Table of Contents

QGroundControl Dev Guide

This guide explains how *QGroundControl* (QGC) works internally, and provides guidelines for contributing code to the project. It is intended for use by developers!

> To learn how to **use** *QGroundControl*, see the User Guide.

> This guide is an active work in progress. The information provided should be correct, but you may find missing information or incomplete pages.

Support

This developer guide will eventually be the main provider of information about *QGroundControl* development. If you find that it is missing helpful information and/or has incorrect information please raise an issue.

Development questions can be raised in the QGroundControl Developer discuss category or in the *QGroundControl* Gitter channel.

Design Philosophy

QGC is designed to provide a single codebase that can run across multiple OS platforms as well as multiple device sizes and styles.

The QGC user interface is implemented using Qt QML. QML provides for hardware acceleration which is a key feature on lower powered devices such as tablets or phones. QML also provides features which allows us to more easily create a single user interface which can adapt itself to differing screen sizes and resolution.

The QGC UI targets itself more towards a tablet+touch style of UI than a desktop mouse-based UI. This make a single UI easier to create since tablet style UI also tends to work fine on desktop/laptops.

Communication Flow

Description of the high level communication flow which takes place during a vehicle auto-connect.

- LinkManager always has a UDP link open waiting for a Vehicle heartbeat
- LinkManager detects a new known device (Pixhawk, SiK Radio, PX4 Flow) connected to computer
    - Creates a new SerialLink connected to the device
- Bytes comes through Link and are sent to MAVLinkProtocol
- MAVLinkProtocol converts the bytes into a MAVLink message
- If the message is a `HEARTBEAT` the MultiVehicleManager is notified
- MultiVehicleManager is notifed of the `HEARTBEAT` and creates a new Vehicle object based on the information in the `HEARTBEAT` message
- The Vehicle instantiates the plugins which match the vehicle type
- The ParameterLoader associated with the vehicle sends a `PARAM_REQUEST_LIST` to the vehicle to load params using the parameter protocol
- Once parameter load is complete, the MissionManager associated with the Vehicle requests the mission items from the Vehicle using the mission item protocol
- Once parameter load is complete, the VehicleComponents display their UI in the Setup view

Firmware Plugins

Although the MAVLink spec defines a standard communication protocol to communicate with a vehicle. There are many aspects of that spec that at up for interpretation by the firmware developers. Because of this there are many cases where communication with a vehicle running one firmware must be slightly different to communication with a vehicle running a different firmware in order to accomplish the same task. Also each firmware may implement a subset of the MAVLink command set.

Another major issue is that the MAVLink spec does not cover vehicle configuration or a common parameter set. Due to this all code which relates to vehicle config ends up being firmware specific. Also any code which must refer to a specific parameter is also firmware specific.

Given all of these differences between firmware implementations it can be quite tricky to create a single ground station application that can support each without having the codebase degrade into a massive pile of if/then/else statements peppered everywhere based on the firmware the vehicle is using.

QGC uses a plugin architecture to isolate the firmware specific code from the code which is generic to all firmwares.

FirmwarePluginManager, FirmwarePlugin

Class Hierarchy (high level)

LinkManager, LinkInterface

A "Link" in QGC is a specific type of communication pipe with the vehicle such as a serial port or UDP over WiFi. The base class for all links is LinkInterface. Each link runs on it's own thread and sends bytes to MAVLinkProtocol.

The `LinkManager` object keeps track of all open links in the system. `LinkManager` also manages automatic connections through serial and UDP links.

MAVLinkProtocol

There is a single `MAVLinkProtocol` object in the system. It's job is to take incoming bytes from a link and translate them into MAVLink messages. MAVLink HEARTBEAT messages are routed to `MultiVehicleManager`. All MAVLink messages are routed to Vehicle's which are associated with the link.

MultiVehicleManager

There is a single `MultiVehicleManager` object within the system. When it receives a HEARTBEAT on a link which has not been previously seen it creates a Vehicle object. `` `MultiVehicleManager `` also keeps tracks of all Vehicles in the system and handles switching from one active vehicle to another and correctly handling a vehicle being removed.

Vehicle

The Vehicle object is the main interface through which the QGC code communicates with the physical vehicle.

Note: There is also a UAS object associated with each Vehicle which is a deprecated class and is slowly being phased out with all functionality moving to the Vehicle class. No new code should be added here.

FirmwarePlugin, FirmwarePluginManager

The FirmwarePlugin class is used an the base class for firmware plugins. A firmware plugin contains the firmware specific code, such that the Vehicle object is clean with respect to it supporting a single standard interface to the UI.

FirmwarePluginManager is a factory class which creates a FirmwarePlugin instance based on the MAV_AUTOPILOT/MAV_TYPE combination for the Vehicle.

> AutoPilotPlugin and AutoPilotPluginManager are deprecated class which also contains firmware specific code. All functionality in these are being moved to the newer FirmwarePlugin and FirmwarePluginManager implementations. No new code should be added here.

User Interface Design

The main pattern for UI design in QGC is a UI page written in QML, many times communicated with a custom "Controller" written in C++. This follows a somewhat hacked variant of the MVC design pattern.

The QML code binds to information associated with the system through the following mechanisms:

- The custom controller
- The global `QGroundControl` object which provides access to things like the active Vehicle
- The FactSystem which provides access to parameters and in some cases custom Facts.

Note: Due to the complexity of the QML used in QGC as well as it's reliance on communication with C++ objects to drive the ui it is not possible to use the QML Designer provided by Qt to edit QML.

Multi-Device Design Pattern

QGroundControl is designed to run on multiple device types from desktop to laptop to tablet to small phone sized screens using mouse and touch. Below is the description of how QGC does it and the reasoning behind it.

Efficient 1 person dev team

The design pattern that QGC development uses to solve this problem is based around making new feature development quick and allowing the code base to be testable and maintained by a very small team (let's say 1 developer as the default dev team size). The pattern to achieve this is followed very strictly, because not following it will lead to slower dev times and lower quality.

Supporting this 1 person dev team concept leads to some tough decisions which not everyone may be happy about. But it does lead to QGC being released on many OS and form factors using a single codebase. This is something most other ground stations out their are not capable of achieving.

What about contributors you ask? QGC has a decent amount of contributors. Can't they help move things past this 1 person dev team concept. Yes QGC has quite a few contributors. But unfortunately they come and go over time. And when they go, the code they contributed still must be maintained. Hence you fall back to the 1 person dev team concept which is mostly what has been around as an average over the last three years of development.

Target Device

The priority target for QGC UI design is a tablet, both from a touch standpoint and a screen size standpoint (think 10" Samsung Galaxy tab). Other device types and sizes may see some sacrifices of visuals and/or usability due to this decision. The current order when making priority based decisions is Tablet, Laptop, Desktop, Phone (any small screen).

Phone sized screen support

At specified above, at this point smaller phone sized screens are the lowest level priority for QGC. More focus is put onto making active flight level displays such as the Fly view more usable. Less focus is placed on Setup related views such as Setup and Plan. Those specific view are tested to be functionally usable on small screens but they may be painful to use.

Development tools used

Qt Layout controls

QGC does not have differently coded UIs which are targeted to different screen sizes and/or form factors. In general it uses QML Layout capabilities to reflow a single set of QML UI code to fit different form factors. In some cases it provides less detail on small screen sizes to make things fit. But that is a simple visibility pattern.

FactSystem

Internal to QGC is a system which is used to manage all of the individual pieces of data within the system. This data model is them connected to controls.

Heavy reliance on reusable controls

QGC UI is developed from a base set of reusable controls and UI elements. This way any new feature added to a reusable control is now available throughout the UI. These reusable controls also connect to FactSystem Facts which then automatically provides appropriate UI.

Cons for this design pattern

- The QGC user interface ends up being a hybrid style of desktop/laptop/tablet/phone. Hence not necessarily looking or feeling like it is optimized to any of these.
- Given the target device priority list and the fact that QGC tends to just re-layout the same UI elements to fit different form factors you will find this hybrid approach gets worse as you get farther away from the priority target. Hence small phone sized screens taking the worst hit on usability.
- The QGC reusable control set may not provide the absolute best UI in some cases. But it is still used to prevent the creation of additional maintenance surface area.
- Since the QGC UI uses the same UI code for all OSes, QGC does not follow the UI design guidelines specified by the OS itself. It has it's own visual style which is somewhat of a hybrid of things picked from each OS. Hence the UI looks and works mostly the same on all OS. Once again this means for example that QGC running on Android won't necessarily look like an android app. Or QGC running on an iPhone will not look or work like most other iPhone apps. That said the QGC visual/functional style should be understandable to these OS users.

Pros for this design pattern

- It takes less time to design a new feature since the UI coding is done once using this hybrid model and control set. Layout reflow is quite capable in Qt QML and becomes second nature once you get used to it.
- A piece of UI can be functionally tested on only platform since the functional code is the same across all form factors. Only layout flow must be visually checked on multiple devices but this is fairly easily done using the mobile simulators. In most cases this is what is needed:
  - Use desktop build, resizing windows to test reflow. Just will generally cover a tablet sized screen as well.
  - Use a mobile simulator to visually verify a phone sized screen. On OSX XCode iPhone simulator works really well.
- All of the above are critical to keep our hypothetical 1 person dev team efficient and quality high.

Future directions

- Raise phone (small screen) level prioritization to be more equal to Tablet. Current thinking is that this won't happen until a 3.3 release time frame (release after current one being worked on).

Font and Palette

QGC has a standard set of fonts and color palette which should be used by all user interface.

```
import QGroundControl.Palette         1.0
import QGroundControl.ScreenTools     1.0
```

QGCPalette

This item exposes the QGC color palette. There are two variants of this palette: light and dark. The light palette is meant for outdoor use and the dark palette is for indoor. Normally you should never specify a color directly for UI, you should always use one from the palette. If you don't follow this rule, the user interface you create will not be capable of changing from a light/dark style.

QGCMapPalette

The map palette is used for colors which are used to draw over a map. Due to the the different map styles, specifically satellite and street you need to have different colors to draw over them legibly. Satellite maps needs lighter colors to be seen whereas street maps need darker colors to be seen. The `QGCMapPalette` item provides a set of colors for this as well as the ability to switch between light and dark colors over maps.

ScreenTools

The ScreenTools item provides values which you can use to specify font sizing. It also provides information on screen size and whether QGC is running on a mobile device.

User Interface Controls

QGC provides a base set of controls for building user interface. In general they tend to be thin layers above the base QML Controls supported by Qt which respect the QGC color palette.

```
import QGroundControl.Controls 1.0
```

Qt Controls

The following controls are QGC variants of standard Qt QML Controls. They provide the same functionality as the corresponding Qt controls except for the fact that they are drawn using the QGC palette.

- QGCButton
- QGCCheckBox
- QGCColoredImage
- QGCComboBox
- QGCFlickable
- QGCLabel
- QGCMovableItem
- QGCRadioButton
- QGCSlider
- QGCTextField

QGC Controls

These custom controls are exclusive to QGC and are used to create standard UI elements.

- DropButton - RoundButton which drops out a panel of options when clicked. Example is Sync button in Plan view.
- ExclusiveGroupItem - Used a a base Item for custom controls which supports the QML ExclusiveGroup concept.
- QGCView - Base control for all top level views in the system. Provides support for FactPanels and displaying QGCViewDialogs and QGCViewMessages.
- QGCViewDialog - Dialog which pops out from the right side of a QGCView. You can specific the accept/reject buttons for the dialog as well as the dialog contents. Example usage is when you click on a parameter and it brings up the value editor dialog.
- QGCViewMessage - A simplified version of QGCViewDialog which allows you to specify buttons and a simple text message.
- QGCViewPanel - The main view contents inside of a QGCView.
- RoundButton - A round button control which uses an image as its inner contents.
- SetupPage - The base control for all Setup vehicle component pages. Provides a title, description and component page contents area.

Fact System

The Fact System provides a set of capabilities which standardizes and simplifies the creation of the QGC user interface.

Fact

A Fact represents a single value within the system.

FactMetaData

There is `FactMetaData` associated with each fact. It provides details on the Fact in order to drive automatic user interface generation and validation.

Fact Controls

A Fact Control is a QML user interface control which connects to a Fact and it's `FactMetaData` to provide a control to the user to modify/display the value associated with the Fact.

FactGroup

Top Level Views

This section contains topics about the code for the top level views: settings, setup, plan and fly.

Settings View

- Top level QML code is **AppSettings.qml**
- Each button loads a separate QML page

Setup View

- Top level QML code implemented in **SetupView.qml**
- Fixed set of buttons/pages: Summary, Firmware
- Remainder of buttons/pages come from AutoPilotPlugin VehicleComponent list

Plan View

- Top level QML code is in **MissionEditor.qml**
- Main visual UI is a FlightMap control
- QML communicates with MissionController (C++) which provides the view with the mission item data and methods

Fly View

- Top level QML code is in **FlightDisplayView.qml**
- QML code communicates with `MissionController` (C++) for mission display
- Instrument widgets communicate with active Vehicle object
- Two main inner views are:
  - `FlightDisplayViewMap`
  - `FlightDisplayViewVideo`

File Formats

This section contains topics about the file formats used/supported by *QGroundControl*.

Parameters File Format

```
# Onboard parameters for Vehicle 1
#
# # Vehicle-Id Component-Id Name Value Type
1    1    ACRO_LOCKING    0    2
1    1    ACRO_PITCH_RATE    180    4
1    1    ACRO_ROLL_RATE    180    4
1    1    ADSB_ENABLE    0    2
```

Above is an example of a parameter file with four parameters. The file can include as many parameters as needed.

Comments are preceded with a `#`.

This header: `# MAV ID COMPONENT ID PARAM NAME VALUE` describes the format for each row:

- `Vehicle-Id` Vehicle id
- `Component-Id` Component id for parameter
- `Name` Parameter name
- `Value` Parameter value
- `Type` Parameter type using MAVLink `MAV_PARAM_TYPE_*` enum values

A parameter file contains the parameters for a single Vehicle. It can contain parameters for multiple components within that Vehicle.

Plan File Format

> All values unless otherwise specified are in meters.

```
{
    "fileType": "Plan",
    "version": 1
    "groundStation": "QGroundControl",
    "mission": {
        "version": 2
        "firmwareType": 12,
        "vehicleType": 2,
        "cruiseSpeed": 15,
        "hoverSpeed": 5,
        "plannedHomePosition": [
            47.632939716176864,
            -122.08905141,
            40
        ],
        "items": [
             ...
        ],
    },
    "geoFence": {
        ...
    },
    "rallyPoints": {
        ...
    },
}
```

Above you can see the top level format of a Plan file. Plan files are stored in JSON file format and contain the mission, geo-fence and rally-points associated with the flight plan.

Key	Description
fileType	Must be "Plan".
version	The version for this file. Current version is 1.
groundStation	The name of the ground station which created this file.
mission	The mission associated with this flight plan.
geoFence	(Optional)
rallyPoints	(Optional)

Mission Object

The following values are required:

Key	Description
version	The version for the mission object. Current version is 2.
firmwareType	The firmware type that this mission was created for using the MAVLink MAV_AUTOPILOT enum values.
vehicleType	The vehicle type that this mission was created for using the MAVLink MAV_TYPE enum values.
cruiseSpeed	
hoverSpeed	
items	The list of mission item objects associated with the mission.
plannedHomePosition	The planned home position to show on the map when you are editing the mission. Values with array are latitude, longitude and altitude.

Mission Item - `SimpleItem`

A simple item represents a single MAVLink MISSION_ITEM command.

```
{
    "autoContinue": true,
    "command": 22,
    "frame": 2,
    "params": [
        0,
        0,
        0,
        0,
        47.633127690000002,
        -122.08867133,
        50
    ],
    "type": "SimpleItem"
},
```

The values in a `SimpleItem` map directly to the values in MISSION_ITEM:

Key	Description
autoContinue	MISSION_ITEM.autoContinue
command	MISSION_ITEM.command
frame	MISSION_ITEM.frame
params	MISSION_ITEM.param1,2,3,4,x,y,z

Mission Item - `ComplexItem`

A complex item is a higher level encapsulation of multiple MISSION_ITEMS treated as a single entity.

```
{
    "complexItemType": "survey",
    "type": "ComplexItem",
    "version": 3,
    ...
},
```

Complex items have two additional values associated with them:

Key	Description
complexItemType	Specifies the type of complex item. QGroundControl currently supports the following types: survey, fwLandingPattern
version	Specifies the version for this complex item.

Special handling for DO_JUMP mission item

Since DO_JUMP command requires you to specify the sequence number to jump to and the mission file format does not specify sequence numbers it require special handling.

First you must assign a unique identifier to the mission item you want to jump to:

```
{
    ...
    "doJumpId": 100
}
```

The `doJumpId` can be any value greater than 0 and must uniquely identify a DO_JUMP target.

Then in the actual DO_JUMP mission item you reference this unique id in the MISSION_ITEM.param1 value:

```
{
    ...
    "command": 177,
    "params": [
        100,
        ...
    ],
    ...
},
```

When the mission is loaded the actual DO_JUMP sequence number will be determined and filled in.

Complex Item - Survey

A survey creates a flight path over a polygonal area in a mission.

> All values unless otherwise specified are in meters.

```
{
   "camera": {
        "focalLength": 16,
        "groundResolution": 3,
        "imageFrontalOverlap": 10,
        "imageSideOverlap": 10,
        "name": "Sony ILCE-QX1",
        "orientationLandscape": true,
        "resolutionHeight": 3632,
        "resolutionWidth": 5456,
        "sensorHeight": 15.4,
        "sensorWidth": 23.199999999999999
    },
    "cameraTrigger": true,
    "cameraTriggerDistance": 25,
    "complexItemType": "survey",
    "fixedValueIsAltitude": false,
    "grid": {
        "altitude": 50,
        "angle": 0,
        "relativeAltitude": true,
        "spacing": 30,
        "turnAroundDistance": 0
    },
    "manualGrid": true,
    "polygon": [
        [
            47.633933816132817,
            -122.08937942845
        ],
        [
            47.634139864633021,
            -122.08781838280333
        ],
        [
            47.633395194285789,
            -122.08872496945037
        ]
    ],
    "type": "ComplexItem",
    "version": 3
}
```

A Survey is represented as a JSON object which is stored within the `items` array. It stores all the meta data associated with a Survey. It does not store the individual waypoints within the Survey. Those are generated when the mission is loaded by QGroundControl.

Key	Description
camera	Object which specifies the values associated with the camera being used for the survey. Only required if `manualGrid` is `false` .
cameraTrigger	Specifies whether the camera should be triggered at the specified `cameraTriggerDistance` intervals.
complexItemType	Identifies this complex mission item as a `survey` .
fixedValueIsAltitude	Specifies whether Altitude should be kept constant when modifying other values from the Survey user interface. Only used by the QGroundControl UI.
grid	Object which specifies the values associated with the survey grid.
manualGrid	`true` : Grid values were specified manually by the user, `false` : Grid values are based on camera settings specified by the `camera` object.
polygon	The polygon array which represents the polygonal survey area. Each point is a latitude, longitude pair for a polygon vertex.
type	Specifies that this item is a `ComplexItem` .
version	Version number for the Survey Complex Mission Item format. Current version is 3.

Grid Object

The `grid` object specifies the values associated with the survey grid.

```
"grid": {
    "altitude": 50,
    "angle": 0,
    "relativeAltitude": true,
    "spacing": 30,
    "turnAroundDistance": 0
},
```

Key	Description
altitude	The altitude for all transect waypoints within the grid.
angle	The angle in degrees for the transect paths.
relativeAltitude	`true` : `altitude` is relative to home, `false` : `altitude` is AMSL
spacing	The spacing in between each transect.
turnAroundDistance	The distance to fly past the polygon edge prior to turning for the next transect.

Camera Object

The `camera` object specifies the values associated with the camera being used for the survey. This object is only required if `manualGrid` is `false` .

```
"camera": {
    "focalLength": 16,
    "groundResolution": 3,
    "imageFrontalOverlap": 10,
    "imageSideOverlap": 10,
    "name": "Sony ILCE-QX1",
    "orientationLandscape": true,
    "resolutionHeight": 3632,
    "resolutionWidth": 5456,
    "sensorHeight": 15.4,
    "sensorWidth": 23.199999999999999
},
```

Key	Description
focalLength	Focal length of camera lens in millimeters.
groundResolution	Target ground resolution in cm/px.
imageFrontalOverlap	Percentage of frontal image overlap.
imageSideOverlap	Percentage of side image overlap.
name	Name of camera being used. Should correspond to one of the cameras known to QGroundControl. Use `"Custom Camera Grid"` for custom camera specifications,
orientationLandscape	`true` : Camera installed in landscape orientation on vehicle, `false` : Camera installed in portrait orientation on vehicle
resolutionHeight, resolutionWidth	Image pixel resolution.
sensorHeight, sensorWidth	Sensor dimensions in millimeters.

Rally Points File Format

```
{
    "fileType": "RallyPoints",
    "groundStation": "QGroundControl",
    "points": [
        [
            47.634309760000001,
            -122.08936869999999,
            50
        ],
        [
            47.634244700000004,
            -122.08700836,
            50
        ],
        [
            47.632784270000002,
            -122.08712101,
            50
        ],
        [
            47.632769809999999,
            -122.08939552,
            50
        ]
    ],
    "version": 1
}
```

GeoFence File Format

```
{
    "fileType": "GeoFence",
    "groundStation": "QGroundControl",
    "parameters": [
        {
            "compId": 1,
            "name": "FENCE_ENABLE",
            "value": 1
        },
        {
            "compId": 1,
            "name": "FENCE_TYPE",
            "value": 4
        },
        {
            "compId": 1,
            "name": "FENCE_ACTION",
            "value": 0
        },
        {
            "compId": 1,
            "name": "FENCE_ALT_MAX",
            "value": 0
        },
        {
            "compId": 1,
            "name": "FENCE_RADIUS",
            "value": 0
        },
        {
            "compId": 1,
            "name": "FENCE_MARGIN",
            "value": 0
        }
    ],
    "polygon": [
        [
            47.634457973002796,
            -122.08958864075316
        ],
        [
            47.634371216366716,
            -122.08675086361541
        ],
        [
            47.632610748511105,
            -122.08689033848418
        ],
        [
            47.632610748511105,
            -122.08967983585967
        ]
    ],
    "version": 1
}
```

Developer Tools

QGroundControl makes a number of tools available in debug builds. These ease common developer tasks including setting up simulated connections for testing, and accessing the System Shell over MAVLink.

> Build the source in debug mode to enable these tools.

Tools include:

- Mock Link - Creates and stops multiple simulated vehicle links.

Mock Link

Mock Link allows you to create and stop links to multiple simulated (mock) vehicles in *QGroundControl* debug builds.

The simulation does not support flight, but does allow easy testing of:

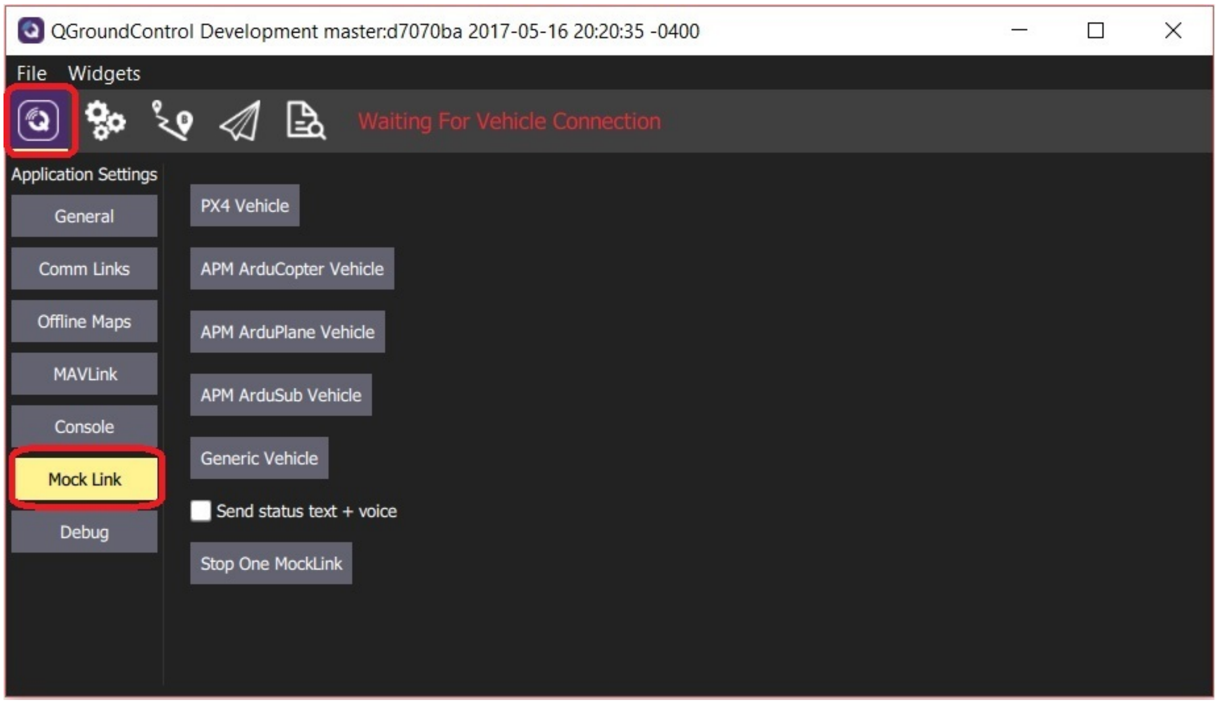
- Mission upload/download
- Viewing and changing parameters
- Testing most setup pages
- Multiple vehicle UIs

It is particularly useful for unit testing error cases for mission upload/download.

Using Mock Link

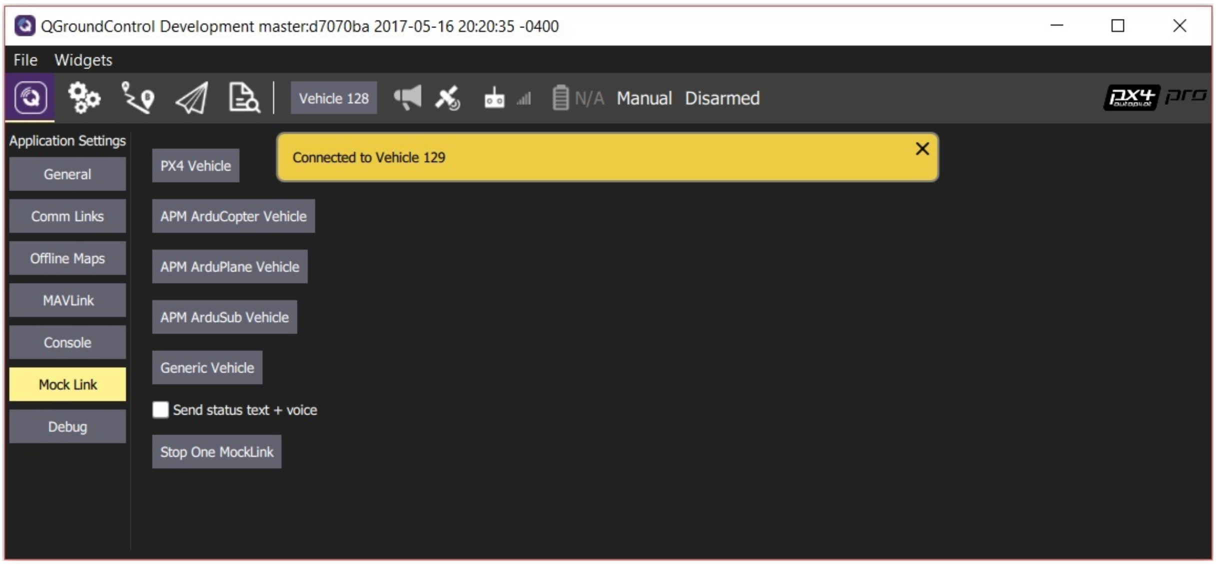
To use *Mock Link*:

1. Create a debug build by building the source.
2. Access *Mock Link* by selecting the *Application Settings* icon in the top toolbar and then **Mock Link** in the sidebar:



3. The buttons in the panel can be clicked to create a vehicle link of the associated type.

    - Each time you click a button a new connection will be created.
    - When there is more than one connection the multiple-vehicle UI will appear.



Connected to Vehicle 129

4. Click the **Stop one Mock Link** to stop the currently active vehicle.

Using *Mock Link* is then more or less the same as using any other vehicle, except that the simulation does not allow flight.

Code Submission

This section contains topics about the contributing code, including coding style, testing and the format of pull requests.

Coding Style

High level style information:

- Tabs expanded to 4 spaces
- Pascal/CamelCase naming conventions

The style itself is documents in the following example files:

- CodingStyle.cc
- CodingStyle.h
- CodingStyle.qml

Unit Tests

QGC contains a set of unit tests which must pass before a change will be considered through a pull request. The addition of new complex subsystems to QGC should include a corresponding new unit test to test it.

You can run unit tests from the command line using the `--unittest` command line option which will runs all tests. You can run a specific unit test by specifying it: `--unittest:RadioConfigTest`.

The full list of unit tests can be found in UnitTestList.cc.

Pull Requests

All pull requests go through the QGC CI build system which builds release and debug version. Builds will fail if there are compiler warnings. Also unit tests are run against supported OS debug builds.