OpenFlight Style Guide

# Overview

OpenFlight is a significantly complex software environment used in systems where loss of the flight vehicle, loss of mission, or loss of flight data would hinder research. OpenFlight has a large group of contributors and users at the University of Minnesota and externally. Although Simulink is a fairly intuitive coding environment, a style guide is necessary for ensuring consistency and readability across the code base. This style guide also encourages that best practices are used when developing software. This is a living document and will be updated as new best practices and style guidelines are discovered.

# Process

## GitHub

**Rule:** The branch and pull request workflow in GitHub (https://guides.github.com/introduction/flow/) shall be used for modifying and updating OpenFlight.

**Rationale:** The master branch is potentially used for active flight tests. Using a branch to commit code under development, until it has been tested and reviewed, ensures that development code does not affect research activities. The pull request process enables tracking changes, feedback, and open issues in the OpenFlight software while providing space to review changes before merging into the master branch. Simulink diagrams are source code, and as such, should be subject to the same "best practices" embraced by the software engineering community at large. Use of configuration management tools and techniques to manage diagram, library, input, and utility file change history should not be considered optional. The ability to manage change, back up from mistakes, and leverage tools that enable automatic assembly of a prescribed collections of files is key to highly efficient development and application of model-based controls. Using GitHub enables development by a large group of contributors.

**Rule:** Develop and commit one idea at a time.

**Rationale:** By keeping commits on branches small and focused on single ideas, clarity and the merging process are improved. Small encapsulated commits improve clarity through commit messaging and keep changes easy to see and understand. When a pull request is made to merge a branch back to the master, some changes may be accepted while others may require additional testing or improvements. By keeping commits focused on single ideas, the merging process is simplified by enabling this process of picking and choosing ideas to merge into the master.

**Rule:** Commit messages should explicitly explain in detail the changes or additions made.

**Rationale:** Commit messages are useful for guiding reviewers during the merging process and to help users understand changes made to the software. Useful commit messages help quickly identify why a change or improvement was made, what components of the software were modified, and how those modifications may affect the rest of the software. Commit messages become part of the documentation of that component of software.

**Rule:** All diagrams shall be tested (and pass) producing no errors or warnings prior to commitment.

**Rationale:** With a large user base and significant impact if software does not work reliably, it is crucial to ensure that code works without errors or warnings. Although warnings may seem like only an annoyance, they point to potential problems with the software and, if many warnings are present, can hide important messages from the user.

# File System and Data Organization

## File Naming

**Rule:** Simulink model files shall have the .mdl name extension

**Rationale:** This is the standard extension that MATLAB expects to find. Most modern operating systems will attempt to associate .mdl files with MATLAB/Simulink. A typical design model name might be vehicleResponse.mdl.

**Rule:** Simulink model names shall be descriptive camel case starting with a lower case letter, i.e. *flightDynamics.mdl.* If a particular vehicle already has named models (i.e. miniMUTT), be sure to follow the same naming convention for all related models.

**Rationale:** Consistent naming practices help improve code readability and allow users to quickly identify models within the OpenFlight file structure.

**Rule:** Simulink library filenames shall have a \_lib.mdl extension, i.e. *engineModel\_lib.mdl*

**Rationale:** Complex designs are difficult to manage as a single, large, flat .mdl file especially where multiple people collaborate on design. Simulink includes a facility for reusable library entries, which is encouraged. Libraries are not treated the same as flat .mdl files and it is important to clearly differentiate these file types.

## Subsystem Naming

**Rule:** Subsystem model names shall be a descriptive camel case starting with a lower case letter.

**Rationale:** Consistent naming practices help improve code readability and allow users to quickly identify subsystems within OpenFlight.

**Rule:** Port names shall be a descriptive camel case name starting with a lower case letter followed by the signal’s units, i.e. *alpha\_deg*

**Rationale:** This port naming convention; although resulting in longer names, provides quick identification of a signal and its units. This reduces the likelihood of errors being made with respect to using an unintended signal or mixing units.

## Signal Naming

**Rule:** Signals are named with: a camel case description of the signal beginning with a lower case letter, a camel case description of the signal’s origination, and the signal’s units. Every time a signal is modified, a new name and origination will be used. For example, angle of attack from sensor processing and measured in degrees would be *alpha\_senProc\_deg* whereas filtered angle of attack from the filtering and estimation block would be *alphaFilt\_filtEst\_deg*.

**Rationale:** This signal naming convention; although resulting in long signal names, provides quick identification of a signal, its origination, and its units. This reduces the likelihood of errors being made with respect to using an unintended signal or mixing units. Additionally, debugging is facilitated by providing a simple method to track signals to where they were last modified.

**Rule:** Buses shall have the \_bus extension in place of the units, i.e. *filtEst\_bus*.

**Rationale:** This bus naming convention helps identify buses from signals improving code readability.

# Signal Routing

**Rule:** Sample times for all blocks shall be set to sampleTime.

**Rationale:** Simulink is inconsistent in the default sample time assigned to blocks. Using inherited sample times can increase the likelihood of errors. Explicitly setting the sample time ensures that the block is setup to function as the designer intended. By using a global workspace variable for the sample time enables this to be easily modified if the base frequency of the flight software is modified.

**Rule:** Feedback paths will be drawn with lines

**Rationale:** Goto and from blocks are excellent at reducing clutter in Simulink diagrams; however, feedback paths are often critical to the functionality of a Simulink block and goto / from blocks mask this feedback functionality. By drawing all feedback paths with lines, clarity into the blocks function is enhanced.

**Rule:** Global goto / from blocks will not be used

**Rationale:** While local goto / from blocks are useful for reducing clutter, global blocks reduce clarity by masking a blocks true functionality. Additionally, global blocks increase the risk of accidentally overwriting common signal names.

**Rule:** Goto / from blocks shall be named the same as the signal or bus name they are associated with.

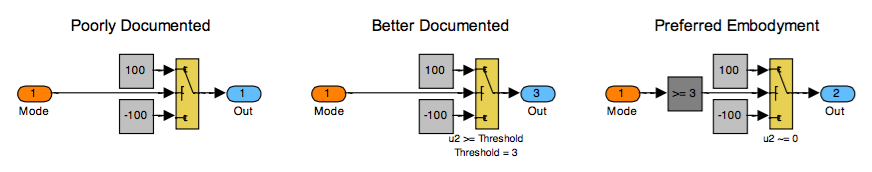
**Rationale:** What’s the point of using a great signal naming convention if you can’t keep track of signals once they reach a goto or from block? Worse, using generic names for these blocks leads to potentially overwriting signals.

**Rule:** Buses shall be used instead of mux blocks.

**Rationale:** Mux blocks reduce code clarity and require careful book keeping of signal order. In a large and complex software code base, maintaining this order is time consuming and error prone. Using buses enhances clarity by enabling selection of signals by the (descriptive) signal names, greatly reducing potential errors and enabling faster coding.

# Code Clarity

**Rule:** All necessary information to recreate a diagram must be available on a printed paper copy of the model.

**Rationale:** Access to the original model may not be available for code review or when attempting to reuse good ideas from past programs. A paper copy should be sufficient to reproduce the working code. See the example below: All three models function the same way, but only the center and right are fully documented. Note: If only Boolean switches are used throughout an entire model, the “u2 ~= 0” notation on the right most drawing may be removed.

**Rule:** Code one idea at a time. Make clear the purpose of the coded idea.

**Rationale:** While it is possible to encode very complex concepts within just a few blocks or even a single block using the under-hood settings, this practice should be discouraged if it hides the intent of the code.

**Rule:** Conditional and comparison block outputs shall be treated as true boolean values.

**Rationale:** The default Simulink data type is a double precision floating pointing value. Boolean results in diagrams (e.g. results from less-than or greater-than operators) may be improperly represented as floating point zeroes or ones, uint8 values, or other depending upon the block used. It is tempting, but incorrect, to use these results directly in arithmetic operations. (e.g. Multiplying a "boolean" 1 or 0 by a computed quantity to mask it from subsequent logic) Mixing data types in this manner is confusing to software engineers and makes porting resultant code to some processors difficult.

**Rule:** Switch blocks driven by logical expressions shall use the “not equal to Zero”, (boolean) input selection.

**Rationale:** Switch blocks should be driven by Boolean inputs. From the previous rule, we know that some versions of Simulink improperly treat Boolean results as double precision floats. To insure proper switch operation in code generation, the default switching logic should use the ~=0 choice. This makes the logic making the choice very clear to the human reader as well as generating very clean, fast acting code.

**Rule:** Block names should be hidden if the block's icon clearly describes its function

**Rationale:** Block names take up unnecessary diagram real-estate if the purpose of the block is clear from its icon. For example, the icon of a summing junction makes its function clear. Having a label on the block that says "summing junction" is redundant and wastes space.

**Rule:** Custom icons on blocks or subsystems should clearly indicate the block's function.

**Rationale:** Icons should be used to add clarity to the diagram. Icons placed on a block replace the default input/output port labeling. The designer must ask themselves which is the clearest way to describe the block's function and interface and then implement the clearest way possible. Remember, the diagram is there to help someone else, or you many years from now, figure out “what is the intent of this diagram.”

**Rule:** Block encapsulation masks should be used when encapsulation of the block will help make the behavioral of the system below clearly understood If the mask does not request input data for customization or multiple instance, double clicking should reveal the Simulink below.

**Rationale:** Electing to mask a block with dialog values and documentation makes it more cumbersome for a designer to examine the underlying diagram unless a forcing function is used on the open-block callback. It is recommended to create such masks only when understanding of the block's function can be enhanced by the encapsulation. Again see above and below.

**Rule:** Masked blocks with custom icons shall contain interface documentation in the block description field of the mask if the block does not behave as unmasked and have labeled ports.

**Rationale:** If not properly constructed to include labels on ports, masked blocks with custom icons by themselves provide no indication of their wiring interface. Without easily accessible documentation, the designer must look under the mask and inspect the diagram to determine the interface for wiring. This is not possible with a paper copy and thus cannot be recreated from a paper copy of the code without seeing subsequent layers which must clearly show the interface requirements.

**Rule:** Documentation of equations, with relevant references, should be made directly in the Simulink diagram above the relevant blocks.

**Rationale:** Documentation aids in debugging Simulink diagrams and equations should be documented along with reference material (i.e. book or paper and equation number). While DocBlocks are one method to document Simulink diagrams, their information is lost when printing the Simulink diagram. Following the concept that “all necessary information to recreate a diagram must be available on a printed paper copy of the model”, documentation should be completed directly in the Simulink model above the relevant blocks.

# Drawing Style Guidelines

**Rule:** In general, signal flow shall be from top down and left to right.

**Rationale:** This flow of information is comfortable to most Western readers, and admittedly represents an anglo-centric bias. However, one should not follow a slavish devotion to this diagram structure since feedback loops generally lend themselves to circular signal flow. The goal should always be to enable to diagram to clearly document its own function.

**Rule:** Wire crossings shall be minimized and unnecessary bends in wires shall be eliminated.

**Rationale:** Drawings are always more difficult to read when there are a multitude of wire crossings and convoluted wire paths. Designers should frequently challenge their peers‟ and their own drawing choices, such as input and output ordering, that might result in a cleaner wiring diagram. While the initially chosen I/O order may make since when a diagram is first created, it may be undesirable as a design grows and evolves. Make changes as necessary to keep the diagram neat. When the desired functionality is achieved, the computer is happy. However, the diagram is only done when it is made neat and understandable to the human reader/reviewer.

**Rule:** Wire crossings shall never connect. “T” intersections shall always connect.

**Rationale:** Although Simulink adds little dots to show wire connections. These often get misplaced when elements of a drawing are moved. Worse yet, if a drawing is several generations old in photo copy on anything but top line equipment, information can be lost on which wires cross and which connect. Make it clear, follow this rule as it will protect your information on even poor paper copies. It also makes your drawings much easier to read.

**Rule:** Spend time under the hood – open the dialog box for every Simulink element used. Choose options wisely. Document unusual choices with notes or call-backs.

**Rational:** Making the proper selections for data-type source, whether or not to saturate on overflow, how to round integer values, all play an important role in determining the quality, size and operation of the code generated. Become familiar with these operations, set the choices appropriately. Document unusual selections using the block properties annotations tab. Go as far as to create a library with properly set up, color coded, blocks. For example, color the block with the “standard” choices the default yellow, then use other shades of yellow for similar blocks with non-usual selections to help call attention. Place a key which explains the color coding and selections somewhere on the page.

**Rule:** A single, consistent, color scheme shall be used for all diagrams within a model.

**Rationale:** Inconsistent color usage decreases efficiency when navigating or studying diagrams. Consistent use of color makes instant visual identification of specific design elements possible. The following color assignments have been found to be non-distracting and enhance efficiency in navigating diagrams. They also produce varying shades of gray on black-and-white printers preserving much of the color information and allow easy readability of black on color or black on gray for all the below colors.

**Background Colors**

* White background for all diagrams
* Orange background for input nodes and “from” blocks
* Light Blue background for output nodes
* Yellow background for math blocks to make them stand out from the background
* Light Gray (from the custom palette) background for constants
* Gray background for logical operators and comparisons
* Magenta background for "from workspace" and any other prototype/target specific calibratable input blocks
* Blue or Green background for to-workspace - prototype/target specific test point output blocks
* Dark Green background for "goto" blocks
* Cyan background for grouped subsystems or library links to grouped subsystems
* White background for all elements which do nothing in code-generation but are required for simulation or to make Simulink happy. This makes them all but disappear in the diagram.
* Red background for attention needing critical operations like merges, or trigger blocks.
* Light Red (from the custom palette) background for items of important but lesser interest than above like function calls, data type conversions where attention is wanted, etc.

**Foreground Enhancements**

* Black foreground for all permanent elements.
* Red foreground for warning messages
* Gray foreground for optional, temporary, or experimental features (grayed out look)

**Library Backgrounds**

A color other than white shall be used for the background of a library. This allows user selection of color for information purposes but lets any user, even those who see the library for the first time, know they are in a library and not in a model.