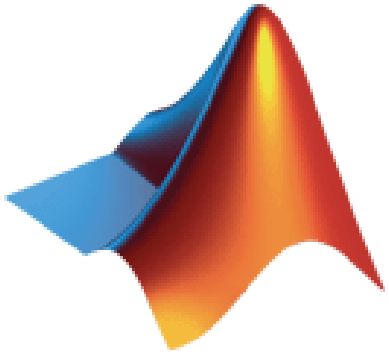
Pixhawk Pilot Support Package (PSP)
User Guide

Version 3.04

ISSUE DATE: June 2018



MathWorks

Pilot Engineering Group

1 Introduction 4
1.1 Basic Software Environment Description 4
1.2 Acronyms/Definitions 5
1.3 Contact Information 5
2 System Requirements 5
2.1 MATLAB/Simulink Toolboxes 5
2.2 Required (Windows) 5
2.3 Required (Linux) 6
3 Installation 6
3.1 Windows – Ubuntu Bash Setup Dependencies 6
3.1 Linux – Setting up Dependencies 6
3.1 Pilot Support Package setup (Windows & Linux) 7
4 Getting Started 9
4.1 Pixhawk Environment 9
4.2 Firmware Startup Preparation 11
4.3 Simulink Code Generation and Compilation 12
4.3.1 Simulink Settings 12
4.3.2 Target Hardware Resource Options 14
4.3.3 Building the Firmware 16
4.3.3.1 Build, Download and Run (Linux) 16
NOTE: please ensure putty or any connection to the Nutshell terminal is closed before attempting an upload! 16
4.3.3.2 Build Only and Manual Download (Windows) 17
NOTE: please ensure putty or any connection to the Nutshell terminal is closed before attempting an upload! 17
4.3.4 Starting the PX4 Simulink Application 18
4.3.5 Firmware and Code Generation structure 18
4.3.6 Hard Real-Time Constraints 20
4.4 Using QGroundControl with Pixhawk PSP for sensor calibration 21
4.5 Simulink Block Library 23
4.6 Example Models 24
4.6.1 px4demo_Parameter_CSC_example.slx 29
4.6.2 px4demo_ADC_example.slx 24
4.6.3 px4demo_input_rc.slx 25
4.6.4 px4demo_rgbled.slx 25
4.6.5 px4demo_tune.slx 26
4.6.6 px4demo_gps.slx 26
4.6.7 px4demo_attitude_plant.slx 27
4.6.8 px4demo_attitude_control.slx 27
4.6.9 px4demo_attitude_system.slx 28
4.6.10 px4demo_fcn_call_uorb_example.slx .....**Error! Bookmark not defined.**
4.6.11 px4demo_write_uorb_example.slx 30
4.6.12 Serial Communication 31
4.6.13 QGroundControl Demos – Parameter Tuning and Messages 32
5 Building your own custom Simulink Block 37
5.1.1 S-Function Approach 37

5.1.2 MATLAB Function blocks and System Objects 37
6 Limitations 38
6.1.1 Support for HIL / Mavlink 38
6.1.2 Supporting C++ uORB Message Data Structures 38
7 Updating to a new version of Pixhawk PSP 40

1 Introduction

The Pixhawk Pilot Support Package (PSP) feature allows users to use Simulink models to generate code targeted for platforms which run the PX4 flight stack. Originally this was targeted for the Pixhawk FMUv2 but has now been made expandable to other boards which run the PX4 software environment.

The PSP provides the ability to build and download to a PX4 board unit. It does not provide exact function behavior blocks for other services running on the Pixhawk (e.g. Attitude Estimation using EFK or SOF). The user will need to use blocks from the base Simulink or possibly the Aerospace blockset for simulating their flight control system model. Once the flight control system (FCS) has been successfully modeled, simulated and verified, the Pixhawk Target can be used to deploy the control system onto the PX4 hardware.

The Pixhawk Simulink blocks allows users to access sensor data and other calculations available to be used in their Simulink model at runtime. Generated code can then be compiled using the PX4 CMake build system.

For windows users:
This package formerly required the Windows PX4 Toolchain Installer v1.4 but since this is no longer maintained by the PX4 developers, this package now uses a different method for cross-compilation. Please see the instructions on more information.

1.1 Basic Software Environment Description

The Pixhawk Pilot Support Package is based off a forked version of the official Pixhawk Firmware. This forked version can be found here.
https://github.com/mathworks/PX4-Firmware/tree/PixhawkPSP_v3.0.3

During the PSP installation process, a download script will automatically clone this repository. This forked version is roughly based off of the 1.6.5 tag
https://github.com/PX4/Firmware/releases/tag/v1.6.5

A NuttX application called “px4_simulink_app” is created using this PSP and code generation tools. This application follows the same code structure and format depicted here.
http://dev.px4.io/tutorial-hello-sky.html

This Pilot Support Package has been tested with the Pixhawk (px4fmu-v2) and the Pixhawk Mini (px4fmu-v3) which can be configured to run different CMake configurations through the installation process. We have tested the “default” configuration but we also allow you to specify your own custom CMake configuration.

Please ensure that you select the correct CMake option which matches the board you are targeting:
https://dev.px4.io/en/setup/building_px4.html
For example:
Pixhawk 1: `make px4fmu-v2_default`
Pixhawk Mini: `make px4fmu-v3_default`

Since this package is generating code for a Simulink PX4 module, our PSP adapts the Simulink code generation and compilation process to fit into the Pixhawk build environment by making

use of CMake. A CMake command is executed compile the Pixhawk Firmware to invoke compilation.

Ideally, one should be familiar with the embedded software environment of the PX4 platform prior to using this Pilot Support Package. For more information on this, refer to the later sections that go into details about the code generation process as well as the PSP installation section.

1.2 Acronyms/Definitions

Pixhawk (PX4) – the Flight Controller Unit providing various sensor value inputs and PWM outputs as well as an ARM Cortex-M4 microprocessor for flight control and management.
PSP – Pilot Support Package. MathWorks software offering customized feature development or updates that are not yet available in the officially released version of MATLAB/Simulink.
TLC – Target Language Compiler
BTI – Built Tool Integration
FMU – Flight Management Unit
PWM – Pulse Width Modulation
RC – Radio Control
Tx/Rx – Transmitter/Receiver
ESC – Electronic Speed Controller
NED – North-East-Down

1.3 Contact Information

Please contact < > for questions on the PX4 Pilot Support Package

2 System Requirements

2.1 MATLAB/Simulink Toolboxes

To generate code from a Simulink model, the following products are needed:

- MATLAB R2017a / R2017b
    - Note: This Pilot Support Package has not been tested on R2018a
- Simulink
- Simulink Coder
- Embedded Coder
- Aerospace Blockset is needed for some of the example models
- Instrument Control Toolbox is needed for some data acquisition examples

2.2 Required (Windows)

To successfully work with the PX4 and deploy the generated firmware to the Pixhawk this additional software is needed. Windows 10 is required for the Pixhawk PSP and the Ubuntu bash terminal setup correctly. Please follow the instructions from Microsoft on how to configure this
https://docs.microsoft.com/en-us/windows/wsl/about

Once this has been done, the next step is to install the necessary PX4 dependencies (Cmake, python, cross-compilers) for the Windows-Ubuntu bash environment. A shell script has been provided to download and set this up.

2.3 Required (Linux)

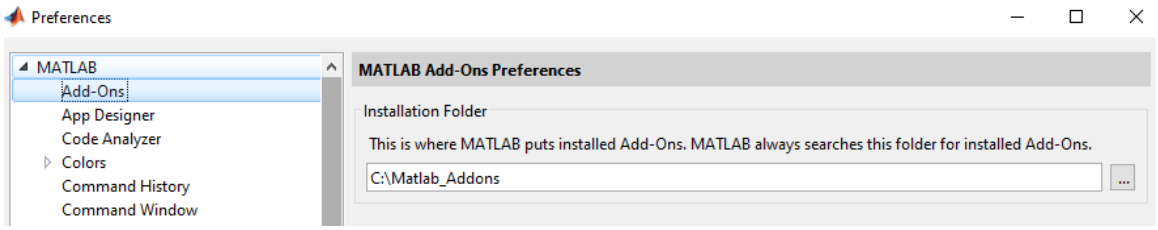
- arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors) 5.4
- Python
- CMake (tested with 3.5.1)

NOTE (Linux): Compilation of the firmware can actually fail on newer versions of the arm-none-eabi-gcc. All tests with the Pixhawk PSP were conducted on version arm-gcc 5.4

A shell script has been provided to download the necessary PX4 dependencies. More detail on this will be provided below.

3 Installation

This PSP is supported on Win64 and Linux platforms. By running the MLTBX installer it will copy/paste files to your MATLAB Add-ons folder which you configure under your MATLAB preferences.



In the above add-ons location, the PSP will be installed in C:\Matlab_Addons\PX4PSP\code\

3.1 Windows – Ubuntu Bash Setup Dependencies

NOTE: This section assumes you have NOT setup your Ubuntu bash for Windows with the necessary PX4 build environment dependencies

After you have setup and installed the Ubuntu-Linux bash shell, the next step is to setup this bash terminal with the correct cross-compiler and other dependencies. A script has been provided in
`<Location of installed PSP>\PX4PSP\code\px4\Win10bash_shell_setup\windows_bash_nuttx.sh`

This script was originally based off a script provided by the PX4 developers:
https://dev.px4.io/en/setup/dev_env_windows.html

The only major difference between the above script and what is provided is that we omitted the downloading of the PX4 Firmware. This is done at a different step.

3.1 Linux – Setting up Dependencies

NOTE: This section assumes you have NOT setup your Linux with the necessary PX4 build environment dependencies.

1) Install gcc-arm-none-eabi 5.4. There are numerous ways to do this - here is one approach which uses the exact same tool-chain used in the Windows 10 Bash

```
wget https://github.com/SolinGuo/arm-none-eabi-bash-on-win10-/raw/master/gcc-arm-none-eabi-5_4-2017q2-20170512-linux.tar.bz2
tar -jxf gcc-arm-none-eabi-5_4-2017q2-20170512-linux.tar.bz2
```

```
exportline="export PATH=$HOME/gcc-arm-none-eabi-5_4-2017q2/bin:\$PATH"
if grep -Fxq "$exportline" ~/.bashrc; then echo " GCC path already set." ; else echo
$exportline >> ~/.bashrc; fi
. ~/.bashrc
```

2) Run install shell script to setup build tools, CMAKE, python, other dependencies

The 'ubuntu_sim_common_deps.bash' script does this for you. This is originally based on the bash script provided here:
https://raw.githubusercontent.com/PX4/Devguide/master/build_scripts/ubuntu_sim_common_deps.sh

You can find this file in
`<Location of installed PSP>\PX4PSP\code\px4\Linux_setup\ubuntu_sim_common_deps.bash`

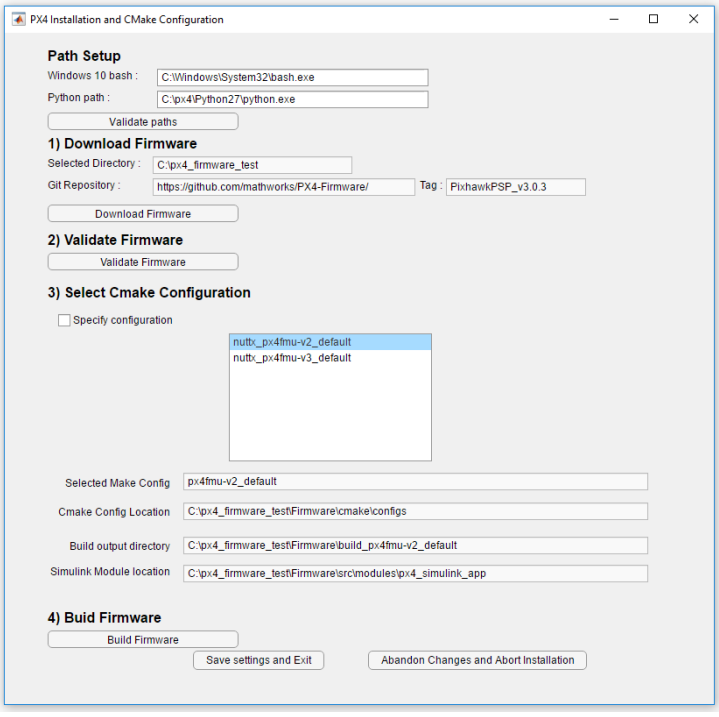
3.1 Pilot Support Package setup (Windows & Linux)

Next, you want to run the following command in MATLAB

```
PixhawkPSP('<Location of Firmware>')
```

Where `<Location of Firmare>` represents the folder path of where the PX4 firmware will be. Note that this folder must exist – you can select a folder with existing firmware so long as that firmware originated from
https://github.com/mathworks/PX4-Firmware/tree/PixhawkPSP_v3.0.3

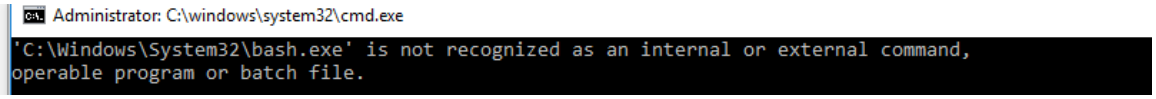
A user-interface menu will now appear



There are several steps to follow here:

Setup Path (Windows Only)
This will setup the location is only specific to Windows. It is used to configure the path to Python and the Ubuntu-Bash for Windows terminal.

NOTE: If you experience build errors in Simulink such as this:



If you get the above error you may need to use “Sysnative” rather than “System32” which can be specified in the “Windows 10 bash” text field.

Download Firmware

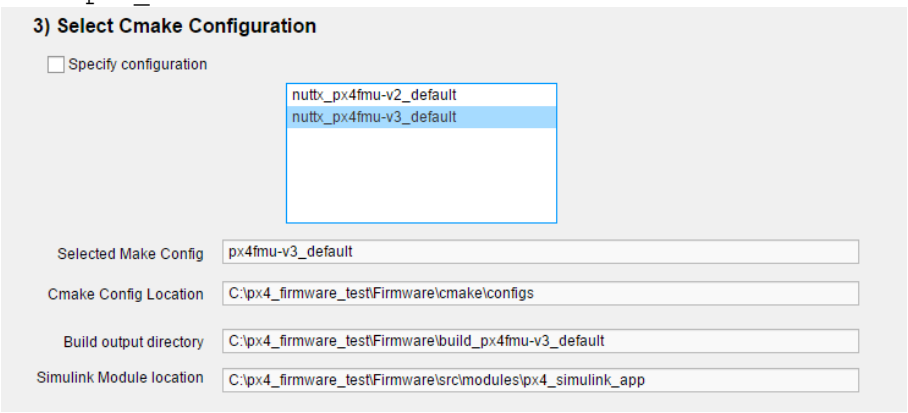
The PX4 firmware which is forked on the MathWorks GitHub will be cloned to the string argument you passed in to the `'PixhawkPSP'` function. The other option is to manually clone the firmware with git commands and then point to the parent folder where the firmware exists using the `PixhawkPSP('<folder location command>')`.

This command will open up a Windows 10 bash terminal and run a git clone firmware command. You can use the Validate Firmware button to confirm that the firmware which was cloned contains the PX4 Simulink module

Cmake Configuration

Next, select the CMAKE firmware. If you're targeting the Pixhawk Mini and the Pixhawk 2.1 Cube, we'll want to select v3. If you plan to target other PX4 platforms, you can click on the "custom" option and enter in the name of the make-file. Note that you will need to make the necessary modifications to the CMAKE / src files to add the PX4 Simulink App. To see which make file corresponds to the correct hardware platform, please see https://dev.px4.io/en/setup/building_px4.html

So for instance, if you wanted to support Pixhawk 3 Pro you would use the “specify” checkbox and then type
`“px4fmu-v4pro_default”`



Build Firmware

Next, build the firmware. This is a step will build most of the firmware such that when it comes to compiling the generated code the time taken on the first build will not build from the very beginning. Building the firmware will also create some necessary files for various header files needed by the build process.

On Windows this may take several minutes.

4 Getting Started

4.1 PX4 Environment

Using the default firmware available for the PX4, you should test to make sure your hardware is correctly configured and works as intended. This includes the correct motor wiring, placement of the PX4 module and any other sensors you may be using. You can use QGroundControl to download and flash the necessary firmware for this test.

You can launch a serial terminal program like TerraTerm or PuTTY and connect to the PX4 and manually run the built-in commands using the nuttx shell. NuttX is the OS that is delivered with the Pixhawk toolchain and will be used for running the code generated from your Simulink models.

You can find out which "Builtin" Apps your firmware has by typing "?" at the nuttx shell prompt "nsh>"

```
nsh> ?
help usage:  help [-v] [<cmd>]
    df kill mkrd rm unset ? echo losetup mh rmdir usleep
    cat exec ls mount set xd cd exit mb mv  sh cp free
    mkdir mw sleep cmp help mkfatfs ps test dd hexdump
    mkfifo pwd umount

Builtin Apps:
  sercon
  serdis
  adc
  attitude_estimator_ekf
  bl_update
  blinkm
  boardinfo
  commander
    …
```

Some useful commands are:

```
 1)  esc_calib – to calibrate your ESCs through the command line interface
    usage:
[-d <device>]      PWM output device (defaults to
                                                    /dev/pwm_output)
[-l <pwm>]         Low PWM value in us (default: 1000us)
[-h <pwm>]         High PWM value in us (default: 2000us)
[-c <channels>]    Supply channels (e.g. 1234)
[-m <chanmask>]    Directly supply channel mask (e.g. 0xF)
[-a]                       Use all outputs
```

2) pwm – to test out your PWM outputs

```
usage:
pwm arm|disarm|rate|failsafe|disarmed|min|max|test|info  ...

          arm                       Arm output
          disarm                    Disarm output

          rate ...                  Configure PWM rates
          [-g <channel group>]   Channel group that should
                                              update at the alternate rate
            [-m <chanmask> ]     Directly supply channel mask
            [-a]                 Configure all outputs
            -r <alt_rate>        PWM rate (50 to 400 Hz)

          failsafe ...           Configure failsafe PWM values
          disarmed ...           Configure disarmed PWM values
          min ...                Configure minimum PWM values
```

```
max ...                 Configure maximum PWM values
  [-c <channels>]       Supply channels (e.g. 1234)
  [-m <chanmask> ]      Directly supply channel mask
                                    (e.g. 0xF)
  [-a]                  Configure all outputs
  -p <pwm value>        PWM value

test ...                Directly set PWM values
  [-c <channels>]       Supply channels (e.g. 1234)
  [-m <chanmask> ]      Directly supply channel mask
                                    (e.g. 0xF)
  [-a]                  Configure all outputs
  -p <pwm value>        PWM value

info          Print information about the PWM device

  -v                    Print verbose information
  -d <device>           PWM output device (defaults to /dev/pwm_output)
```

3) tests – run various built-in test on the pixhawk hardware

```
nsh> tests help
Available tests:
  led
  int
  float
  sensors
  gpio
  hrt
  ppm
  servo
  ppm_loopback
  adc
  jig_voltages
  uart_loopback
  uart_baudchange
  uart_send
  uart_console
  hott_telemetry
  tone
  sleep
  time
  perf
  all
  jig
  param
  bson
  file
  file2
  mixer
  rc
  conv
  mount
  mtd
  mathlib
  help
```

4) top – will list all the NuttX processes running at the time (press ‘q’ to quit)
   Processes: 11 total, 2 running, 9 sleeping
   CPU usage: 37.36% tasks, 0.48% sched, 62.16% idle
   Uptime: 904.233s total, 561.039s idle

```
PID COMMAND                  CPU(ms) CPU(%)  USED/STACK PRIO(BASE) STATE
  0 Idle Task                 561039 62.162     0/    0   0 (  0)  READY
  1 hpwork                     26060  2.799   748/ 1992 192 (192)  w:sig
  2 lpwork                      6784  0.675   628/ 1992  50 ( 50)  READY
 85 top                          116  1.158  1244/ 1696 100 (100)  RUN
  7 nshterm                      121  0.000   884/ 1192 100 (100)  w:sem
  9 px4io                       9444  0.965   796/ 1992 240 (240)  w:sem
 26 sensors_task               38995  4.440  1364/ 1992 250 (250)  w:sem
```

```
  38 px4SimTermTask                  1  0.000   524/ 2040 100 (100)  w:sem
  29 attitude_estimator_ekf     191254 20.945 13004/13992 250 (250)  w:sem
  40 px4SimBaseTask              49428  5.501  1164/ 2552 100 (100)  w:sem
  42 px4SimSchedTask              7819  0.868  1028/ 2040 100 (100)  READY
```

5) SD card logging – useful for logging data to the SD Card

```
sdlog2: usage: sdlog2 {start|stop|status} [-r <log rate>] [-b <buffer size>] -e -
a -t -x
        -r      Log rate in Hz, 0 means unlimited rate
        -b      Log buffer size in KiB, default is 8
        -e      Enable logging by default (if not, can be started by command)
        -a      Log only when armed (can be still overriden by command)
        -t      Use date/time for naming log directories and files
        -x      Extended logging
```

4.2 Firmware Startup Preparation

Executing the default firmware, there are several processes that get executed at system startup. When deploying a custom flight control system you will need to suppress the execution of these processes and instead, run the application generated by Simulink. This is done by a start-up script put on the micro-SD card used on the PX4. In this way you can control which flight control software you want to run just by changing the contents of this script.

The script's filename is **rc.txt**. It should be copied to the SD-card directory **/etc**. A script sample has been provided by the PSP installation and can be found in your Pixhawk Toolchain installation directory: **<Selected Firmware Location>\example_rctxt\rc.txt**. By copying this file to your SD-card in the folder **/etc** the Pixhawk will execute the ***px4_simulink_app*** at system startup. This app is built into the firmware that is flashed onto your PX4 hardware at Simulink Model build time. Simply renaming this file (e.g. rc.txt to rc.txt.simulink) on the SD-card will allow boot-up of the default flight control software.

NOTE: Newer release of the Pixhawk firmware has changed how the boot-up tone is played by moving it to the 'commander' application. Because we are not using the commander application and instead running our own boot sequence from the SD card, you will not hear the boot-up sound. You can manually add a tone alarm sequence in the rc.txt file to indicate successful boot-up.

More information can be found on the Pixhawk website
https://dev.px4.io/en/advanced/system_startup.html

The Pixhawk firmware relies on a publisher-subscriber communication architecture for Inter-Process communication on the PX4. This mechanism is implemented by the uORB or micro-Object-Request-Broker application. It provides the infrastructure that allows threads and applications to share data between each other. Data is exchanged between participants in what is known as "topics". Any task can register themselves as a publisher or subscriber of a particular topic. Topic information is exchanged in defined common "C" structures.

More information on uORB can be found here: https://dev.px4.io/en/middleware/uorb.html

In order for the firmware to properly function, the uorb task must be executed upon startup (uorb start). Many of the Simulink Blocks that generate code interacting with the PX4 hardware rely on the uORB mechanism.

The next section will talk more about how the Pixhawk Pilot Support Package creates the ***px4_simulink_app*** application from generated code.

4.3 Simulink Code Generation and Compilation

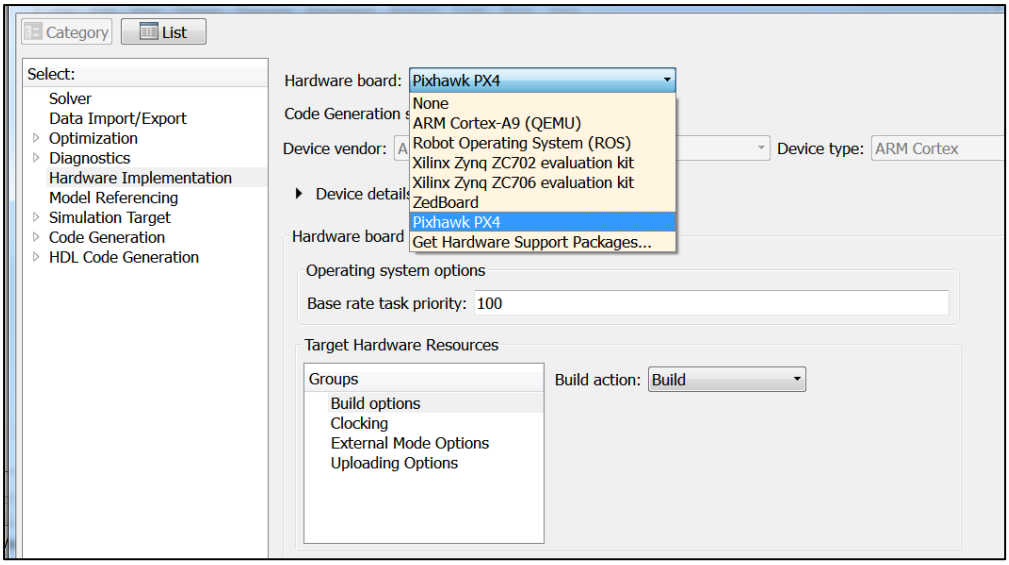
The Pixhawk target uses MathWorks Build Tool Integration (BTI) to allow MATLAB to invoke the ARM-GCC compiler to build ***px4_simulink_app***. The system target file needs to be ert.tlc (Embedded Real-Time) which is available with Embedded Coder. The user is then able to choose the hardware and toolchain. If the target hardware is set to 'Pixhawk', then the appropriate toolchain (Pixhawk) will be chosen automatically.

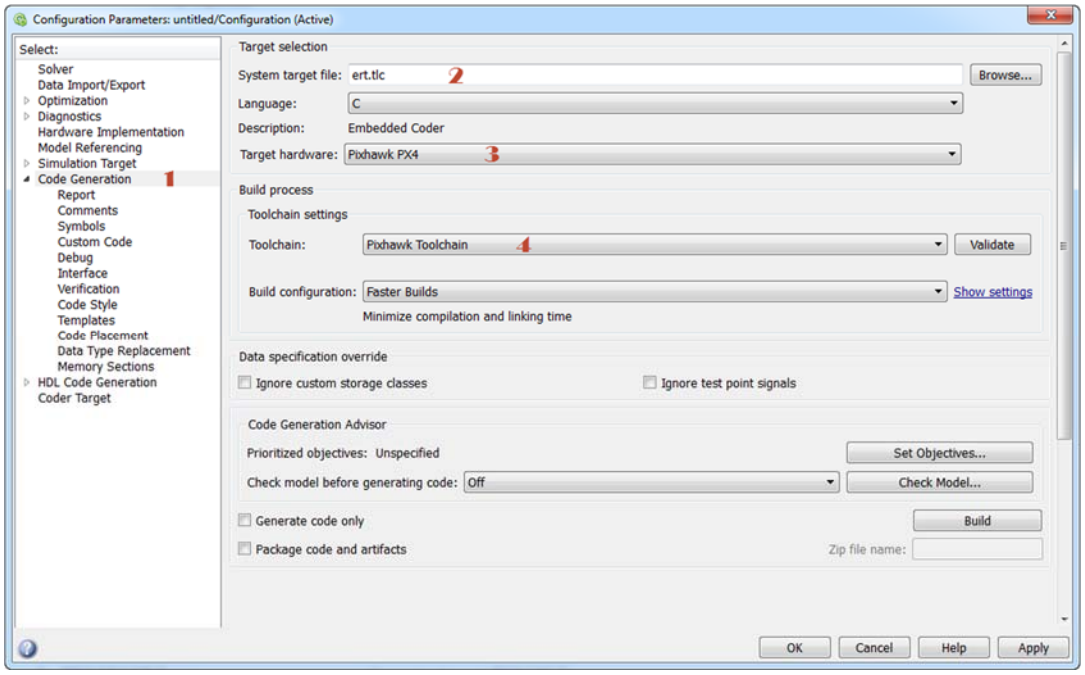
The Pixhawk firmware now uses a CMake build process. We have adapted the PSP to take advantage of this. This is separated into different parts:

1) Code generation of Simulink Model
2) Transfer generated code to \px4\Firmware\src\modules\px4_simulink_app along with a *CMakelist.txt* which describes the necessary source files, include paths and compiler options inherited from Build Tool Integration.
3) Invoke CMake commands to build the entire firmware. Since we already built most of the Firmware this process should advance more quickly than building it for the first time. The only difference being that CMake will now integrate our newly added Simulink generated code for ***px4_simulink_app.*** If you are on a Windows 10 machine this part is done within a Ubuntu bash terminal
4) The firmware image (*.px4 file) will be compiled and placed here \px4\Firmware\build_<firmware type>\
5) If the download option was also selected, user will be prompted to plug in the Pixhawk FMU to upload the firmware. Note that this is only available in Linux. In Windows, you will need to invoke the Firmware download manually.

4.3.1 Simulink Settings

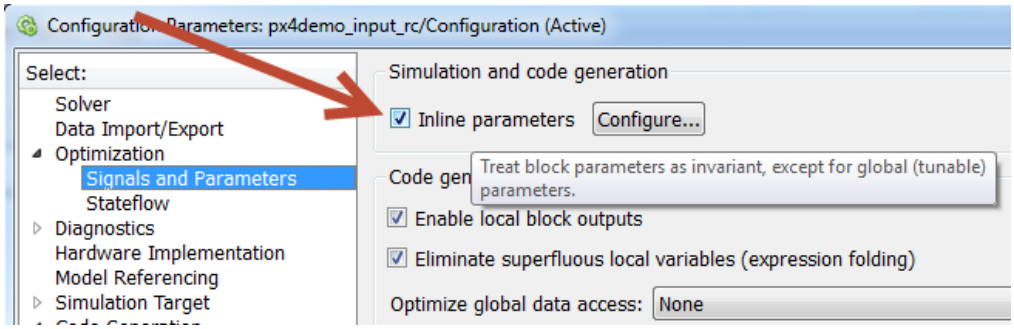
For the model to target the PX4 hardware, the Simulink model must be configured to use the appropriate code generation options. Go to the Hardware Implementation page and select Pixhawk PX4 to do this. Note that in previous releases this selection was done in the Code Generation panel, but nowit has moved into Hardware Implementation. The code generation panel should automatically update the labeled items one through four (1-4) to select the correct compiler and build configurations.





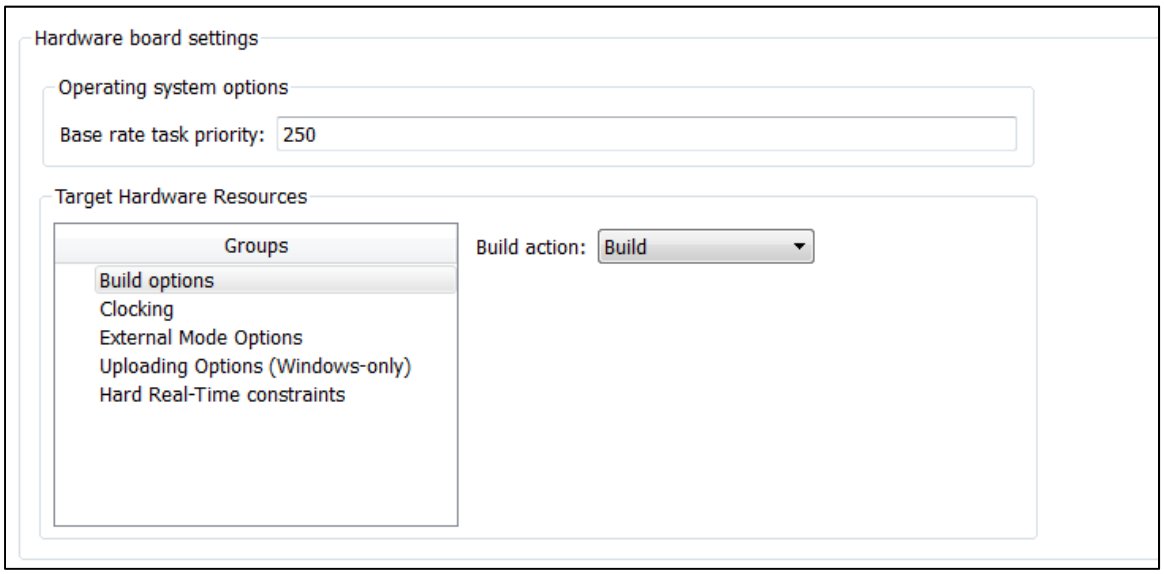
There are a few other settings which are required for this version of the PSP. These are:

1) Solver Type should be set for Fixed-Step (for embedded code generation)
2) Model Optimization Option **Inline Params** must be '**on**' for Pixhawk code generation



Inline parameters setting is highly recommended due to the limited resources in global memory and constraints on the Pixhawk target. Inline parameters places all model parameters (ie: gains) as “inline” constants or variables on the function stack rather. You will receive an error if this setting is not adjusted in your model.

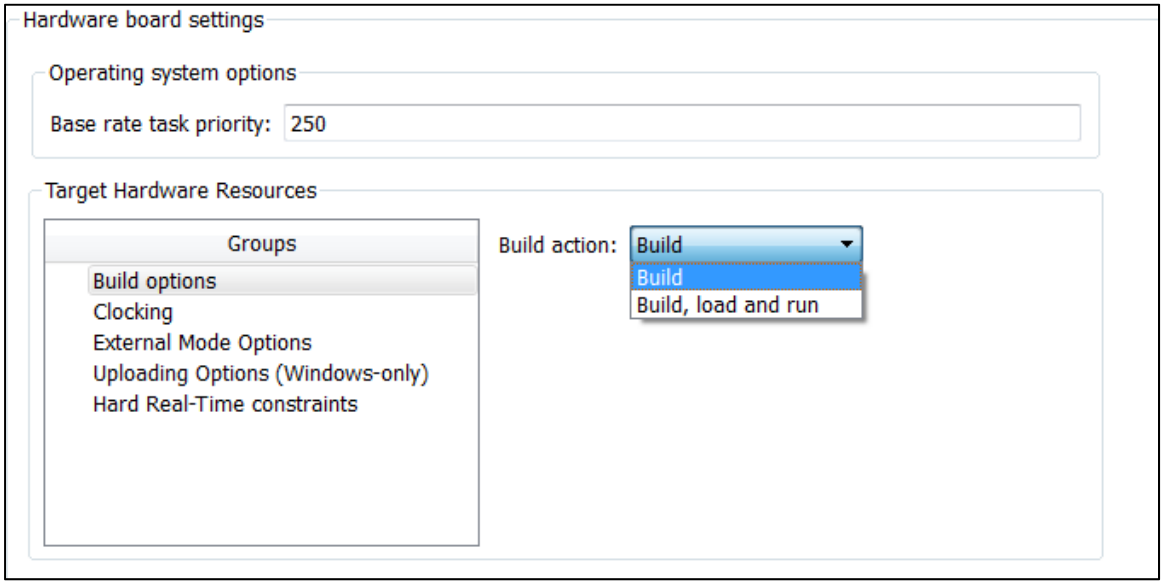
4.3.2 Target Hardware Resource Options



Under the Hardware Implementation pane there are several Target Hardware Resource Options. These are explained in detail below.

1) Base rate task priority:
   - When the generated code begins executing, several threads are spawned, one being the base-rate thread which runs at the model’s base sample rate. The priority of this thread can be adjusted if needed.

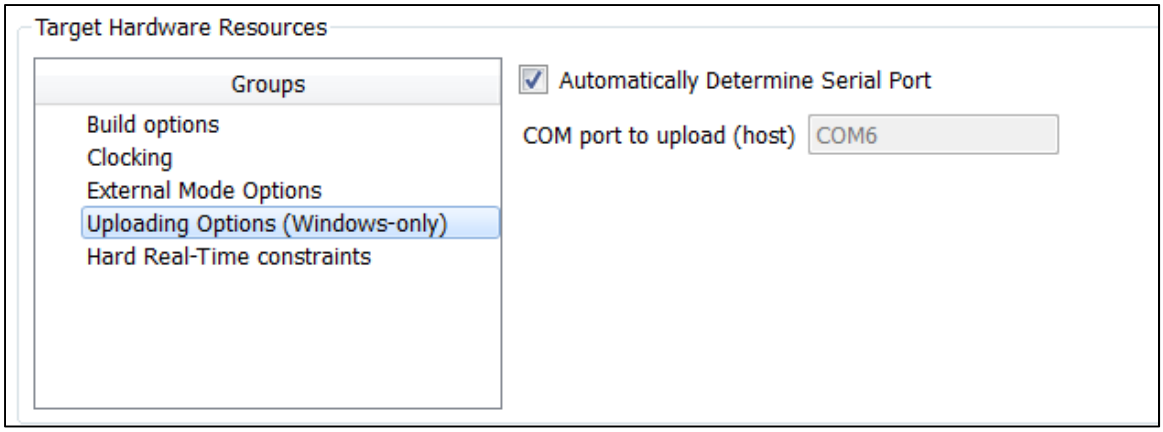
2) Build Options:
    - Build – selecting this will just build the Pixhawk firmware image in /px4/Firmware/Build/ but not actually upload it to the Pixhawk FMU
    - Build, load and run – this will build and upload to the Pixhawk FMU. How it decides to upload is dictated by the “Uploading Options” . Note that in Windows this option does nothing



3) Clocking: Currently not modifiable. Typically, this parameter is utilized by Processor-in-the-Loop. This feature is currently not implemented in the Pixhawk PSP

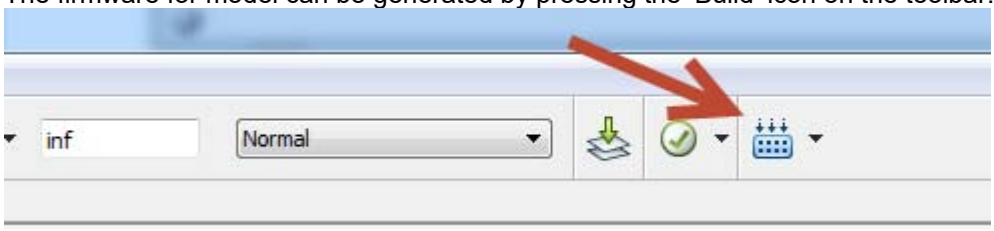
4) External Mode Options – Please see the external mode chapter documentation. These options configure which serial port settings to use to setup external mode communication.

5) Uploading Options (Windows Only) – For uploading to the Pixhawk FMU, we can either force it to connect to a port manually or we can tell MATLAB to search for the correct COM port and connect automatically. Once MATLAB determines the COM port it will continue using it without having to search again or until the COM port value changes for connecting to the PX4 FMU.



4.3.3 Building the Firmware

The firmware for model can be generated by pressing the ‘Build’ icon on the toolbar:



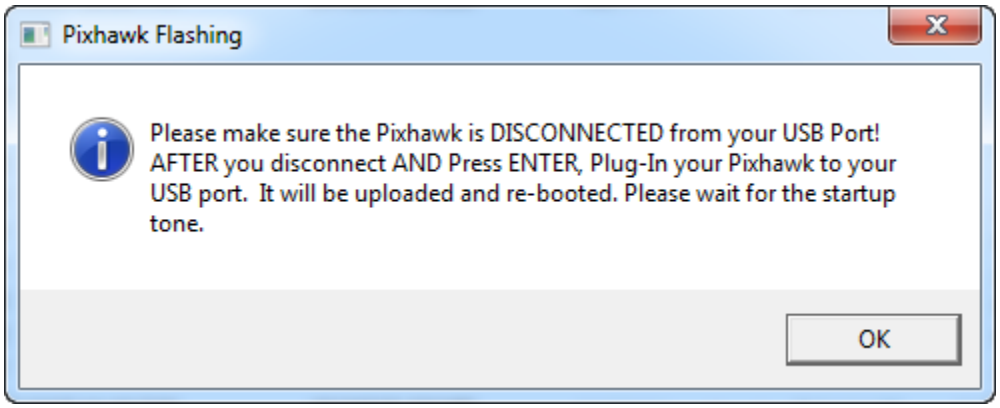
The firmware will then start to build, starting with the generated code then along with the rest of the Pixhawk Firmware using CMake. In Windows 10, the bash terminal will open shortly and begin cross compilation. In Linux, the build will occur within MATLAB/Simulink.

4.3.3.1 Build, Download and Run (Linux)

NOTE: please ensure putty or any connection to the Nutshell terminal is closed before attempting an upload!

If the “Build, Download and Run” option was selected in the hardware implementation panel then the next chain of events will occur after the build process is completed:

The Diagnostic Window will show the progress of the build process. When the firmware is ready and the ‘Build, Load, Run” option is selected, the user will be promoted to make sure that the pixhawk is NOT currently plugged into the computer USB port (see pop-up dialog below). Press OK on this pop-up dialog, then plug in the pixhawk into the USB port. This will start the flashing process. When the process is complete, the PX4 will re-boot and you should hear the start-up tune.



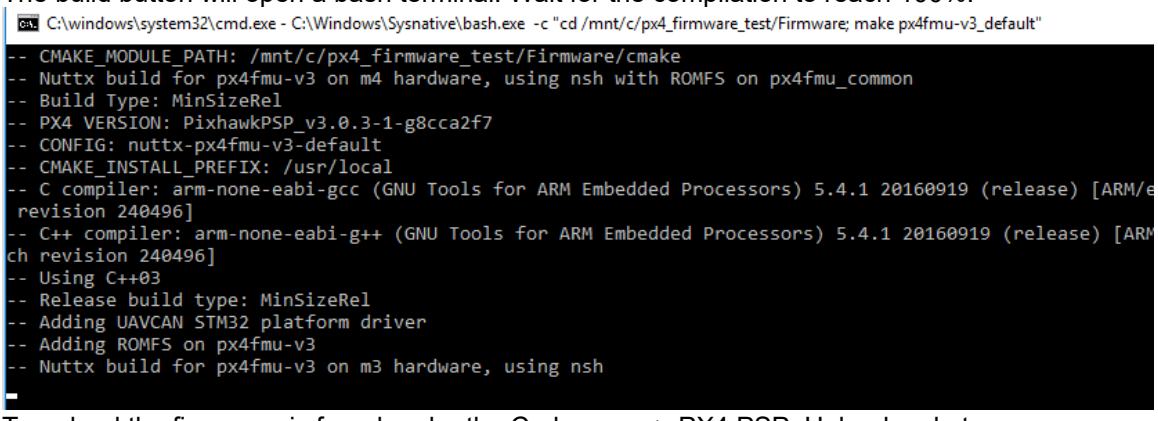
A successful upload using the “Build, Load and Run” option looks something like this in the Simulink Diagnostic Viewer

```
Using Pixhawk PSP COM Port Settings: COM6
Loaded firmware for 9,0, size: 960568 bytes, waiting for the bootloader...
If the board does not respond within 1-2 seconds, unplug and re-plug the U
PX4_SIMULINK = y
Found board 9,0 bootloader rev 4 on COM6
50583400 00ac2600 00100000 00ffffff ffffffff ffffffff ffffffff ffffffff 8c
48943dea 110b788a ba852c6d bbdac2be 082282fd b3e487f6 40dab6dc d49a34fe 3d
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff type: PX4
Erase  : [                    ] 0.0%
Erase  : [=                   ] 5.6%
Erase  : [==                  ] 11.1%
Erase  : [===                 ] 16.7%
Erase  : [====                ] 22.3%
Erase  : [=====               ] 27.9%
Erase  : [======              ] 33.5%
Erase  : [=======             ] 39.0%
Erase  : [========            ] 44.6%
Erase  : [==========          ] 50.2%
Erase  : [===========         ] 55.8%
Erase  : [============        ] 61.4%
Erase  : [=============       ] 67.0%
Erase  : [==============      ] 72.7%
Erase  : [===============     ] 78.3%
Erase  : [================    ] 83.9%
Erase  : [==================  ] 89.4%
Erase  : [====================] 100.0%

Program: [=                   ] 6.7%
Program: [==                  ] 13.4%
Program: [====                ] 20.1%
Program: [=====               ] 26.9%
Program: [======              ] 33.6%
Program: [========            ] 40.3%
Program: [=========           ] 47.0%
Program: [==========          ] 53.7%
Program: [============        ] 60.4%
Program: [=============       ] 67.2%
Program: [==============      ] 73.9%
Program: [================    ] 80.6%
Program: [=================   ] 87.3%
Program: [==================  ] 94.0%
Program: [====================] 100.0%

Verify : [                    ] 1.0%
Verify : [====================] 100.0%
Rebooting.
```

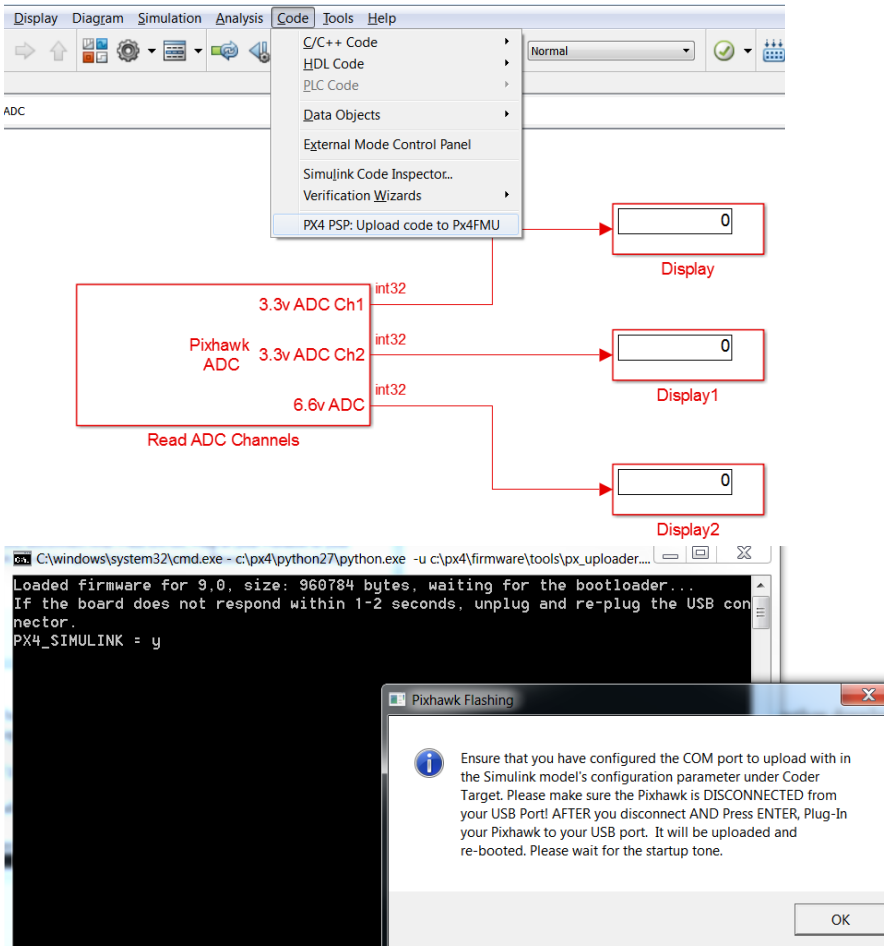
4.3.3.2 Build Only and Manual Download (Windows)

NOTE: please ensure putty or any connection to the Nutshell terminal is closed before attempting an upload!

The build button will open a bash terminal. Wait for the compilation to reach 100%.



To upload the firmware, is found under the Code menu > PX4 PSP: Upload code to Px4FMU. Ensure that the PX4 device is plugged in during this time.



4.3.4 Starting the PX4 Simulink Application

To start the application you can call the command

```
px4_simulink_app start
```

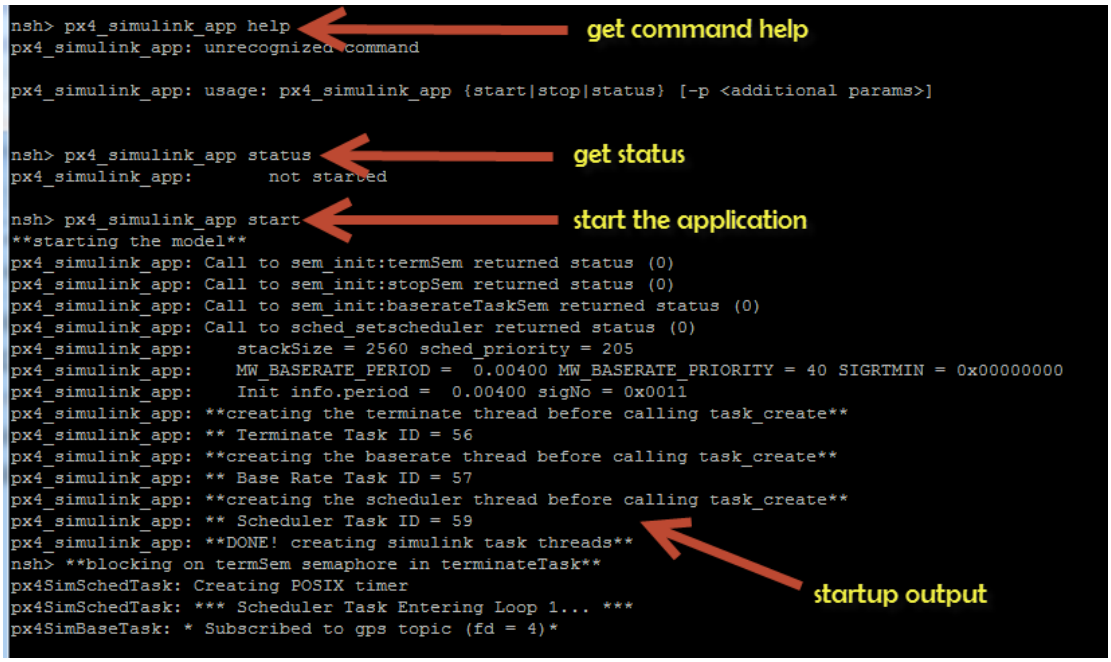
If you have modified your rc.txt file you can automatically start the generated application as soon as the board powers up. The app can be stopped using

```
px4_simulink_app stop
```

```
[px4_simulink_app] usage: px4_simulink_app {start|stop|status} [-p <additional params>]
```

4.3.5 Firmware and Code Generation structure

The Pixhawk PSP generates source code from the model, creates a binary which is then added as a built-in command in the NuttX OS running on the pixhawk. The built-in command is called ***px4_simulink_app*** and it has a command line interface to control its start and stop condition. This application should be included as part of the boot-up script.



During execution the ***px4_simulink_app*** will spawn a task called “PX4_Simulink _Tasks”. When the application initializes it spawns a task which is used to spawn several threads. These threads are the following: base rate thread, subrate thread, a scheduler thread or a terminate thread. The number of subrate threads are dictated by the number of sample-times you have in the model and if the model is set to multi-tasking.

When the application has ended these threads will terminate along with the “PX4_Simulink _Tasks”.

The source file nuttxinitialize.c and PX4_TaskControl.c is responsible for spawning these threads, semaphores and so forth to execute the generated code at the specified sample rates in the Simulink model. This source file can be found in \psp\pixhawk\src

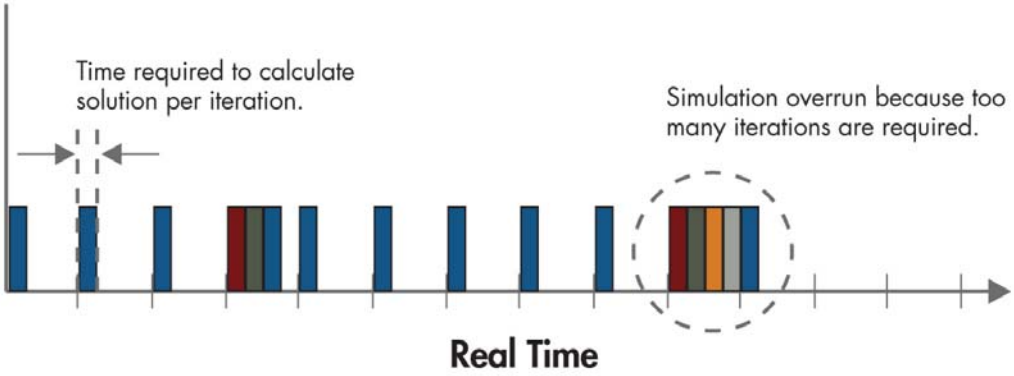
In the previous versions of the Pixhawk PSP we would spawn a thread called schedlerTask which would setup a semaphore that waits on a POSIX timer using functions such as timer_create. Using this method, it was observed that there was jitter in the pace of execution. While this jitter was not enough to cause instability in the system, it was enough to warrant an update. We now employ a High-Resolution Timer (HRT) which was observed to have less jitter to post the base-rate semaphore which is used to set the execution pace of the base-rate thread. To read more about this go here:

https://pixhawk.org/dev/accurately_timed_operations

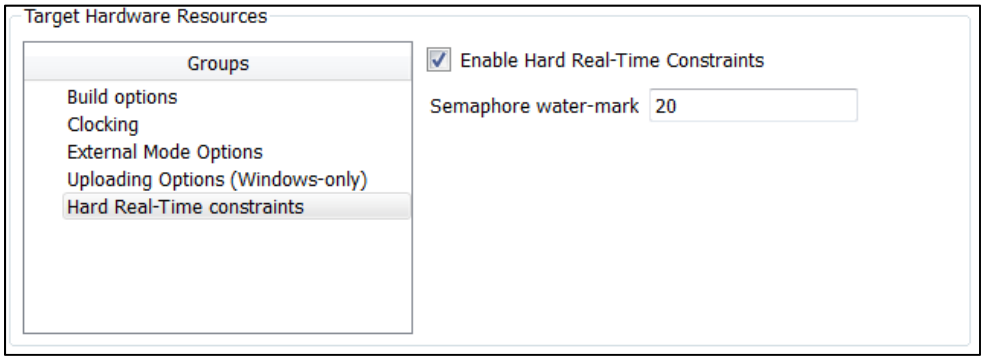
4.3.6 Hard Real-Time Constraints

It is highly recommended to NOT use this during flight tests as there is a chance the system will auto-shut down in midflight.

In the R2016a/R2016b PSP release, a new feature has been added in here to allow users to determine if the flight algorithm is able to meet scheduling deadlines by examining task over-run occurrences. By definition, task over-run means that the generated code was not able to complete a call complete it's task in the specified sample time set by the Simulink model. This is illustrated below:



When enabling Hard Real-Time constraints, the generated code will auto-shut down and report to the Nuttshell terminal when task over-run crosses a certain threshold.



This threshold is dictated by the semaphore water-mark. In the above settings, we allow task over-run to occur at a maximum of 20 times before the application shuts down. The water-mark is here to account for more flexibility in instances where model initialization may have taken longer than a single sample period and causes brief semaphore wind-up but the generated code step function is still able to meet the scheduling deadline.

General use-cases for this feature include:

- Use this to help determine if your flight algorithm does not perform in real time and be able to quantify the severity
- It is highly recommended to NOT use this during flight tests as there is a chance the system will auto-shut down in midflight. You will be warned prior to compiling the Simulink model that this option has been enabled

- Useful for observe transient semaphore wind-up due to initialization

- This is only used to measure against the base (ie: fastest) sample-rate. As such, you may want to use this option with a simplified model with a single-rate that you plan to run that section of the algorithm on. With this option enabled, examine 'baseRateTask(void *arg) to see the instrumentation code we introduce to measure for overrun.

Tips for running at faster periods in real-time include:

- Run with 'faster runs' configuration (-o3 optimizations)
- Reduce the complexity in the model
- Think of better ways of partitioning the model to different sample-rates – this will split the model into different threads. For instance, try moving a section of the model which does not need to run as fast as the base-sample rate into a slower sample rate. This will make that part of the model run less frequently and places it in a lower priority thread
- Use fixed-point instead of floating point math if necessary to ease computational complexity

4.4 Using QGroundControl with Pixhawk PSP for sensor calibration

QGroundControl is a utility which can interact with your Pixhawk FMU through calibration routines, mission planning and parameter adjustments. This utility uses Mavlink serial connection to communicate back to the host computer. For more information on QGroundControl please refer to their website:
http://qgroundcontrol.com/

QGroundControl has had many different releases which may or may not work with the Pixhawk PSP. When running px4_simulink_app we recommend disabling several applications such as the commander and mavlink. This is done because

- We currently do not generate code to interact with Mavlink. We also sometimes require the serial port to be free to access for other things (ie: generic UART communication)

- The commander application has full control over the motors/actuators. Because px4_simulink_app was intended to replace the commander application as the main flight controls system, we disable this app. This unfortunately means that QGroundControl cannot be used simultaneously while px4_simulink_app is running if the commander application is disabled.

The QGroundControl calibration procedure calculates sensor offset and compensation needed for stable flight orientation. These parameters are stored in /fs/mtd_params which is persistent readable/writable memory. To learn more about reading/writing parameters please see http://dev.px4.io/advanced-configurations.html

If you wish run through a calibration routine with QGroundControl, you can follow these steps.

Step 1: Remove the SD card and start the FMU
By removing the SD card we force the FMU to load the default applications such as commander/navigator/Mavlink, etc. This will allow you to use QGroundControl to connect to it in a later step.

Step 2: Start Calibration Routine in QGroundControl
You should now be able to connect to the FMU with QGroundControl and begin the calibration routine as described here:
https://donlakeflyer.gitbooks.io/qgroundcontrol-user-guide/content/SetupView/Sensors.html

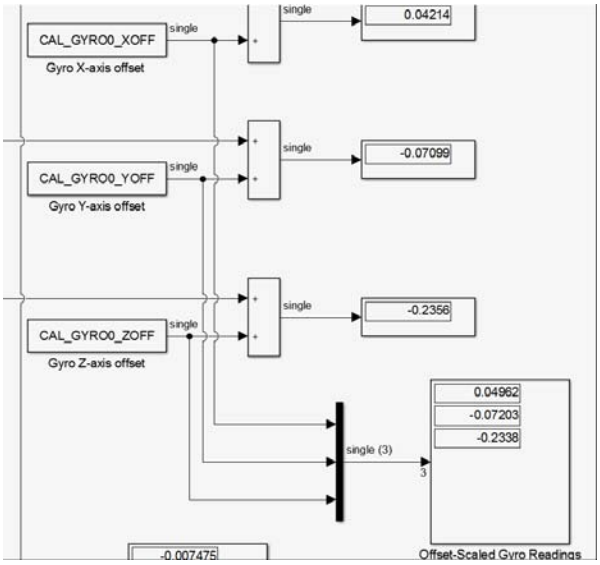
Step 3: Examine Parameters

The calibration routine will update several parameters which will be needed by the attitude estimation / sensor system on the Pixhawk

CAL_ACC1_ID	1114634
CAL_ACC1_XOFF	0.999876
CAL_ACC1_XSCALE	1.04174
CAL_ACC1_YOFF	1.02844
CAL_ACC1_YSCALE	1.01482
CAL_ACC1_ZOFF	1.17767
CAL_ACC1_ZSCALE	1.04818
CAL_GYRO0_ID	2163722
CAL_GYRO0_XOFF	0.0496184
CAL_GYRO0_XSCALE	1
CAL_GYRO0_YOFF	-0.0720289
CAL_GYRO0_YSCALE	1
CAL_GYRO0_ZOFF	-0.233847
CAL_GYRO0_ZSCALE	1
CAL_GYRO1_ID	2228490
CAL_GYRO1_XOFF	-0.0114798
CAL_GYRO1_XSCALE	1
CAL_GYRO1_YOFF	0.0378679
CAL_GYRO1_YSCALE	1
CAL_GYRO1_ZOFF	-0.0696703
CAL_GYRO1_ZSCALE	1
CAL_MAG0_ID	73225
CAL_MAG0_ROT	0
CAL_MAG0_XOFF	0.08003
CAL_MAG0_XSCALE	0.852633

Step 4: Access Parameters in Simulink

These parameters are also accessible in Simulink generated application px4_simulink_app using the Custom Storage Class (CSC) method in one of our example models. Reboot your Pixhawk FMU after the above calibration step with the modified rc.txt file via SD card insertion. Here is a snap-shot of this example model running in external mode where we display several parameters from the above list.

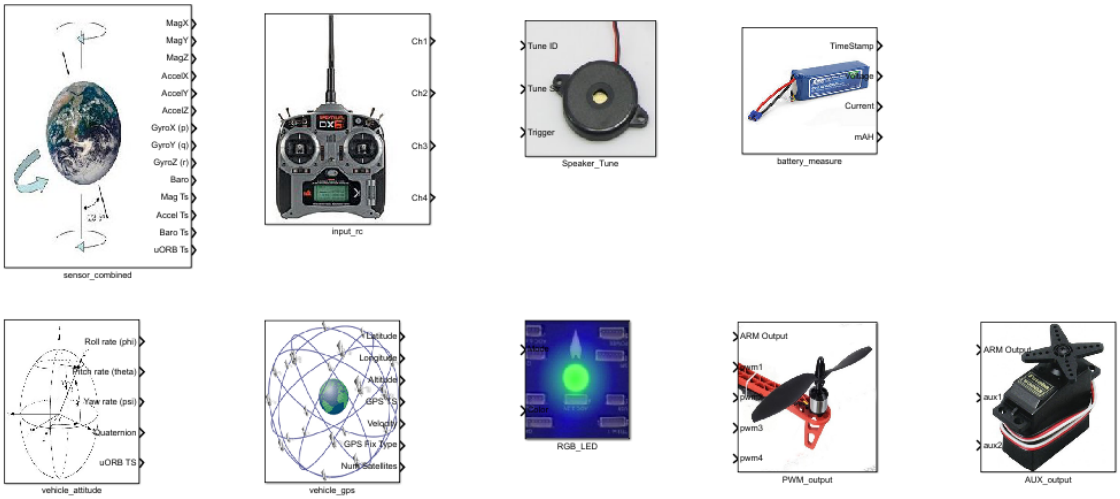


Please examine some of the example models such as `px4demo_Parameter_CSC_example.slx`

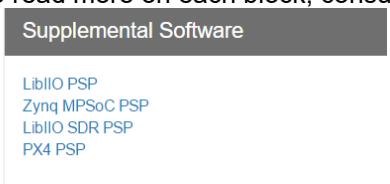
4.5 Simulink Block Library

A few Simulink Blocks have been provided for the user to interface to the hardware of the PX4. These allow for **code generation** only and do not provide for plant modeling behavior. It is recommended that your control model be a Model Block in your Simulink simulation model and then be re-used in your implementation model which would tie in these hardware interface blocks. The library filename is called pixhawk_sllib.slx and will be available in your Simulink Library browser under **Pixhawk Target Blocks**. It consists of four (4) sub-libraries:

1) ADC and Serial Port,
2) Misc Utility Blocks
3) Sensors and Actuators
4) uORB Read/Write Blocks.



To read more on each block, consult the documentation in the MATLAB help guide:



Click on the PX4 PSP then go to “Blocks”

Name	Summary
Read ADC Channels	Outputs external ADC inpusc 6.6V Analog Input. Note that Please see this link for the la
Serial	UART serial port System Obj
AUX_output	AUX PWM Output Block
PWM_output	PWM Output Block
RGB_LED	LED Output Block
Speaker_Tune	Tone Output Block
battery_measure	Battery Input Block
input_rc	RC Input Block
sensor_combined	Sensor Input Block
vehicle_attitude	Vehicle Attitude Block
vehicle_gps	GPS Input Block

4.6 Example Models

There are several simple “test” models available for you to make sure everything is correctly installed and working. It is recommended to try one of these initial test models before trying a complete flight control system model.

Example models are located here:
<Selected PX4 Firmware Directory>\examples\

These files were originally copied from:
<MATLAB Add-ons location>\PX4PSP\code\examples

These example models cover areas such as:

- GPS, ADC and Attitude estimation uORB access
- Writing commands to actuators (PWM and AUX outputs)
- Generic uORB read/write
- Serial transmit/receive
- Interacting with Mavlink & QGroundControl
- Defining PX4 parameters tunable by QGroundControl

For some of these examples, you will need to establish a serial terminal connection to the PX4 hardware with a program such as TerraTerm or PuTTY to examine stdout print statements. This can be done by running the command:

```
nsh> px4_simulink_app start
```

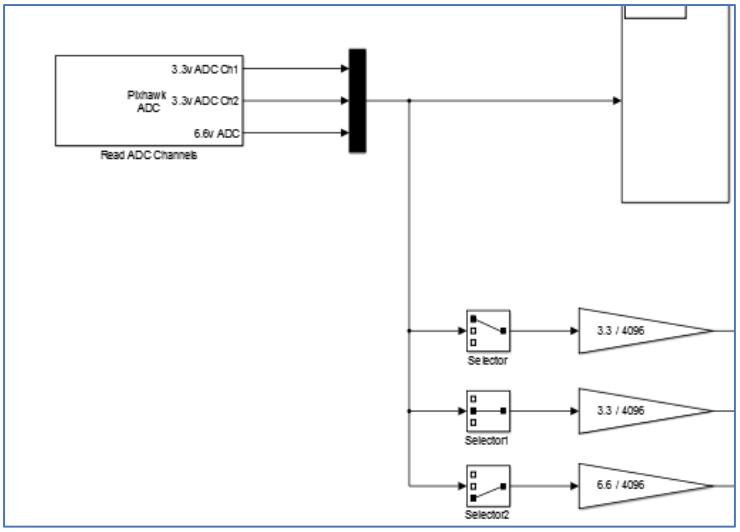
If have edited the rc.txt boot script to start ***px4_simulink_app*** at boot-time, then you will need to stop it, then re-start it with these commands (since there is no stdout console available at boot-up time the printf statements in the code can’t output any text):

Then

```
nsh> px4_simulink_app stop
```

4.6.1 px4demo_ADC_example.slx

Select the different ADC channels through the options in the block. This block was written as a system object.

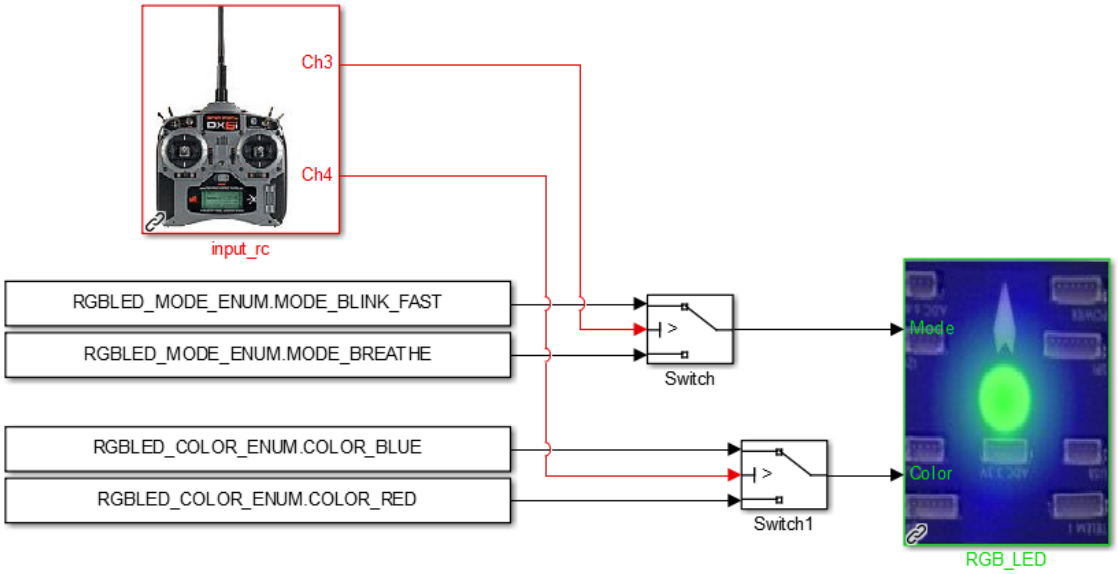


System objects are another alternate method of block authoring. The source code is written as MATLAB class. To view the source code, a link is provided in the block description.

NOTE: This model is to be demonstrated using external mode

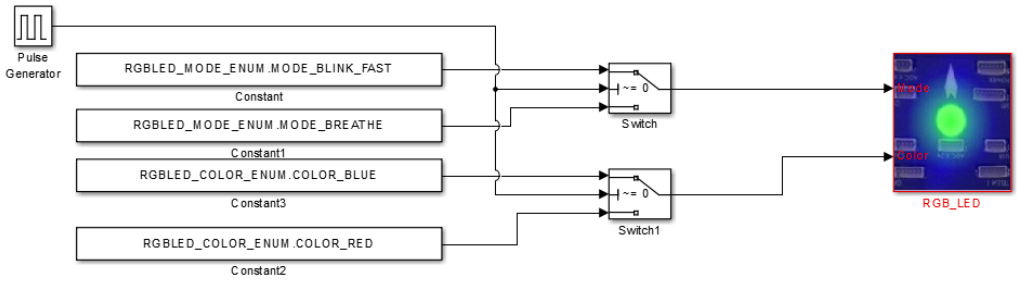
4.6.2 px4demo_input_rc.slx

This model will test the RC transmitter block. Use the RC Transmitter to control the color and mode of the RGB LED on the pixhawk. Channel 3 is typically the “Thrust” or the left vertical joystick control. Channel 4 is typically the “Yaw” or the right horizontal joystick control.



4.6.3 px4demo_rgbled.slx

A simple model that show how to program the RGB_LED library block. Every second the RGB LED changes from blinking-fast blue color to “breathing” red color.

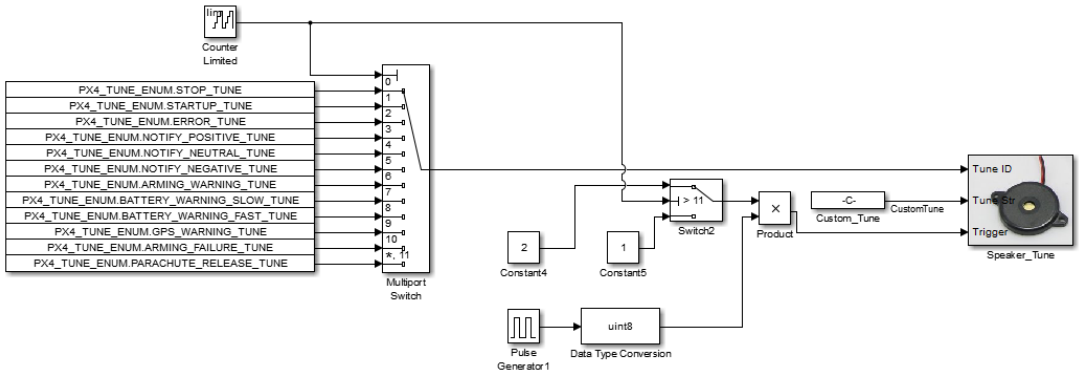


Note: enumerations for LEDs are as follows (in MATLAB) :

```
SL_MODE_OFF (0)
SL_MODE_ON (1)
SL_MODE_DISABLED (2)
SL_MODE_BLINK_SLOW (3)
SL_MODE_BLINK_NORMAL (4)
SL_MODE_BLINK_FAST (5)
SL_MODE_BREATHE (6)
```

4.6.4 px4demo_tune.slx

To test various tunes, this model plays all the pre-defined tunes plus a user-custom tune cycling every 10 seconds.

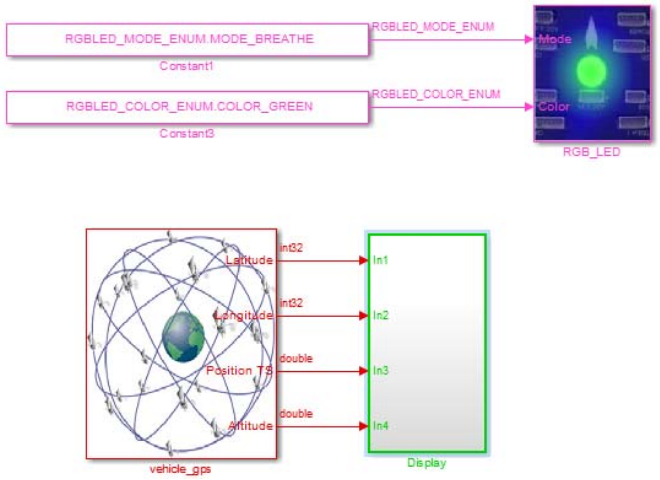


4.6.5 px4demo_gps.slx

A test model has been provided to test out the GPS Block. This model will print out information to a terminal window once a second and the RGB LED will “breathe” Green.

The output will look similar to this:

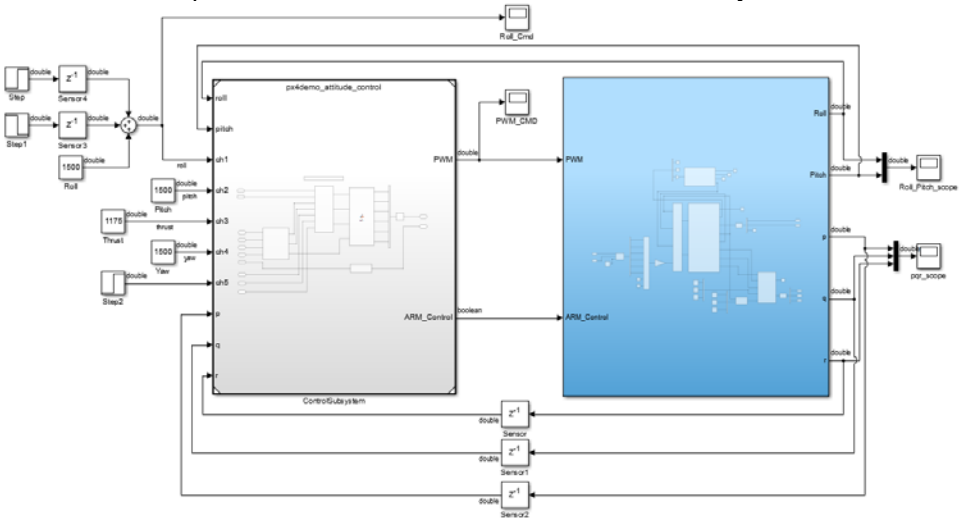
```
GPS->TS: 145797580 LatLon:[424428886 -834372413] Alt:284975
GPS->TS: 146999228 LatLon:[424428895 -834372410] Alt:285028
GPS->TS: 148394689 LatLon:[424428890 -834372428] Alt:285335
GPS->TS: 149594309 LatLon:[424428903 -834372470] Alt:285481
GPS->TS: 150795017 LatLon:[424428926 -834372550] Alt:285325
GPS->TS: 152191507 LatLon:[424428958 -834372596] Alt:285062
GPS->TS: 153395127 LatLon:[424428963 -834372619] Alt:285384
GPS->TS: 154598715 LatLon:[424428951 -834372553] Alt:286188
GPS->TS: 156001888 LatLon:[424428946 -834372496] Alt:286502
GPS->TS: 157191915 LatLon:[424428954 -834372484] Alt:286525
GPS->TS: 158404578 LatLon:[424428968 -834372459] Alt:286505
GPS->TS: 159805210 LatLon:[424429016 -834372511] Alt:285842
GPS->TS: 161004798 LatLon:[424429026 -834372452] Alt:285885
GPS->TS: 162200302 LatLon:[424429036 -834372406] Alt:285856
```



4.6.6 px4demo_attitude_plant.slx

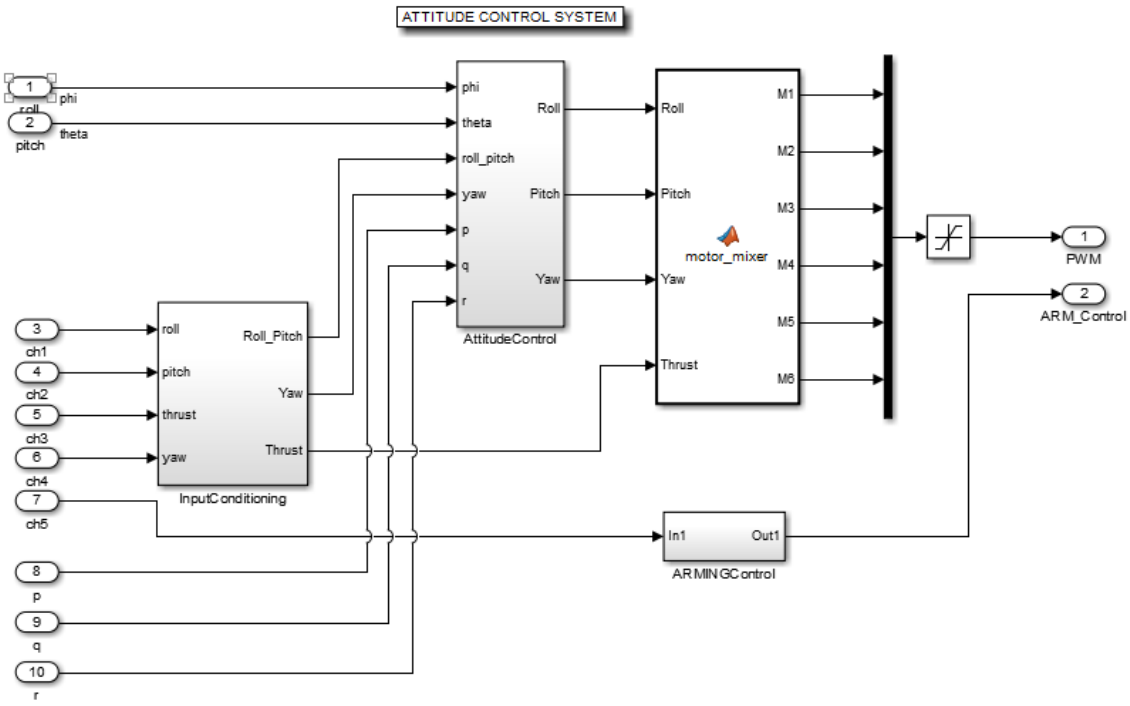
To design and simulation your flight control you will need a test-bench model. It is recommended that you create your test bench model that will provide the stimulus and plant/environment/feedback behavior for your flight control and use a Model (Reference) Block for your control system model.

Here is an example of a model to simulate an attitude control system:



4.6.7 px4demo_attitude_control.slx

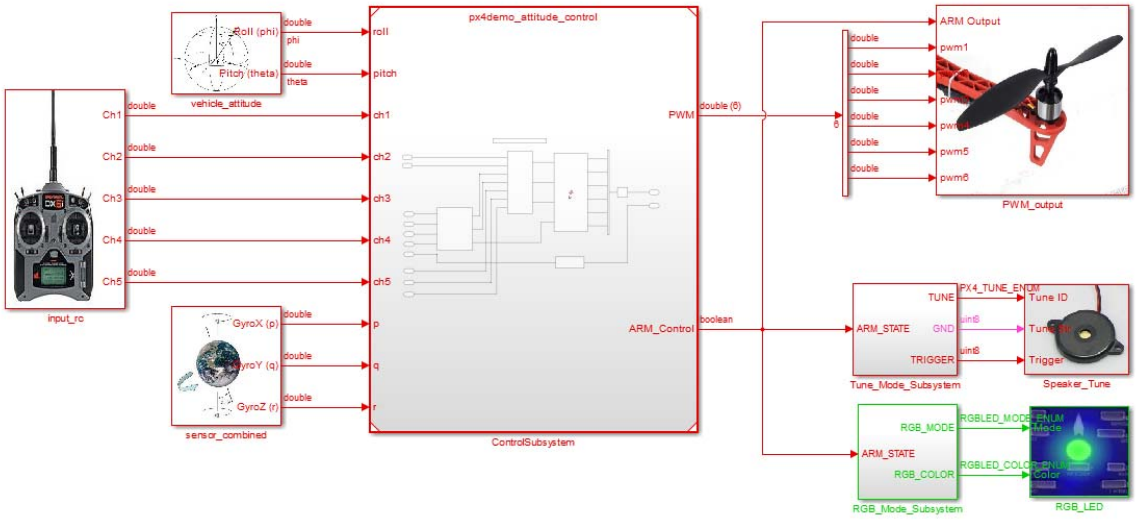
This model contains the heart of the attitude flight control model. It should have the identical configuration parameters as the parent model.



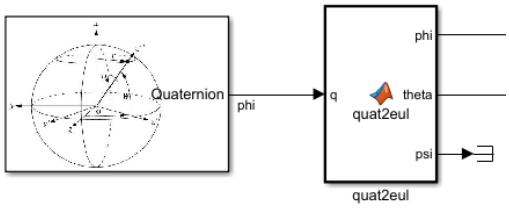
4.6.8 px4demo_attitude_system.slx

After the flight control system has been successfully simulated, it can be used in an "implementation" model that the user can use to generate code and deploy to the Pixhawk PX4 hardware.

Here is the same Control Model referenced in a system model for deployment. The RED/GREEN colors indicate the different sample rates of the model (RED = 250Hz, GREEN = 2Hz).



Note that in Firmware v1.6.5 and beyond the vehicle_attitude uORB topic only outputs in quaternion.

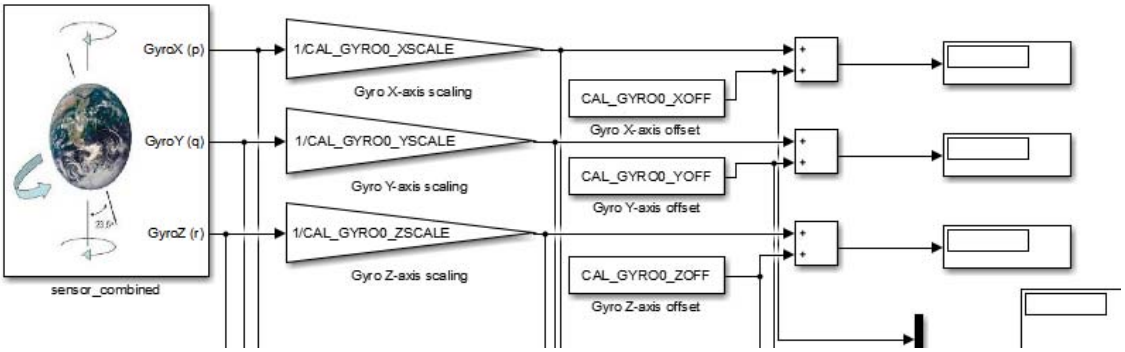


To convert to euler angles this model contains a block which can convert quaternion to euler angles. The code is based off this:

https://github.com/PX4/Matrix/blob/471e96ff6f5f22018b782441c6a8df19d8294181/matrix/Euler.hpp#L132

4.6.9 px4demo_Parameter_CSC_example.slx

px4demo_ParameterUpdate_CSC_example.slx



The Pixhawk Px4FMUv2 uses many parameters to store and access during various operations. Much of these include sensor/actuator calibration data and are stored in flash memory which is accessible by MTD via NuttX.

You can see the list of default parameters here:
https://pixhawk.org/firmware/parameters
Guide to configuring parameters:
http://dev.px4.io/advanced-configurations.html

The Pixhawk PSP allows you to access these parameters using Embedded Coder's Custom Storage Class feature. A parameter is first defined in the MATLAB workspace with specific parameter properties which is then accessed in the generated code.

To make use of this, use the following syntax:

```
Pixhawk_CSC.Parameter( CELL_ARRAY )
```

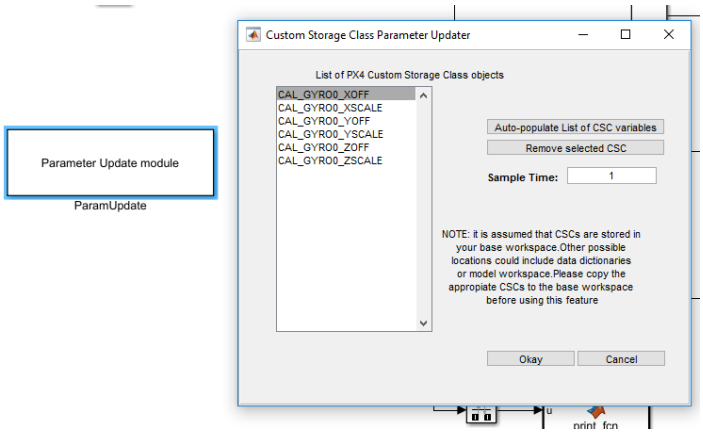
Where CELL_ARRAY is a MATLAB cell array composed of a value (int32 or single) and a string of the parameter. For instance:

```
CAL_GYRO0_XSCALE = Pixhawk_CSC.Parameter( {single(1), 'CAL_GYRO0_XSCALE'}
)
```

Parameters can either be int32 or single/floating precision. Please ensure you select the correct data type and the string name matches. Note that for the model below, all parameters have been defined in the model callback function.

NOTE: This model "`px4demo_ParameterUpdate_CSC_example.slx`" is to be demonstrated using external mode

If parameters are changing and the model requires to use these newly updated parameters, then adding this block in your model can enable such capabilities.



You can use this block to auto-populate a list of storage classes used by the model. You can pick and choose which parameters require updating at run-time.

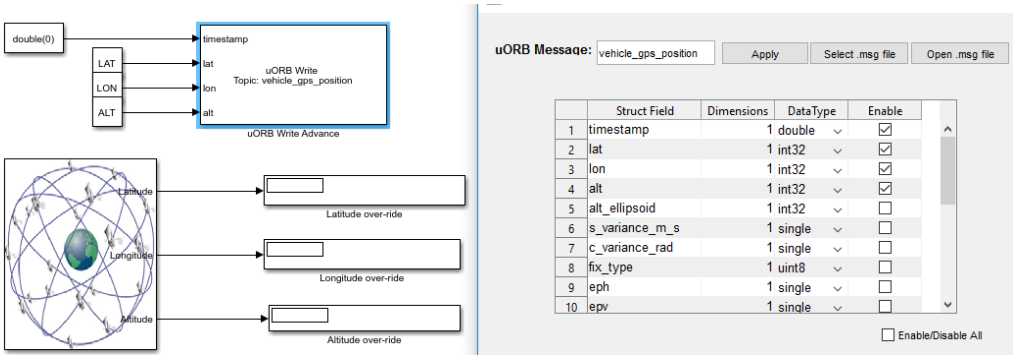
A version of the above model with the update block can be found here:
`px4demo_ParameterUpdate_CSC_example.slx`

This model is to be run without external mode. Printf statements can be viewed by running ***px4_simulink_app.***

4.6.10 px4demo_write_uorb_example.slx

This model demonstrates how one can write data to uORB topics. The uORB write block writes to the struct elements 'lat','lon' and 'timestamp' to the GPS topic. The GPS block then outputs the same value we are writing to by first advertising the GPS topic and then publishing data. You can define whatever topic to write to and its individual struct elements

1) Define LAT, LON and TIME in the MATLAB work-space with assigned values. Ensure they are matching data types to what the block expects.
2) Run in external mode
3) Tune values LAT, LON and TIME and watch the values change in the display from the output of the GPS block

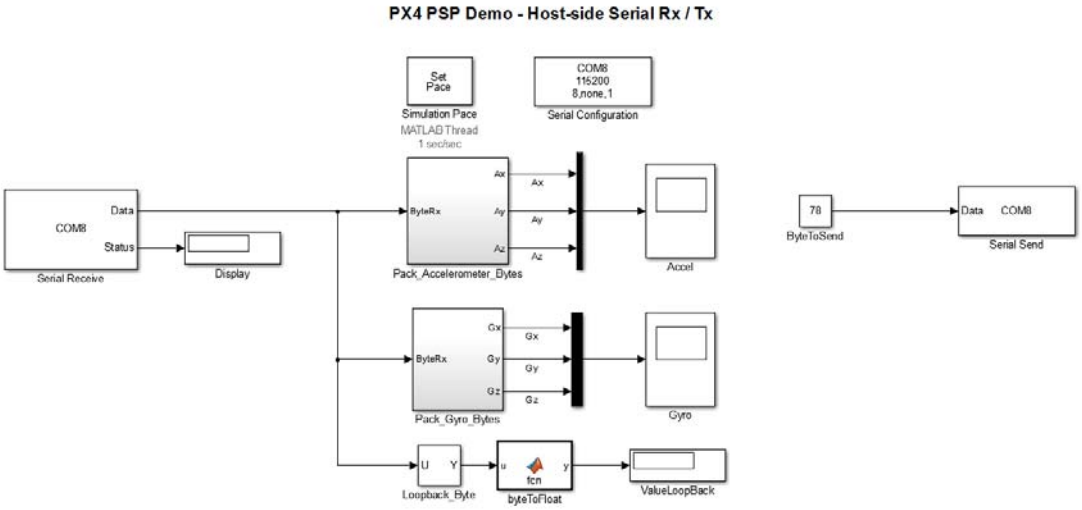


The “uORB Write Advance” block is used which allows you to write to the entire data structure of a uORB topic. Use the UI by double clicking on the block and select which uORB topic struct element to write to. Note that data type and dimenions have been resolved.The older uORB Write block has a limited set but is still accessible in the Simulink PX4 Library

NOTE: This model is to be demonstrated using external mode

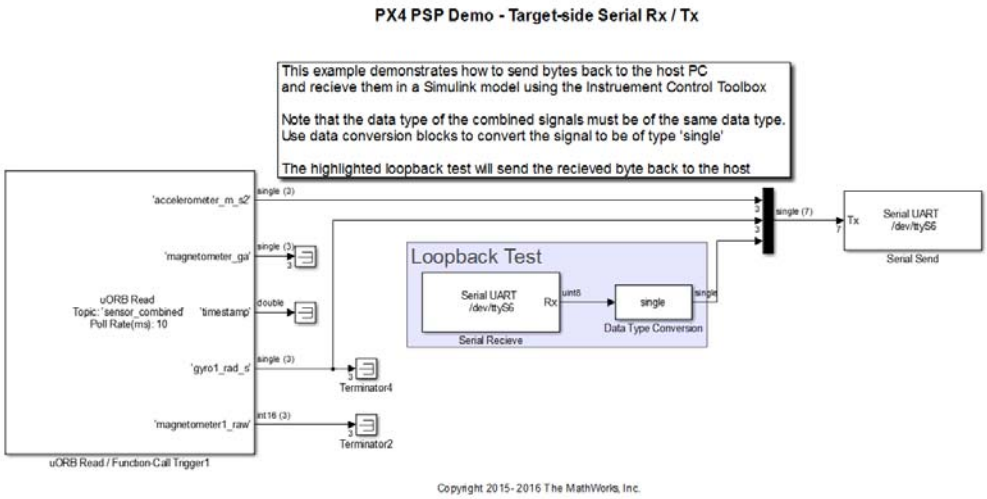
4.6.11 Serial Communication

Two models have been provided to demonstrate how to setup serial communication
px4demo_HostSerial_TxRx.slx



This model does not undergo code generation, it resides on the host PC and is responsible for sending/receiving data to the Pixhawk Px4FMU over serial. The scopes will show accelerometer and gyro readings. A loopback display block is used to show the value that we send to the Pixhawk is sent back.

px4demo_Serial_TxRx.slx



This model is the one that will be deployed the Pixhawk FMU. It will fetch data from a uORB topic and send it off over serial (ttyS6). Loopback data is received and sent back into the serial send block.

4.7 QGroundControl Demos – Parameter Tuning and Messages

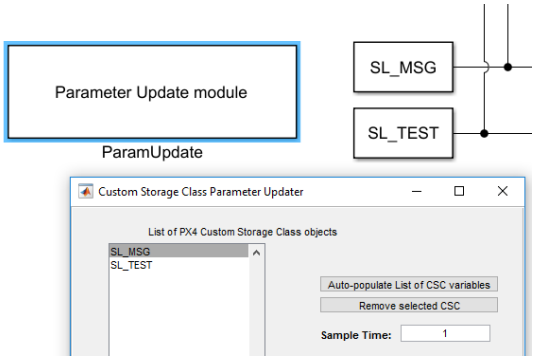
Included within this are demos which allow for parameter tuning and sending debug messages over QGroundControl (QGC)

px4demo_QGC_tune.slx
<Selected PX4 Firmware Directory>\examples\qgc_tune_parameter\

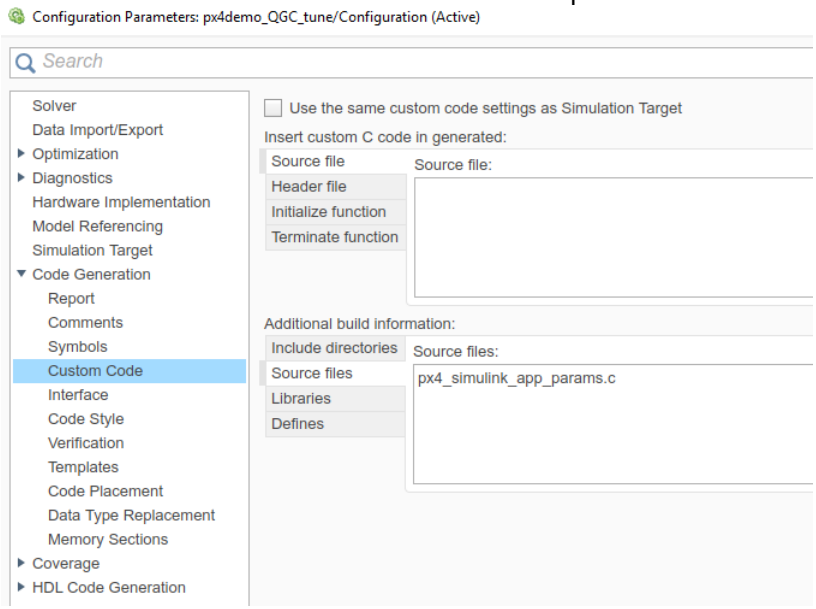
This model contains defines two parameters, “SL_MSG” and “SL_TEST”. In Simulink, they are assigned as custom storage classes. To adhere to PX4’s firmware parameter definition scheme, we also need to include an additional source file `px4_simulink_app_params.c`.

```
/**
 * Sample Simulink Param
 *
 * <longer description, can be multi-line>
 *
 * @unit number
 * @min 0
 * @max 100
 * @decimal 0
 * @increment 1
 * @reboot_required false
 * @group simulink
 */
PARAM_DEFINE_INT32(SL_MSG, 10);
```

As for the model itself, the ‘ParamUpdate’ block was used to ensure parameters are updated when QGC tunes these parameters over Mavlink:



We then add this source file to the custom code panel:



Next, to deploy this model on to the PX4 target:

1) Before compiling the model, you may need to delete parameters.xml within your build folder. This is to force re-generation of this XML file which will contain newly defined parameters from the model.
   `<Firmware Location>\Firmware\build_<fimware_variant>\parameters.xml`
   Where `<firmware_variant>` could be px4fmu-v3_default or px4fmu-v2_default, etc

2) Build the model

   Confirm that after the build, parameters.xml within
   `“<Firmware Location>\Firmware\build_<fimware_variant>\”`
   Actually contains SL_TEST and SL_MSG like so:

```
<group name="simulink">
  <parameter default="10" name="SL_MSG" type="INT32">
    <short_desc>Sample Simulink Param</short_desc>
    <long_desc>&lt;longer description, can be multi-line&gt;</long_desc>
    <min>0</min>
    <max>100</max>
    <unit>number</unit>
    <decimal>0</decimal>
    <increment>1</increment>
    <reboot_required>false</reboot_required>
    <scope>modules/px4_simulink_app</scope>
  </parameter>
  <parameter default="14.7" name="SL_TEST" type="FLOAT">
    <short_desc>Sample Simulink Param</short_desc>
    <long_desc>&lt;longer description, can be multi-line&gt;</long_desc>
    <min>0</min>
    <max>10000</max>
    <unit>number</unit>
    <decimal>0.01</decimal>
    <increment>1.0</increment>
    <reboot_required>false</reboot_required>
    <scope>modules/px4_simulink_app</scope>
  </parameter>
  <parameter default="14.7" name="SL_TEST2" type="FLOAT">
    <short_desc>Sample Simulink Param</short_desc>
    <long_desc>&lt;longer description, can be multi-line&gt;</long_desc>
    <min>0</min>
    <max>10000</max>
    <unit>number</unit>
    <decimal>0.01</decimal>
    <increment>1.0</increment>
    <reboot_required>false</reboot_required>
    <scope>modules/px4_simulink_app</scope>
  </parameter>
```

3) Setup your rc.txt file to enable mavlink. For reference, here is the line of code in the rc.txt that does this

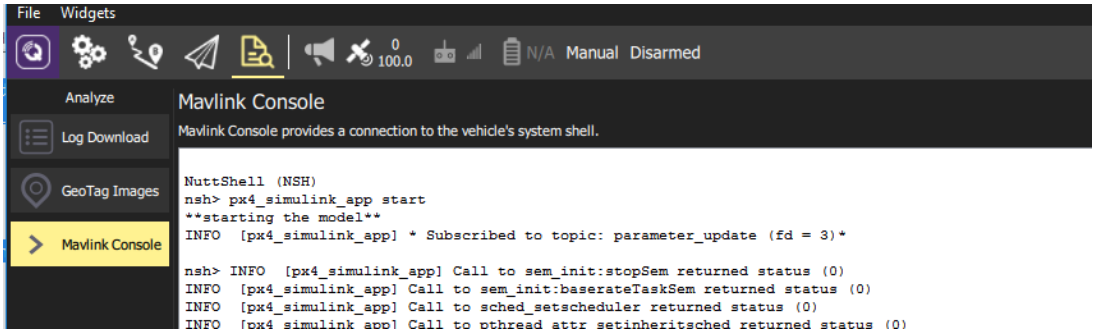
```
mavlink start -d /dev/ttyACM0 -b 57600
```

You will want to make sure that no other application is using the same serial port.

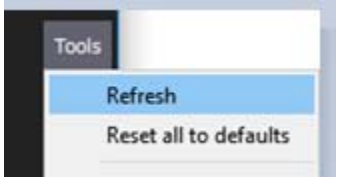
4) Program the board

5) Start up QGC – connection should be established in a few seconds. This has been tested on QGC v3.2.4

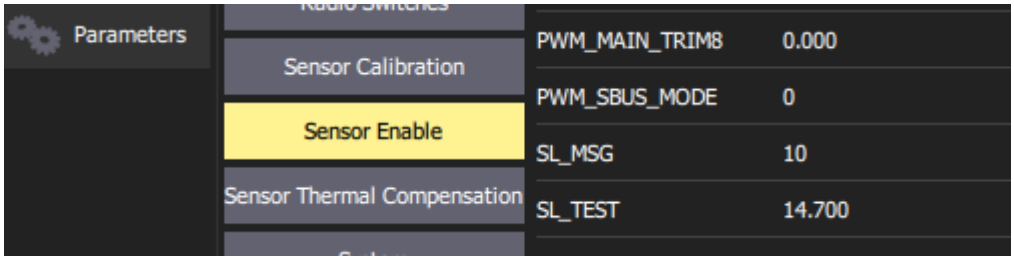
6) If your px4_simulink_app is not running you can start it up by going to the Mavlink console



7) Go back to the parameters. Click on tools->refresh



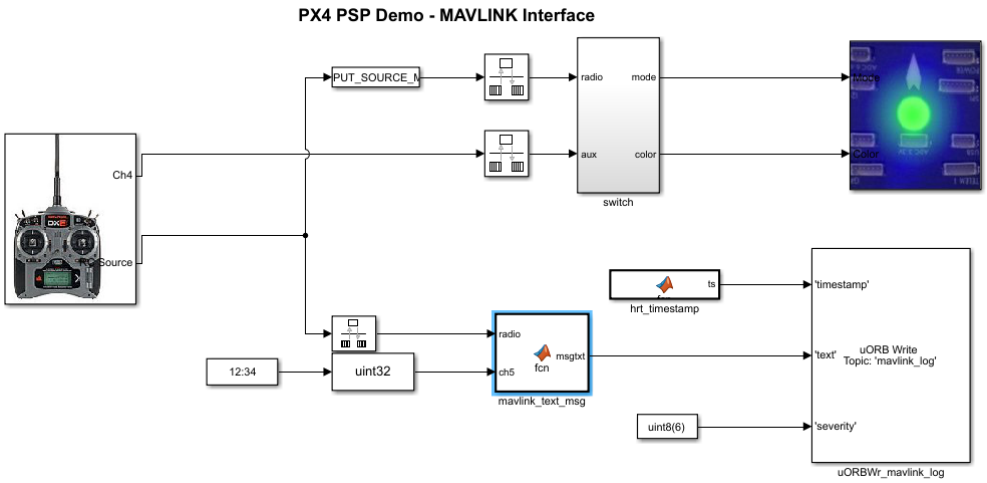
8) To trigger refresh of the parameters you first select a different parameter group other than Default and then clicked on Default again. SL_MSG and SL_TEST should now show up



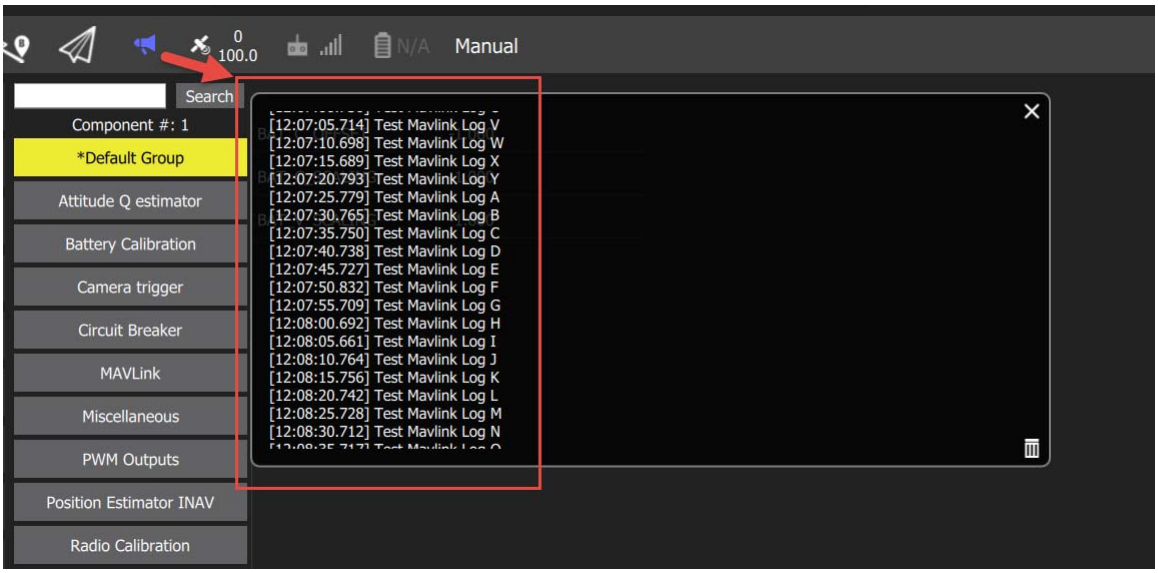
The PX4 application will begin printing values to the screen. You should be able to tune values within here and watch the values change accordingly.

```
INFO  [px4_simulink_app] SL_MSG = 10SL_TEST =  14.7000S
INFO  [px4_simulink_app] SL_MSG = 10SL_TEST =  14.7000S
INFO  [px4_simulink_app] SL_MSG = 10SL_TEST =  14.7000S
INFO  [px4_simulink_app] SL_MSG = 10SL_TEST =  14.7000S
INFO  [px4_simulink_app] SL_MSG = 10SL_TEST =  14.7000S
```

px4demo_QGC_tune.slx



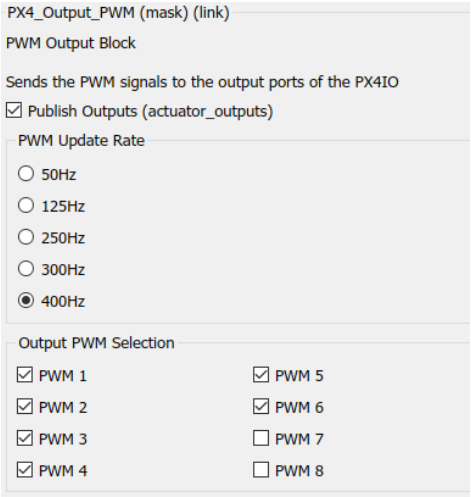
This example model uses rc-controller input and appends it to a message defined within the “mavlink_text_msg” block. This string is then passed on to a uORB write block which writes to the “mavlink_log” topic. This topic is then viewable within QGroundControl as shown below



Like the previous example, MAVLINK must be enabled to establish this communication. You can experiment with trying different “severity” levels to match the expected messaging behavior that’s specific to QGC.

Viewing published PWM data in QGroundControl

A new addition was added to the PX4 PWM write block.



By enabling the “Publish Outputs” this block will also publish data to the uORB topic “actuator_outputs”.

5 Building your own custom Simulink Block

There are several reasons you may want to consider building your own Simulink block. The most common reason is the need to interface generated code with custom hand-code. This could be to interface with driver code which talks to various sensors/actuators or to send data over to another interface. Whatever the reason may be, MATLAB and Simulink offer many ways to accomplish this.

5.1.1 S-Function Approach

All the blocks in this Pilot Support Package were created by writing S-functions with TLC and System Objects.

There are many ways to create S-Functions and the accompanying TLC code.

- Write it by hand along with the TLC from scratch
- Use S-function builder
- Use Legacy Code Tool to interface existing hand-written code.
- Use a combination of all the above. S-function builder or Legacy Code Tool can be used to create a starting point for you to start modifying the S-function MEX source file as well as the TLC.

We have provided an example in this version of the PSP (C-MEX and TLC) for users to learn from. Please See:

<PSP install>\px4\examples\BlockCreation\

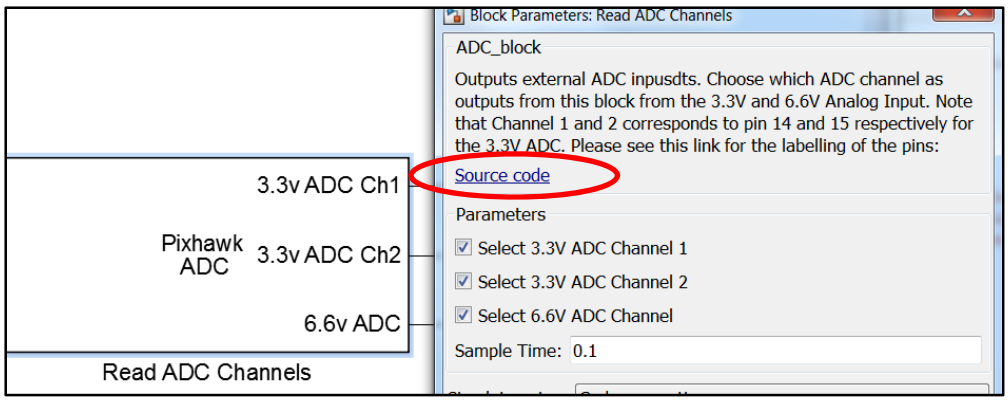
Apply the MEX command on the sfun_px4_battery_example.cpp block to generate a valid MEX file for the S-function. Use this block as an example as to how to create blocks with S-Functions.

For more documentation on S-Functions, please see:
http://www.mathworks.com/help/releases/R2016a/rtw/block-authoring.html

5.1.2 MATLAB Function blocks and System Objects

The logging block was written using a MATLAB Function block. MATLAB Coder syntax is used to describe the interface to hand-code. Please examine the contents of this block for more information on how this was accomplished

Another method that exists is using System Objects. These types of blocks make use of MATLAB Coder's capability of transforming MATLAB Code into C-code. System Objects are written using an object-oriented approach. Please see the ADC and Serial blocks as examples of how to write such blocks.



Click on the 'Source code' hyperlink to open up the MATLAB System Object code for these types of blocks.

For more information on MATLAB Coder and System Objects, please also see:

https://www.mathworks.com/help/simulink/ug/creating-an-example-model-that-uses-a-matlab-function-block.html

https://www.mathworks.com/help/simulink/slref/coder.ceval.html

https://www.mathworks.com/help/simulink/system-objects.html

6 Limitations

The supplied Simulink blocks do not support any simulation behavior. These are merely there to provide code generation to interface the control system to the actual hardware drivers necessary in the firmware. It is advised that you use **Model Referencing** to separate your control system so that you can re-use the model in your simulation as well as the implementation model (used for code generation).

Currently, the optimization option "**Inline Parameters**" must be turned on. This eliminates the use of global data being created which has shown to cause compilation errors due to limited global memory space.

6.1.1 Support for HIL / Mavlink

We currently do not support interactions with HIL or Mavlink with the px4_simulink_app. This is something we wish to investigate in the future and will require significant changes / updates to the way we generate code for this application. Additional code will need to be added to each of the blocks to allow routing of signals when in a HIL environment. If you have suggestions or contributions to help in this area, please feel free to reach out to MathWorks Pilot Engineering.

6.1.2 Supporting C++ uORB Message Data Structures

The current uORB read block is only able to convert uORB messages into Simulink bus objects if the topic is not treated as a C++ object. Several messages are treated as a C++ object where the data structure will not be compatible in C. At the moment, we only generate C code for ***px4_simulink_app***. This means that thing such as memcopies or

memory layout cannot be assumed to be contagious. Here's an example of a uORB topic that uses C++ notation.

Message File: **battery_status.msg**

```
uint64 timestamp        # microseconds since system boot, needed to integrate
float32 voltage_v       # Battery voltage in volts, 0 if unknown
float32 voltage_filtered_v  # Battery voltage in volts, filtered, 0 if unknown
float32 current_a       # Battery current in amperes, -1 if unknown
float32 current_filtered_a  # Battery current in amperes, filtered, 0 if unknown
float32 discharged_mah      # Discharged amount in mAh, -1 if unknown
float32 remaining       # From 1 to 0, -1 if unknown
int32 cell_count        # Number of cells
bool connected          # Wether or not a battery is connected
#bool is_powering_off       # Power off event imminent indication, false if unknown


uint8 BATTERY_WARNING_NONE = 0       # no battery low voltage warning active
uint8 BATTERY_WARNING_LOW = 1        # warning of low voltage
uint8 BATTERY_WARNING_CRITICAL = 2   # alerting of critical voltage

uint8 warning    # current battery warning
```

The header file which gets generated looks something like this:
C:\px4\Firmware\build_px4fmu-v2_default\src\modules\uORB\topics\battery_status.h

```
#ifdef __cplusplus
struct __EXPORT battery_status_s {
#else
struct battery_status_s {
#endif
        uint64_t timestamp;
        float voltage_v;
        float voltage_filtered_v;
        float current_a;
        float current_filtered_a;
        float discharged_mah;
        float remaining;
        int32_t cell_count;
        bool connected;
        uint8_t warning;
#ifdef __cplusplus
        static const uint8_t BATTERY_WARNING_NONE = 0;
        static const uint8_t BATTERY_WARNING_LOW = 1;
        static const uint8_t BATTERY_WARNING_CRITICAL = 2;

#endif
};
```

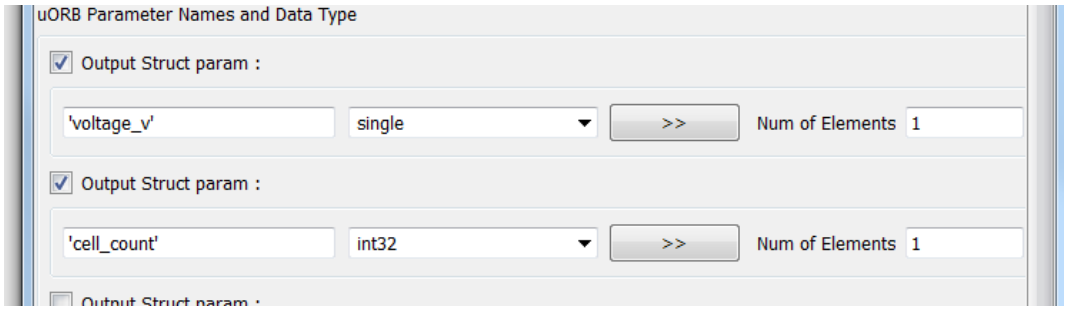
The battery_status uORB topic was written with non-C struct notation:

BATTERY_WARNING_NONE,
BATTERY_WARNING_LOW,
BATTERY_WARNING_CRITICAL.

Because the PixhawkPSP generates C code we cannot instantiate this struct and copy data elements such as BATTERY_WARNING_NONE over like a normal C struct without getting a compiler error. The "Battery_Status" uORB data structure must be treated as a 'singleton' or global since it contains global data.

We have provided a block which is like the uORB read block in the current library but supports the ability to access struct elements from a C++ uORB object, however, the ability to actually read the global elements is not supported. Please look at the example inside: **\px4\SampleSFcn**

Example:



However, attempting to do this:

Will result in compiler error. Therefore, the official shipped version of the uORB read block that uses bus objects will reject these types of data structures completely while this version of the block will still allow you to use any data structure.

If you wish to support the C++ global member variables such as the one in the example above, you will probably need to write your own block that does a copy of data from a C++ struct into local C variables. This function/code could live inside a C++ source file and gets used only if the topic will contain C++ data-structures. Consult the previous chapter on block creation for more tips on how to do this as well. This may be addressed in a future version of the PSP. Alternatively, you can try editing the .msg file and commenting out the "constants" used. This will work only if you aren't running other software that may rely on these to be defined as part of the topic structure (e.g. commander).

6.1.3 Updating to a new version of Pixhawk PSP

There may be times when a new version of the Pixhawk PSP will need to be installed. This could be based on newer versions of Simulink being released, bug fixes, new blocks or enhancements of both the base product and the PSP.

If you run into any issues with the use of this PSP please contact your MathWorks sales representative or Pilot Engineering group directly. Do not go through technical support for issues with this PSP. Do go through technical support for issues related to MATLAB/Simulink outside the scope of this PSP.