.rpi_virtualwire
  rxpin: 23
  bps: 2000
```

XPlane Flight Simulator Plugin

This plugin allows FIX-Gateway to be used to communicate to the very popular *X-Plane Flight Simulator.*

This plugin lacks a lot of work and is not really usable in it's current state

Configuration

```
xplane:
  load: yes
  module: plugins.xplane
  # IP address where the X-Plane simulator is running
  ipaddress: 127.0.0.1
  # UDP Ports to use for sending and receiving data
  # These should match the configuration in the
  # "Net Connections" Menu of X-Plane
  udp_in: 49001   # Port to received data from X-Plane
  udp_out: 49002  # Port to send data to X-Plane

  # These are the X-Plane data indexes that we will write.  These
  # would match the
  #idx8 : CTLPTCH, CTLROLL, CTLYAW, x, x, x, x, x
  idx25: THR1,  THR2,  x, x, x, x, x, x
  #idx28: PROP1, PROP2, x, x, x, x, x, x
  idx29: MIX1,  MIX2,  x, x, x, x, x, x
```

pyAvTools

pyAvTools is an open source aviation tool kit, providing useful modules for Python based Aviation apps, including:

- Data management package for charting objects
- Flight Information eXchange (FIX) database module
- ***A hard-button based user interface layer utilizing Qt (no keyboard or mouse required)***
    - Uses a rotary encoder + push buttons
    - NumberWidget (Show/Change a number)
    - ChoiceWidget (Show/Change a radio-button style section)
    - SelectMenuWidget (Show/Change a menu style section)
    - FIXDisplay (Show a collection of FIX database items)
- Others for future consideration

It can be downloaded here: https://github.com/makerplane/pyAvTools

Installation

Begin by cloning the Git repository:

```
git clone git@github.com:makerplane/pyAvTools.git
```

or

```
git clone https://github.com/makerplane/pyAvTools.git
```

If you'd like to install the program permanently to your system or into a virtualenv you can issue the command:

```
sudo pip3 install .
```

from the root directory of the source repository. **Caution** This feature is still in development and may not work consistently.

CAN-FiX-ArduinoLib

This is a library to interface Arduino devices to a CAN-FIX network.
It can be downloaded here:

https://github.com/makerplane/CAN-FIX-ArduinoLib

For information on installing libraries, see: http://arduino.cc/en/Guide/Libraries