AR Drone 2.0 Toolbox: Word documentation

## This document is a collection of all the individual pages that make up the documentation provided with the AR Drone 2.0 toolbox.

## The documentation is still under development and open to feedback

## Last updated by Sanne Marx on 9-6-2016

AR Drone 2.0 Toolbox

The AR Drone 2.0 Toolbox provides hardware support for the Parrot AR Drone 2.0, a commercially available quad-copter.



**Contents:**

**Concepts**

Learn about how the AR Drone 2.0 functions, driver support is achieved and code generation works for the AR Drone 2.0 target.

**Examples**

Follow the numbered examples from getting your first model running on the AR Drone 2.0 until getting it to fly or follow full step by step instructions for getting the AR Drone 2.0 ready to be used as a demo.

**Documentation**

External information resources about the drone hardware and the software implementation and release notes for this toolbox.

# Concepts

The here included subsections explain the global concepts involved in the creation and use of the AR Drone 2.0 toolbox.

## **Ar Drone 2.0 hardware**

The AR Drone 2.0 contains various sensors, processing boards and motors. In order to be able to fly the AR Drone 2.0 it is important to understand the hardware limitations.

## **Driver support**

Reading the sensor data or video streams as well as sending control inputs to the motors of the AR Drone 2.0 requires interfacing between Simulink blocks and the drone operating system. This section explains how c code drivers can be implemented as Simulink blocks.

## **Code generation**

Simulink allows for easy prototyping using a graphical interface to construct models. These models have to be translated into c code that is compatible with the AR Drone 2.0. This section explains the workflow that is executed by the Embedded Coder when you press the "Build" button to get the model running on the AR Drone 2.0.

Ar Drone 2.0 specifications

The AR Drone 2.0 contains various sensors, processing boards and motors. In order to be able to fly the AR Drone 2.0 it is important to understand the hardware specifications and limitations. The Parrot AR Drone was released in 2010 with the improved AR Drone 2.0 releasing in 2012 with a faster processing unit, better cameras and more sensitive sensors. As this toolbox provides support for version 2.0 of the AR Drone the hardware discussed below will only be that of the AR Drone 2.0.

In depth information is available via the menu entries and the general specifications can be viewed below.



**General specifications**

Structural:

* Carbon fiber tubes : Total weight 380g with outdoor hull, 420g with indoor hull
* High grade 30% fiber charged nylon plastic parts
* Foam to isolate the inertial center from the engines’ vibration

Processing:

* 1GHz 32 bit ARM Cortex A8 processor with 800MHz video DSP TMS320DMC64x
* Linux 2.6.32 using BusyBox
* 1GB DDR2 RAM at 200MHz

Sensors:

* 3 axis gyroscope 2000°/second precision
* 3 axis accelerometer +-50mg precision
* 3 axis magnetometer 6° precision
* Pressure sensor +/- 10 Pa precision
* Ultrasound sensors for ground altitude measurement

Motors:

* 4 brushless inrunner motors. 14.5W 28,500 RPM
* 1 8 MIPS AVR CPU per motor controller
* Emergency stop controlled by software
* Fully reprogrammable motor controller
* Water resistant motor’s electronic controller

Cameras:

* HD front facing camera: 1280i \* 720p @ 30FPS
* QVGA down facing camera: 320 \* 240 @ 60FPS

Connectivity:

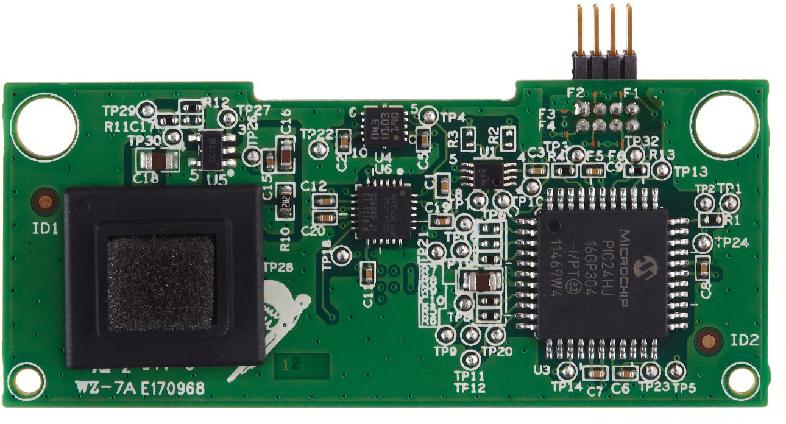
* USB 2.0 high speed for extensions
* Wi-Fi b g n

Sensors

The AR Drone 2.0 contains a number of sensors, most of which are combined as an IMU board to be used for flight control. This section explains the workings, specifications and limitation of all sensors on the AR Drone 2.0

**IMU Sensors**

The IMU board contains the majority of the sensors on the AR Drone 2.0 providing 3 axis accelerometer, gyroscope and magnetometer sensors, barometer and temperature sensor and the IMU board also interfaces with the ultrasound sensor. The individual sensors run at different frequencies with the IMU board outputting the sensor data at 200Hz.



**3 Axis Accelerometer**

The used accelerometer is a Bosch BMA150, digital, tri axial acceleration sensor. It can be used with sensing ranges of +-2g, +-4g and +- 8g. The AR Drone 2.0 uses the +-4g mode.



**Purpose**:

The 3 axis accelerometer measures the acceleration acting on the drone in 3D. The main use for it is to find the rotation of the drone. The gravitational pull acting on the drone is a known acceleration of C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833635282.png. If the drone is in a stable hover then the x and y components of the accelerometer should output a value that maps to zero acceleration and the z component should map to g acceleration.

**Specifications**

* Acceleration range: C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833635343.png
* Digital output size: 10 bit
* Sensitivity: C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833635374.png

**Limitations:**

* The output value 2048 should correspond to 0g with +- 512/g to cover the full range but it can be off slightly and should be calibrated if high accuracy is desired.
* The temperature of the sensor will effect the measurement accuracy.

[**Bosch BMA150 Digital, tri axial acceleration sensor Data sheet**](http://odroid.com/dokuwiki/lib/exe/fetch.php?media=en:universal_motion_joypad:bma150.pdf)

**3 Axis gyroscope**

The AR Drone 2.0 IMU board outputs gyroscope data in 3 axis using the InvenSense IMU-3000.

**Purpose**:

The gyroscopes can measure the angular velocity of the drone. This is useful to aid the accelerometer in calculating the rotation of the drone as it helps distinguish between accelerations caused by drone movements and the acceleration the drone feels due to earth's gravitational pull.



**Specifications**:

* Rate range: C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833635456.png
* Digital output size: 16 bit
* Sensitivity C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833635517.png

[**InvenSense Inc.: 3-Axis Gyroscope: IMU-3000**](https://store.invensense.com/ProductDetail/IMU3000-InvenSense-Inc/415075/)

**3 Axis Magnetometer**

The AR Drone 2.0 has a 3 axis magnetometer which does not appear to be used in the official flight application. Driver support for the magnetometer is implemented in this toolbox but the performance of the magnetometer can be unreliable and should be used with caution.

**Purpose**:

The magnetometer can measure the earths magnetic field in three directions, allowing the drone to calculate it's global orientation.

**Specifications**:

* Precision: C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833635578.png

**Limitations**:

* The magnetometer data seems to stop updating at random intervals using the IMU data retrieval method as provided by the Paparazzi community. A workaround to this is to send a reset signal to the IMU board.
* The magnetic field experienced by the AR Drone is specific to where on earth you are and must be calibrated. If you wish to use the AR Drone in an area where high tech machinery is used which uses strong magnets this will interfere with the relatively weak magnetometer signal.

**Ultrasound sensor:**

The ultrasound sensor consists of a Prowave 400ST160 ultrasonic transmitter and a 400SR160 ultrasonic receiver. They are mounted on the bottom of the AR Drone 2.0 and update at 25Hz.



**Purpose**:

The ultrasound sensor can measure the distance to objects below the AR Drone 2.0 allowing for altitude control with respect to objects below it.

**Specifications:**

[**Prowave 400ST/R160 Air Ultrasonic Ceramic Transducers**](http://www.prowave.com.tw/pdf/T400S16.PDF)

**Limitations**:

* The ultrasound sensor does not work when the object below it is closer than 30 cm.
* If the surface below the AR Drone is highly irregular or angled it will scatter the ultrasound waves reducing the performance.
* The ultrasound sensor can only measure relative altitude and can not be used at as high of an altitude as the barometer.

**Barometer:**

The AR Drone 2.0 contains a barometer which measures the air pressure around the drone.

**Purpose**:

As the air pressure drops with increasing altitude the barometer can be used to calculate the AR Drone's global altitude.

**Limitations**:

* The barometer is less accurate than the ultrasound sensor in their overlapping range
* Wind and other weather conditions effect the barometer measurement meaning it has to be calibrated every time weather conditions change.

Actuators

The AR Drone 2.0 is a quadcopter meaning it has 4 motors which can be actuated. Each motor also has it's own LED and these too can be controlled using the AR Drone 2.0 toolbox.

**Motors**

The AR Drone 2.0 has 4 brushless motors, each controlled by their own brushless control board with an ATMEGA8L 8bit microcontroller and a cutout system that turns off the motor if the propeller hits anything.



**Purpose**:

* The motors drive the propellers to achieve flight and can be individually controlled to rotate the drone around the pitch, roll and yaw axis or provide lift to control the drone's altitude. The torque of the motor is increased via a 8.5:1 gear reduction to the propeller.

**Specifcations**:

* Power: 15 Watt
* Rpm range: C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833717261.png
* Control input bits: 9 bit per motor
* PWM control sensitity: 0.195%

**Usage**:

The motors are given commands by writing to the correct GPIO pins (171 to 174) on /dev/ttaO0 and require a specific set of commands to be initialized. The motors are controlled using pulse width modulation (PWM) with the desired power amount being encoded in a 9 bit number for each motor. These are formatted into 5, 8-bit numbers of the form: *001aaaaa aaaabbbb bbbbbccc ccccccdd ddddddd0 .* The first 3 bits are the start bits followed by 9 bits for the PWM signals for each individual motor (a,b,c,d label the motors) and a single end bit.

**LEDs**

The AR Drone 2.0 has a red-green LED on each motor board.

**Purpose**:

The standard AR Drone LED behavior uses the LEDs to show the status of the AR Drone 2.0 using the following codes:

* All LEDs are red: A problem has occurred
* LEDs flash red in sequence: Motor boot up sequence
* All LEDs flash green: The AR Drone 2.0 is taking off or landing
* 2 Green 2 Red LEDs: The green LEDs indicate the front of the AR Drone 2.0 during flight

**Usage**:

As the LEDs are on the motor board the motor board has to be initialized before the LEDs can be used. Changing the LEDs color then can be done by writing the desired modes to /dev/ttaO0 which can be different for each LED.

* Turn LED off: 0
* Turn LED red: 1
* Turn LED green: 2
* Turn LED orange: 3

# Code generation

Using Embedded Coder® we can generate readable, compact, and fast C code for use on the AR Drone 2.0. Embedded Coder enables additional configuration options and advanced optimizations for fine-grain control of the generated code's functions, files, and data. These optimizations improve code efficiency and facilitate integration with legacy code, data types, and calibration parameters used in prototyping and production.

This section will explain how the code for the AR Drone 2.0 is generated and how the resulting code operates on the AR Drone 2.0

## **Contents:**

Build process:

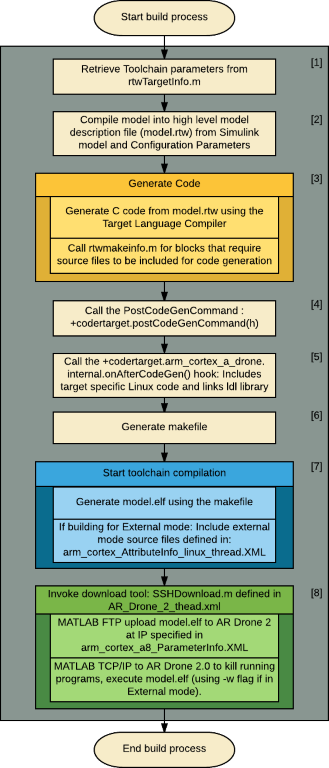
The build process describes the work flow executed by Simulink when you press the Build button, covering both the native Simulink steps as well as additional processes specific to the AR Drone 2.0 target.

Generated code:

The code generated for the AR Drone 2.0 target consists out of multiple separate source files covering different aspects of the models execution. This section will explain how they interact to produce the full executable.

Build process

Simulink allows for easy prototyping using models consisting of various interconnected blocks, Simulink can then generate C code that is compatible with the AR Drone 2.0 from those models. This section explains the workflow that is executed by the Embedded Coder when you press the "Build" button to get the model running on the AR Drone 2.0, an overview of it can be seen in the below image.



**Build process steps:**

When a user presses the "Build" button a number of different steps are executed in a set order by the Embedded Coder. Within these steps there are multiple places for custom code to hook into the build process to define target specific operations. The build process as explained here will contain both the default Embedded Coder behavior and the custom hooks. The numbers in the headers point to where each process is located in the above image.

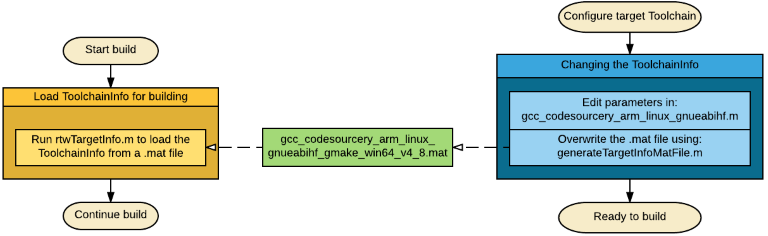
**Toolchain parametersC:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833944061.png:**

The first thing loaded by the build process is the ToolchainInfo describing the registered toolchain. The mandatory ToolchainInfo contains the information of what software and options should be used to generate the final executable for the target, such as the used assembler, compiler, linker and archiver (which are all part of the CodeSourcery toolchain in the case of the AR Drone 2.0) as well as the make application (gmake since it is a UNIX target). Multiple configurations can be created with different options for each invoked tool. The default configurations are:

* **Faster Builds**: Set up the toolchain to save build time by omitting debug and optimization options
* **Faster Runs**: Include toolchain specific optimization options to increase execution time (-O2 as a compiler option for level 2 optimization using the CodeSourcery C compiler)
* **Debug**: Include toolchain specific debug options (-g -D"\_DEBUG" as the CodeSourcery compiler option for the AR Drone 2.0 target)

MATLAB will look for the "rtwTargetInfo.m" file (which is located on the Simulink Project path) to locate the ToolchainInfo. The rtwTargetInfo file points to a .mat file ("gcc\_codesourcery\_arm\_linux\_gnueabihf\_gmake\_win64\_v4\_8.mat") which contains the ToolchainInfo.

**Changing the ToolchainInfo parameters:** If you wish to change the ToolchainInfo you can change the parameters used to generate the .mat file in "gcc\_codesourcery\_arm\_linux\_gnueabihf.m". With the parameters set as desired you can then generate a new .mat file using "generateTargetInfoMatFile.m" at which point rtwTargetInfo.m will use the new .mat file for any future builds. The work flow of changing the ToolchainInfo and how it is accessed by the build process is also visible in the image below.



**Model compilationC:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833944123.png:**

The build process begins with the Simulink software compiling the block diagram to an \*.rtw file. During this stage, Simulink:

* Evaluates simulation and block parameters
* Propagates signal widths and sample times
* Determines the execution order of blocks within the model
* Computes work vector sizes, such as those used by S-functions

When Simulink completes this processing, it compiles a partial representation of the model. This description is stored in a language-independent format in the ASCII file "model.rtw", The "model.rtw" file is the input to the next stage of the build process and contains the parameters set in the Configuration Parameters which describes a number of key options for the next stage of the build process (such as the ToolchainInfo). If you wish to look at the .rtw file you can enable the option "Retain .rtw file" found at "Configuration Parameters >> All Parameters".

**Code generationC:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833944174.png:**

The Simulink Coder™ code generator uses the Target Language Compiler (TLC) and a supporting TLC function library to transform the partial model description stored in "model.rtw" into target-specific code.

The target language executed by the TLC is an interpreted programming language designed to convert a model description to code. The TLC executes a TLC program comprising several target files (.tlc scripts). The TLC scripts specify how to generate code from the model, using the "model.rtw" file as input (which holds all previously specified parameters as the Configuration Parameters which contain the ToolchainInfo are compiled into the "model.rtw" file).

The TLC:

* Reads the "model.rtw" file
* Interprets and executes commands in a system target file, in the case of the AR Drone 2.0 this is the Embedded Coder "ert.tlc" file.
* Interprets and executes commands in block-level target files. For blocks in the Simulink model, there is a corresponding target file that is either dynamically generated or statically provided.
* Writes a source code version of the Simulink block diagram. The user specified options inside the "Configuration parameters" such as the "Optimization" , "Interface" and "Symbols" parameters define how the C code is generated.

**Post code generation commandC:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833944225.png:**

The post code generation command can be set via the "Configuration Parameters >> All Parameters >> Post code generation command" parameter and is the first hook executed after generating code in the AR Drone 2.0 build process. The MATLAB function placed here will be called after the Code generation is complete but before any executables are build. In the case of the AR Drone 2.0 this command is "codertarget.postCodeGenHookCommand(h)", an internal encrypted (p code) file which runs target specific checks on the used parameters and output code. It is passed an instance of the class "coder.internal.ModelCodegenMgr" which contains information about the Embedded Coder paths and the target build options.

**On After Code Gen HookC:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833944256.png:**

The "onaftercodegenhook" parameter in the "arm\_cortex\_a8\_AttributeInfo\_linux\_thread.xml" file points to a MATLAB function which should be called after code generation and is the second hook called after code generation in the AR Drone 2.0 build process. For the AR Drone 2.0 the to be called MATLAB function is the custom function: "codertarget.arm\_cortex\_a\_drone.internal.onAfterCodeGen.m", it performs the following actions:

* Check if the "High priority Value Indicates High Priority Task" parameter is set to true, which is not the case if coming from an older version of MATLAB.
* Replace the host-specific "UDP block" and "linuxinitialize" files with target-specific ones.
* Include the -ldl tag to the linker
* Launch the QEMU emulator if the target is the QEMU ARM emulator instead of the AR Drone 2.0. **Note:** Use of the QEMU simulator is not fully supported.

**Make file generationC:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833944297.png**

For the final step of the Code Generation process a make file is generated, it contains the process of generating a target specific executable based on:

* User specified ToolchainInfo parameters such as the compiler options and executable type.
* Object links for existing C code, for example Simulink blocks created using the Legacy Code Tool.
* Inclusion of the external mode source files if the Configuration Set for External mode is selected.

An extra automatically generated make file is included to set hardware specific Code Sourcery compiler options in: **codertarget\_assembly\_flags.mk**. It contains the following flags:

* **-march=armv7-a :** This specifies the architecture of the used chipset, in this case the ARMv7-A architecture of which the ARM Cortex A-8 is part of.
* **-mcpu=cortex-a8 :** The specific CPU of the AR Drone 2.0 is the ARM Cortex A-8, which is specified using this flag.
* **-mfpu=neon :** The ARM Cortex A-8 has two different hardware floating point hardware accelerators: VPF and Neon. We use Neon as while it is not fully IEEE 754 compliant like VPF is, it is significantly faster (Neon is about 8 times faster according to an example provided by Texas Instruments).
* **-mfloat-abi=softfp :** Specifies which floating-point ABI to use. Permissible values are: ‘soft’, ‘softfp’ and ‘hard’. Specifying ‘soft’ causes GCC to generate output containing library calls for floating-point operations. ‘softfp’ allows the generation of code using hardware floating-point instructions using soft-float calling conventions. ‘hard’ allows generation of floating-point instructions and uses FPU-specific calling conventions.

**Toolchain compilationC:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833944368.png**

After all code is generated by Simulink the external toolchain needs to compile this code into a target specific executable. For this Code Sourcery is used which provides support to compile C code for the ARM Cortex A-8 inside the AR Drone 2.0. During this process the following operations are performed:

1. Call gmake on the previously generated makefile
2. Compile all the C code source files the model depends on and create object (\*.o) files to link them, such as: model.c, external mode source files and any C code generated by S function blocks.
3. Create the standalone executable from the compiled sources

The end result is an .elf file which is the executable type used by the AR Drone 2.0.

**Loading the executable on the AR DroneC:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654833944399.png**

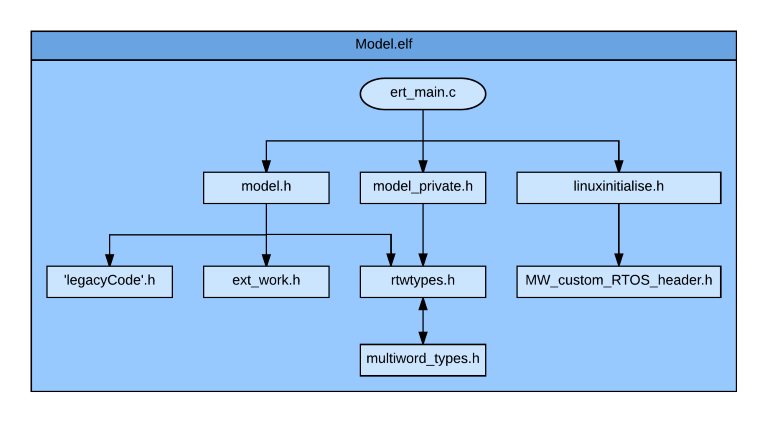
When the executable is created it still lives inside the Build folder and needs to be copied to the AR Drone 2.0 filesystem. To do this the 'SSHDownload.m' file is called since it is set as a toolchain from the AR\_Drone\_2\_thread.XML file. This MATLAB function does the following things:

1. Retrieve the AR Drone 2.0 IP address from the configuration set.
2. Set up an FTP connection with the AR Drone using the MATLAB 'ftp' command to upload the executable to the AR Drone using the MATLAB 'mput' command.
3. Set up a TCP/IP connection with the AR Drone using the MATLAB 'tcpip' command to kill the AR Drone 2.0's default program and execute the new executable.

After this is complete the model will be running on the AR Drone 2.0. If the simulation mode was set to Normal mode this means the model is now executing, if the External mode was selected the model is now waiting for you to connect to it from the model menu using the connect to target button such that it can transfer parameters. After this you can press 'Run' in the model window to execute the model in external mode.

Generated code

The Simulink build process generates multiple C code files, these are combined with existing MATLAB code and your own included legacy code to get the final target specific executable. This section will explain the most important code for the AR Drone 2.0 and how it functions in the final executable. An overview of these files and how they depend on each other can be seen in the image below.



Model.elf

The final AR Drone 2.0 compatible executable of your Simulink model 'model.slx' will be 'model.elf'. The Executable and Linkable Format (ELF) is a common standard file format for executables and a standard binary file format for Unix systems. This file is generated at the end of the toolchain build process by Code Sourcery from the different source files visible in the image above and explained below.

ert\_main

The main entry point of the code is located in ert\_main.c. ERT stands for Embedded Real-Time, using Embedded Coder we are able to generate code that runs in real time on our embedded target, the AR Drone 2.0. It is inside this file that the **main** function is defined.

The **main** function does the following things:

1. **Initialize the model:** The initialization function inside model.c is called.
2. **Initialize External mode:** If the model is configured to run in external mode, the **ert\_main** source will contain an initialization of external mode after which the code waits for the external mode start packet.
3. **Initialize the Real Time Operating System:** The initialization function **MyRTOSInit** within **linuxinitialize** is called to set up the real time threads required by the code. Two arguments are passed to the model, the base rate of the model specified in the configuration parameters and the number of different sub rates required for the model.
4. **Set up a wait thread:** The **main** function performs no other action and simply waits until the **stopSem** parameter signals the model to stop. This is done at the end of the **terminateTask** function within **ert\_main.**

The **ert\_main** file contains a number of functions for starting and terminating tasks.

**baseRateTask:** This is a mandatory function which performs the following steps while no stop command has been received:

1. **Wait until the next real time step**: In real time execution the base rate calculations take less than a single base rate time step takes. This is synchronized by having the task wait.
2. **Update external mode:** Check if an external mode stop command has been received
3. **Call the model step function:** The model step function defined in **model.c** is called, if multiple rates are used the ID of the rate is passed as an argument with '0' defined as the base rate task ID.

**subRateTask (optional):** This function is only present for models with multiple rates, it executes the following steps while no stop command has been received:

1. **Wait until the next real time step**: A sub rate task is synchronized in the same manner as the base rate task but uses a different timer.
2. **Call the model step function:** The model step function defined in **model.c** is called with the base rate task ID as argument.

**terminateTask:** When the base rate task receives a stop command it will exit the while loop and call **terminateTask** which will then:

1. **Signal all sub rate tasks to complete:** If the model has sub rate tasks code is generated that signals the sub rate tasks to destroy themselves, the function then waits for these all to complete.
2. **Shut down external mode:** The shutdown function is called for external mode.
3. **Call the model terminate function:** Within **model.c** the terminate function of the model is defined, this is called for any model specific cleanup processes.

model

The **model.c** file defines the **model\_initialize, model\_step** and **model\_terminate** functions. These functions govern the initialization, real time update and termination of the Simulink model you generated the code for.

**model\_initialize:** The first function called within **model.c** from **ert\_main** is the **model\_initialize** function used to initialize the following things:

1. **Initialize the real-time model:** Reserve the memory needed for the model, block in and outputs, work states and set the stop time and the base rate.
2. **Set up external mode info:** Verify that the model and the generated code are up to date using checksums.
3. **Call legacy initialization functions:** If you make use of legacy code which define initialization functions these are called here.

**model\_step:** This function is called every base rate time step, if all parts of the model run at the base rate this contains the code that represents your Simulink model. If you use multiple rates then this will contain a switch statement such that different step functions are called based on the argument passed to **model\_step** with a step function for every rate, each performing their respective calculations as defined in the Simulink model.

**model\_terminate:** This function will call the terminate functions of legacy code implementations which define a terminate function.

model\_private

The **model\_private** source is used to define private macros used by the generated code to access rtModel, if your model has multiple rates the different step functions will also be externally defined from here.

linuxinitialize

MATLAB with Embedded Coder provides the linuxinitialize source which is used to set up threads that monitor the execution of the code on the AR Drone 2.0 Linux operating system, using the clock on the AR Drone 2.0 it is able to ensure that the tasks are performed in real-time.

**myRTOSInit** is the function call which sets up the scheduler and is called from **ert\_main** and performs the following actions:

1. **Set the affinity for all threads:** The AR Drone 2.0 only has 1 CPU on which all threads will execute.
2. **Create a base rate thread:** The base rate thread attributes are set and then passed to a ptread creation function which will set up the thread that runs at the Simulink model base rate.
3. **Create sub rate threads:** If the model uses sub rates the attributes are set and then passed to a ptread creation function which will set up the threads that runs at rates other than the base rate.
4. **Create the scheduler task:** This thread will be set for the scheduler task, which creates the timer used for synchronizing the real-time execution of the model.

legacyCode

Any legacy code that you include via an S-function block will be included by and called from **model**. These files will usually contain an initialization, step and terminate function.

MW\_custom\_RTOS\_header.h

This header defines the parameters needed by **linuxinitialize.c** such as the base rate task priority, base rate period and number of sub rate tasks. It also externally defines the different Tasks and semaphores used by **linuxinitialize.c**

rtwtypes

This file is used to define the single word data types used by MATLAB as target specific data types including their maximum and minimum values.

multiword\_types

This file is used to define the multi word data types used by MATLAB or specified by you as target specific data types including their maximum and minimum values.

ext\_work.h

This file is the entry point for including the external mode interface and is only included if external mode is enabled for code generation.

# Driver support

Reading the sensor data or video streams as well as sending control inputs to the motors of the AR Drone 2.0 requires interfacing between Simulink blocks and the drone operating system. There are a number of steps required to go from the sensor and actuator interfaces as present on the AR Drone 2.0 Linux environment to C code drivers created by the Paparazzi community and finally wrapping these C code drivers into Simulink blocks ready for code generation.

## **Contents:**

Integrating legacy code

This section explains the process of wrapping C code drivers such that not only can they be used in Simulink but they can also be compiled into target specific code for running on any hardware.

Video drivers:

The AR Drone 2.0 has two video cameras which can be interfaced with using the Video4Linux2 API.

Sensor drivers:

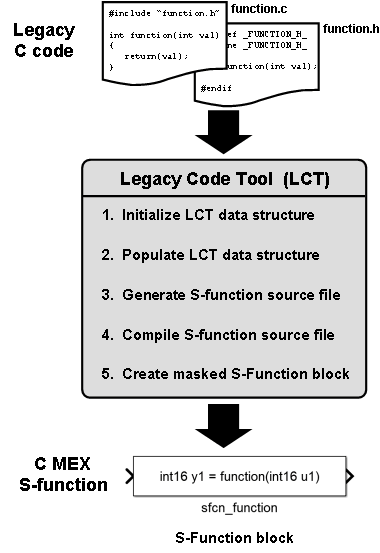
The AR Drone 2.0 IMU can be read using the I2C protocol.

Actuator drivers:

The motor boards on the AR Drone 2.0 can be send commands to control the motor speeds and the LED colors. This requires a specific initialization procedure and knowledge of the structure of the input commands.

Integrating legacy code

Legacy code refers to existing C (or C++) functions, such as device drivers, lookup tables and general functions and interfaces. The Legacy Code Tool allows you to integrate legacy code into Simulink models. The following diagram illustrates a general procedure for using the Legacy Code Tool:



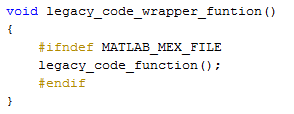
**Wrapping the legacy code**

The Legacy Code Tool can integrate four function specifications into your Simulink block (of which you must specify at least one):

* **InitializeConditionsFcnSpec**: A nonempty string specifying a reentrant function that the S-function calls to initialize and reset states.
* **OutputFcnSpec**: A nonempty string specifying the function that the S-function calls at each time step.
* **StartFcnSpec**: A string specifying the function that the S-function calls when it begins execution. This function can access S-function parameter arguments only.
* **TerminateFcnSpec**: A string specifying the function that the S-function calls when it terminates execution. This function can access S-function parameter arguments only.

For example the IMU driver uses the InitializeConditionsFcnSpec to open a serial connection with the IMU board. The OutputFcnSpec is used to call an update function which retrieves the latest IMU data. Finally the TerminateFcnSpec closes the serial connection to the IMU.

While the C code for interfacing with the IMU could be compiled and run on a PC, the initialization procedure would instantly result in an error as there is no IMU to connect to. To this end the legacy code is wrapped by placing it inside a MATLAB check as below:



This means the "legacy\_code\_function" will only be executed if the compilation target is not MATLAB meaning the IMU block will do nothing when simulated on the PC which is the desired behavior. Wrapping Legacy Code in these MATLAB target checks can be done in two manners:

* Call the unmodified legacy code from a new wrapper source file which checks whether the target is MATLAB or not (as is done for the IMU\_Navdata.c file via IMU\_Navdata\_wrapper.c) ,
* Modify the original Legacy Code C file such that the target check surrounds the desired code (which is how video.c is implemented).

If the legacy code function might be updated in the future and does not need modification before wrapping the wrapper file method makes it faster to update the legacy code implementation.

TODO: Check if this causes differences in code generation

**Generating S functions from legacy code**

S functions which can be used in Simulink simulations are created using the Legacy Code Tool by calling:

specs = legacy\_code('initialize')

This gives you a Legacy Code Tool data structure which allows you to specify the options for the Legacy Code Tool. In specs you must then specify the SFunctionName, desired FcnSpecs (as explained above), source files and paths and code language. The first step is to specify the specs such that an S function can be generated, which is the code that drivers a Simulink block. For the example of the IMU block this looks as below:

IMU\_Block\_def = legacy\_code('initialize');

IMU\_Block\_def.SFunctionName = 'IMU\_Sfcn\_mex';

IMU\_Block\_def.InitializeConditionsFcnSpec = 'void MDL\_IMU\_start()';

IMU\_Block\_def.TerminateFcnSpec = 'void MDL\_IMU\_term()';

IMU\_Block\_def.OutputFcnSpec = 'void MDL\_IMU\_step(IMU\_Packets y1[1], int32 y2[1])';

IMU\_Block\_def.SourceFiles = {'IMU\_Navdata\_Wrapper.c'};

IMU\_Block\_def.IncPaths = {''};

IMU\_Block\_def.Options.useTlcWithAccel = false;

IMU\_Block\_def.Options.language = 'C';

IMU\_Block\_def.SampleTime = 'parameterized';

In this definition there are a few things to note:

* y1 And y2 are the pointers to the IMU navdata packet and checksum flag which will form the two outputs of the IMU Simulink block.
* The only piece of C code that is referenced is the wrapper file. As at first the definition is created to generate an S function for use in a Simulink block, which does not yet need to be compiled into target specific code the IMU\_Navdata.c source is not yet needed as all calls to it are wrapped inside MATLAB target checks.
* The 'useTlcWithAccel' option which speeds up execution by in lining code is set to 'false' as recommended for driver blocks by the Legacy Code Tool documentation.
* The sample time is set as parameterized meaning the user can specify the sample time.

The Legacy Code Tool can then be called again using this specification to actually generate the S function:

legacy\_code('generate\_for\_sim', IMU\_Block\_def);

This command combines two Legacy Code Tool commands:

legacy\_code('sfcn\_cmex\_generate',specs);

legacy\_code('compile',specs);

Meaning first the S function is generated and then compiled at which point it can be used in a Simulink block. If you want to automatically generate this Simulink block you can do so via:

legacy\_code('slblock\_generate', specs);

In this case this is not done because the block is already present in the AR Drone 2 library and just the S functions will be compiled when a user opens the project for the first time.

**Setting up the S function for code generation**

Now that we have an S function in Simulink which uses legacy code it is necessary to set it up for compilation to the target, in this case the AR Drone 2.0. However once the steps are followed to enable an S function for code generation it can be compiled to run on any target hardware.

First the source files that are necessary for execution on the hardware must be added to the Legacy Code specification, in the example of the IMU block this is done via:

IMU\_Block\_def.SourceFiles = {'IMU\_Navdata.c','IMU\_Navdata\_wrapper.c'};

IMU\_Block\_def.HeaderFiles = {'IMU\_Navdata.h'};

IMU\_Block\_def.SrcPaths = {'.'};

Now that the desire is to actually access the IMU the legacy code in 'IMU\_Navdata' must be included for code generation. As the script which is used for the Legacy Code Tool process is inside the same folder as the source code the SrcPath is defined as '.'.

This then enables us to create the target language compiler (tlc) file which specifies how the S function will be compiled into C code. This is done using:

legacy\_code('sfcn\_tlc\_generate', IMU\_Block\_def);

Finally you need to create an rtwmakecfg.m file which will link to the extra target specific source and header files which we defined earlier. This is the only file which does not receive a unique filename based on the S function name and as such if you wish to create multiple S functions in a single folder the rtwmakecfg should be created for all S functions at once. This is done using:

legacy\_code('rtwmakecfg\_generate', [IMU\_Block\_def; other\_def\_1; ...

other\_def\_N]);

At this point you are all set to create Simulink models that use legacy code and compile them to any target hardware such that your model can be executed on the AR Drone 2.0 independent of Simulink.

Video drivers

The AR Drone 2.0 has two video cameras. One forward facing HD (1280i \* 720) camera which can record at 30 Fps and another downward facing camera that records QVGA (320 \* 240) video at 60 Fps. The AR Drone 2 Toolbox provides Simulink blocks that can read both camera outputs and these blocks can be compiled to C code which runs on the AR Drone 2.0 such that video processing can be done on the AR Drone 2.0 independent of Simulink.

**The legacy video driver**

The TI OMAP processor used inside the AR Drone 2.0 interfaces with the cameras using the Video4Linux2 API of which [extensive documentation](https://linuxtv.org/downloads/v4l-dvb-apis/index.html) is available. A working implementation of a video driver was made in 2011 by Hugo Perquin. He kept a blog detailing his progress with hacking into the AR Drone 2.0 which you can read at [http://blog.perquin.com](http://blog.perquin.com/). This implementation was updated by Daren Lee as wrapped C code that can be implemented into an S function block using the Legacy Code Tool.

The resulting video.c file contains two of the following functions (one for each camera, front camera: N = 1, bottom camera N = 2):

* void videoInitN (void)
* void videoGrabImageN (unsigned char\* mybuf)
* void videoCloseN (void)

These functions are used in the Legacy Code Tool definition as the StartFcnSpec, OutputFcnSpec and TerminateFcnSpec.

**Initializing the camera**

The StartFcnSpec for each camera's Simulink block is *void videoInitN(void)*. This si the function that is called when compiled model is executed on the AR Drone 2.0

This function goes through a number of steps to initialize the specific camera using the Video4Linux2 arguments.

* Define the camera device location as a file descriptor (/dev/video1 for the front camera, /dev/video2 for the bottom camera).
* Define the video resolution (1280\*720 for the front camera, 320\*240 for the bottom camera)
* Set the amount of frames that should be buffered (set to 1 for lowest latency)
* Open the camera device location and query the capabilities of the camera to verify it is working (using VIDIOC\_QUERYCAP).
* Create a Video4Linux2 buffer and set the pixelformat (buffer type = V4L2\_BUF\_TYPE\_VIDEO\_CAPTURE and pixel format = V4L2\_PIX\_FMT\_UYVY). Query the camera to see if it supports this color map.
* Query whether the camera supports memory mapping (set the buffer type to V4L2\_MEMORY\_MMAP).
* Allocate memory for the specified amount of buffers and memory map them to the camera file descriptor.
* Queue the buffers (using VIDIOC\_QBUF)
* Start the camera (send VIDIOC\_STREAMON)

The initialization function is then complete allowing the grabbing of images.

**Grabbing images**

The OutputFcnSpec for each camera's Simulink block is *void videoGrabImageN(unsigned char\* mybuf)*. When specifying this function to the Legacy Code Tool the input argument is specified as *void videoGrabImageN( uint8 y1 [outputSize])* as the MATLAB uint8 specification is identical to the C unsigned char specification (0-255).

Within Simulink the sample time can be set for the camera blocks and with this frequency the videoGrabImageN function is called.

This function goes through the following steps:

* Monitor the camera file descriptor until it is ready to be accessed
* Dequeue the camera's output buffer (using VIDIOC\_DQBUF) placing it in the memory mapped buffer created during initialization.
* Memory copy the frame in the camera buffer to the 'mybuf' buffer pointed at by the argument of videoGrabImageN.

The argument outputSize passed to videoGrabImageN using the Legacy Code Tool must be a fixed number representing the amount of uint8's expected. Both cameras use sub sampling of the pixels meaning every 2 pixels are represented using four uint8 values which leads to the inputSize values:

* Front camera output size: 1843200
* Bottom camera output size: 153600

**Stopping the camera**

The TerminateFcnSpec for each camera's Simulink block is *void videoCloseN( void )* and is called when the model terminates.

This function works by:

* Unmapping all memory maps
* Closing the camera file descriptor

**Accessing the camera source blocks in Simulink**

The video.c file contains all functions necessary for using both cameras on the AR Drone 2.0 in separated function calls. Using the Legacy Code Tool two Simulink blocks are generated by the /AR\_Drone\_Target/block/video/Generate\_AR\_Drone\_Video.m file (which is automatically called by the startup script of the Simulink project if it detects the video library does not yet exist). Both blocks depend on the video.c file but can work independently or simultaneously on the AR Drone 2.0. The video source blocks are available in the AR Drone 2 library under the Video sub-library.

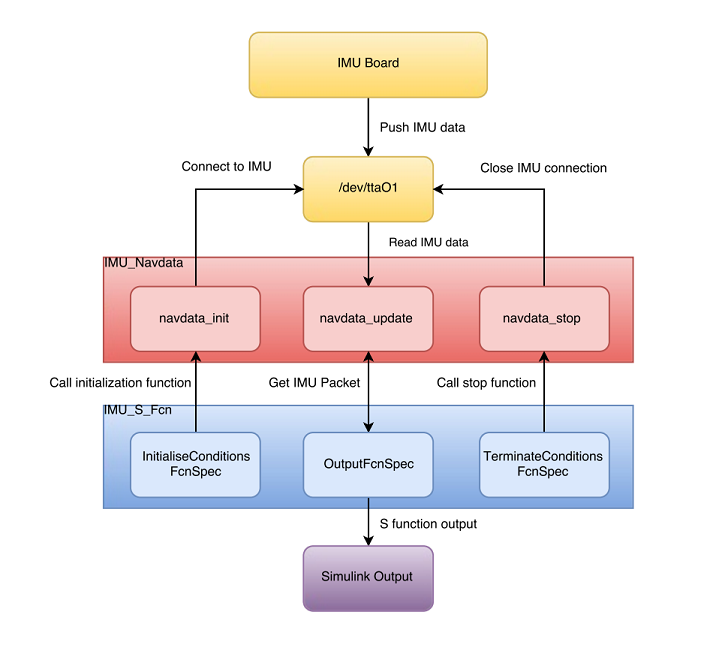
To learn more about how legacy code can be implemented please read the 'Integrating Legacy Code' section of this documentation.

Sensor drivers

All sensors on the AR Drone 2.0 can be read using legacy C code implementations created by the Paparazzi community. All but one sensor can be read from the AR Drone 2.0's IMU board, the only extra sensor is the battery sensor.

**The legacy IMU driver**

The IMU driver is based on the navdata C source files created by the Paparazzi community, which are still under development as of march 2016. The version used in this toolbox can therefor differ from the latest version from Paparazzi but the concepts listed below will remain the same with the process through which the IMU data goes from the IMU on the drone to Simulink as shown below. The Paparazzi implementation uses the files "navdata.c" and "navdata.h". The AR Toolbox versions of these files have been adjusted to work without external Paparazzi files and are placed in the AR Toolbox as "IMU\_Navdata.c" and "IMU\_Navdata.h".



**The IMU packet data structure**

The IMU outputs data in a specific format. In order to use this data format in the C code driver it is defined as the IMU\_Packet struct in IMU\_Navdata.h. This same data struct forms the output of the Simulink block and as such a Simulink bus object is created which uses IMU\_Navdata.h as the header file to define the signals by name and type.

IMU\_Packets contains the following fields:

* taille unsigned int 16 Size of the IMU Packet
* nu\_trame unsigned int 16 IMU Packet identifier number
* ax,ay,az signed int 16 Raw accelerometer data (x, y, z direction)
* vx,vy,vz signed int 16 Raw gyroscope data (x, y, z direction)
* temp\_acc unsigned int 16 Accelerometer temperature
* temp\_gyro unsigned int 16 Gyroscope temperature
* ultrasound unsigned int 16 Raw ultrasound data

Unused / unknown ultrasound parameters:

* us\_debut\_echo unsigned int 16
* us\_fin\_echo unsigned int 16
* us\_association\_echo unsigned int 16
* us\_distance\_echo unsigned int 16
* us\_cuve\_time unsigned int 16
* us\_curve\_value unsigned int 16
* us\_curve\_ref unsigned int 16
* nb\_echo unsigned int 16
* sum\_echo unsigned int 32
* gradient signed int 16
* flag\_echo\_ini unsigned int 16

Final known IMU Packet data:

* pressure signed int 16 Raw barometer data
* temperature\_pressure unsigned int 16 Barometer temperature
* mx,my,mz signed int 16 Raw magenetometer data (x, y, z direction)
* chksum unsigned int 16 Checksum for validating packet integrity.

**Retrieving IMU data**

**Initialization**

The initialization function within the driver code opens a serial connection to /dev/ttaO1 and set the UART Baud rate to 460800 bits per second and is defined as:

(int IMU\_Navdata\_init())

A separate thread is then created to handle reading of the IMU data. This initialization function is specified as the InitializeConditionsFcnSpec within the legacy code tool such that it is called before the program receives the 'Start' command from the Simulink PC. If the process is unsuccessful the legacy function will return a non zero value which will cause the wrapper function to print an error.

**Update**

With the thread set up the IMU data can be read at will using the update function:

(int IMU\_Navdata\_update())

This function calls a sub function which reads from the IMU connection:

(void Acquire\_Navdata\_Bytes())

* If the received packet starts with a NAVDATA start byte the IMU data at /dev/ttaO1 is flushed.
* The checksum is then calculated to verify the packet integrity.

TODO: FINISH

# Actuator Drivers:

# TODO: Write actuator drivers part

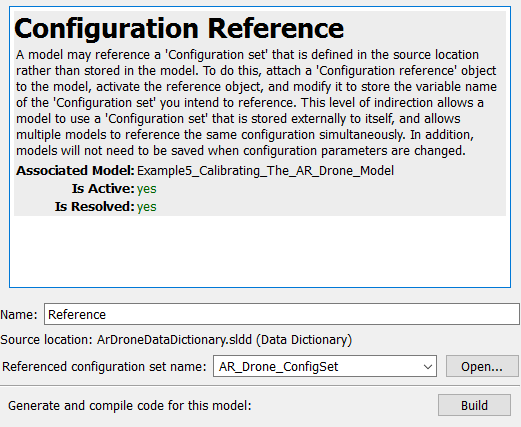
Configuration parameters

The configuration parameters determine how the Simulink Coder software generated code and builds an executable from your model. You modify the model configuration parameters for code generation in the Code Generation and Optimization panes of the Configuration Parameters dialog box. What options are available depends on the selected target, in this case the AR Drone 2.0

To open the Configuration Parameters dialog box, press control + E or click the model configuration parameters icon:

C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654835130840.png

Any model can access the configuration parameters for the AR Drone 2.0 by using a reference to the AR Drone Data Dictionary in which the configuration parameters sets are saved. This is why when you open the configuration parameters window you will be presented with the Configuration Reference menu as below:



Clicking **Open...** will bring you to the configuration parameters. You can choose two different reference configuration set names:

* **AR\_Drone\_ConfigSet:** This configuration set is configured such that the code generated by Simulink runs on the AR Drone 2.0 in external mode.
* **AR\_Drone\_ConfigSet\_NOEXT:** This configuration set is configured such that the code generated by Simulink runs on the AR Drone 2.0 in normal mode.

The most important settings on each pane of the configuration set will be explained below.

**Solver pane:**

The solver tab in the configuration parameters window has a number of key settings related to how the code is sampled:

* **Solver options:** The solver must be set to Fixed-step Type and discrete (no continuous states) as continuous states and Variable-step are not supported for AR Drone 2.0 code generation.
* **Fixed-step size (fundamental sample time):** This should be set to the highest rate needed for the implementation. As the AR Drone 2.0 sensors are updated at 200Hz setting this to 1/200 is advised
* **Tasking mode for periodic sample times:** This setting defines how models with multiple different sample times are handled with respect to models with a single sampling time. This must be set to auto such that if only one sampling time is specified the model has one step function and if there are multiple sampling times there will be multiple step functions called at different rates.
* **Higher priority value indicates higher task priority:** This must be enabled else a higher priority value would indicate a lower task priority.

**Optimization pane:**

The Optimization tab contains options for generating code that executes faster or uses less memory, some of the key settings are:

* **Default for underspecified data type:** Specify the default data type to use for inherited data types if Simulink software could not infer the data type of a signal during data type propagation. The AR Drone 2.0 supports the MATLAB default of double precision floats but this could be set to single precision for increased efficiency at the cost of accuracy.
* **Remove " ... " zero initialization:** This is left disabled for both root level I/O and internal data to ensure that the memory is not in an unknown state when the model executes.
* **Remove code from floating-point to integer conversions that wraps out-of-range values:** This is disabled as there are cases where doubles are converted to integers (for controlling the motors for example) and it is vital that these do not overflow.
* **Signals and parameters >> Default parameter behavior:** Set this to tunable if you wish to be able to change all parameters during runtime in external mode, set this to Inlined to reduce the global RAM usage and increase efficiency of the generated code. As normal mode execution is usually done without parameter tuning this should be set to 'inlined' for Normal mode.

**Hardware implementation pane:**

The hardware implementation pane is where you can select the desired external target (The AR Drone 2.0) and set some target specific parameters. Some key components are:

* **Hardware board:** This must be set to "AR Drone 2 - Thread Timer Scheduler" with **Device vendor** set to 'ARM Compatible' and **Device type** to 'ARM Cortex' as the AR Drone 2.0 uses an ARM Cortex A-8 CPU
* **Base rate task priority:** This value indicates the priority of the model within the Linux operating system on the AR Drone 2.0. The default value is 40, setting this higher will increase the priority of the generated code.
* **AR Drone IP Address:** This is the IP the AR Drone 2.0 has in the wireless network it sets up. The default IP is '192.168.1.1' but if it is changed on the AR Drone, changing the IP in this menu will make sure the build process uses the correct IP.
* **Clocking:** The AR Drone 2.0 has an 800Mhz ARM Cortex A-8

**Simulation target pane:**

This pane allows you to include custom C code. For example: if the model contains a MATLAB function block which uses a structure defined in some C code this should be included here.

* **Insert custom C code in generated: >> Header file:** Here we set '#include "IMU\_Navdata.h" ' such that we can use a MATLAB function block to process IMU\_Navdata packets.

**Code generation pane:**

The code generation pane contains settings for how the target specific executable is created. Some key settings are listed below:

* **System target file:** This must be set to the Embedded Coder target 'ert.tlc' on which the AR Drone 2.0 toolbox relies, the **Language** must be set to C.
* **Build process:** The toolchain is 'Code Sourcery' and the build configuration can be set to faster builds, faster runs or debug, depending on your need.
* **Code generation objectives:** Before enabling 'Check model before generating code' you must set the objectives against which the Code Generation Advisor will check your model. This will not impact the generated code but simply give advice on what settings you should change for your specific objectives.

**Report tab:**

In the report tab you can enable the creation of a code generation report with links between the model and the code, this is useful for if you want to see how a specific piece of the model is generated into C code. By default no report is generated.

**Interface tab:**

* **Code replacement library:** Although there is a code replacement library for the ARM Cortex A-8 used in the AR Drone 2.0, code replacement is not currently supported for the AR Drone 2.0 and should be left to 'none'.
* **Support:** The code requires floating point numbers as the default calculation is done as a double or single, absolute time to support state-flow absolute time transitions and non-finite numbers for the use of 'inf' in some initialization code. The other supports are left disabled to improve efficiency.
* **Code interface packaging:** If you have multiple copies of a sub-system you can set this to 'Reusable function' to improve efficiency by using a pointer to a dynamic function instead of statically defining it for each occurrence.
* **Data exchange:** If you are using external mode the 'Interface' parameter must be set to 'External mode' using the 'tcpip' transport layer and the MEX-file arguments: 'extModeParams'. extModeParams.m is a function which retrieves and outputs the IP address from the configset.

Data dictionaries

A data dictionary is a persistent repository of data that are relevant to your model. You can also use the base workspace to store design data that are used by your model during simulation. However, a data dictionary provides more capabilities.

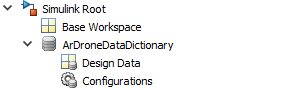
The dictionary stores design data, which define parameters and signals, and include data that define the behavior of the model. The dictionary does not store simulation data, which are inputs or outputs of model simulation.

**The AR Drone Data Dictionary**

The AR Drone toolbox uses a Data dictionary, you can view it by opening the Model Explorer from a Simulink model or the Editor, select the Model Explorer icon C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654835267390.pngfrom the toolbar or by pressing control + H. Alternatively you can manually load it from the .sldd file via:

open('ArDroneDataDictionary.sldd')

This will allow you to see the ArDroneDataDictionary contents listed in the Model Hierarchy as shown below:



which contains a number of vital components:

* **Configuration sets:** The configuration sets 'AR\_Drone\_Config' and 'AR\_Drone\_Config\_NOEXT' are both saved in the data dictionary
* **Bus signals:** some of the more complex models supplied with the AR Drone 2.0 toolbox use Bus signals to reduce the visual clutter. Some of these are defined in the data dictionary such that they can be reused between models, such as the 'IMU\_Packets' bus which is used in any model which reads the AR Drone 2.0 IMU data.
* **Calibration data:** To achieve higher performance the sensors of the AR Drone 2.0 need to be calibrated. The data dictionary contains the calibration parameters such that all models have access to them. The provided calibration interface loads and saves the calibration data from and to the data dictionary.

**Using the data dictionary with a new model**

If you make a new model that you want to run on the AR Drone 2.0 it will need to be linked to the AR Drone Data Dictionary. To do this follow these steps:

1. Open the model you wish to link to the data dictionary.
2. In the Simulink Editor, click **File** > **Model Properties** > **Link to Data Dictionary**.
3. In the **Model Properties** dialog box, set **Defined in** to **Data Dictionary** and click **Browse** to locate the ArDroneDataDictionary which is located in /AR\_Drone\_Target/
4. Click **Yes** in response to the message that explains how Simulink migrates design data stored in the base workspace. This will add that data to the data dictionary.
5. Click **OK** in the **Model Properties** dialog box. A notification appears in the Simulink Editor, reporting that your model is now linked to the data dictionary.

If no data was imported you are now done, else you need to continue with the following steps to permanently save the added data to the data dictionary:

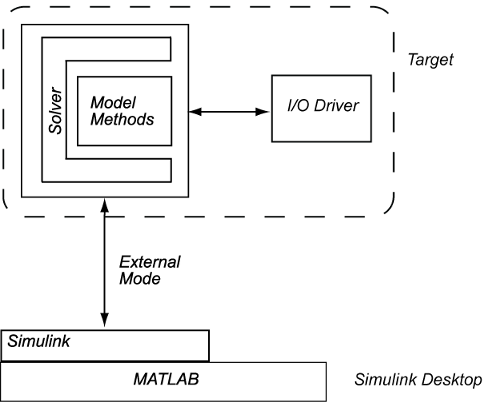
1. In the Simulink Editor, click the data dictionary badge C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654835267542.pngin the bottom left corner to open the dictionary.
2. Right-click the dictionary node and select **Save Changes**.

Simulation modes:

Simulink supports a number of different simulation modes which also change the way in which code is generated. The two modes which are interesting for the AR Drone 2.0 toolbox are:

* **External** **mode**: Run the code on an external target with the ability to read data in real time and tune parameters on the target
* **Normal** **mode**: Run standalone code on an external target with no overhead from communication layers

This is visualized as well in the following image, where the connection between the Simulink desktop and the target is only present in the case of external mode:



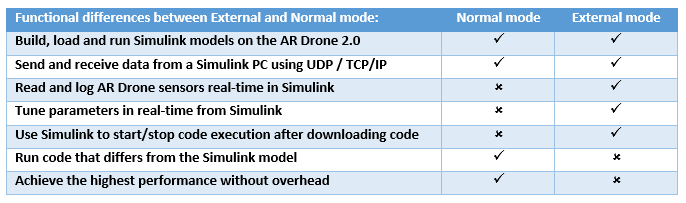
Using a combination of External and Normal mode you can iterate and optimize a design quickly and safely using the following workflow:

1. **Design** and test an implementation on a simulated plant model running in normal mode on a Simulink PC.
2. **Test** the implementation on the target hardware using External mode to monitor and tune the system.
3. **Deploy** standalone normal mode code running on the external target.

This page will document the differences between normal and external mode for code generation for the AR Drone 2.0 target.

**External versus normal mode**

External and normal mode can both be used for generating code for the AR Drone 2.0 by selecting the desired configuration set. The standard AR\_Drone\_Config is used for External mode and AR\_Drone\_Config\_NOEXT is used for normal mode. The for this target important differences can be seen in the table below:



To decide which simulation mode to use you should compare the items listed in the table with your goals. In general External mode is the simulation mode of choice when prototyping. It allows you to change the parameters in the model and then monitor the performance changes of the AR Drone 2.0 in real time using Simulink scopes. When you are satisfied with the model it can make sense to switch to normal mode. If you want to maximize the performance you can switch to normal mode for a slight performance increase as the external mode communication layer is no longer included. It is still possible to communicate with the AR Drone 2.0 running in Normal mode by implementing UDP or TCP/IP blocks in your Simulink model. If you do this you will need a second Simulink model running on your computer in Normal mode which sends or receives the data.

# Documentation

The documentation section provides extra resources and information about the information used for this toolbox and its progress.

## **Release notes**

The release notes document the changes between releases of the AR Drone 2.0 toolbox

## **External resources**

External resources documenting the hardware of the AR Drone, software implementations made by paparrazi and general quadcopter information.

## **Frequently asked questions**

Awnsers to frequently occuring questions and troubleshooting topics.

Release notes

The release notes document the changes between releases of the AR Drone 2.0 toolbox

**Version 0.2**

* The project now builds all models to the Build folder, removing out of date model referencing build failures
* Data is now stored in a data dictionary, including the configuration set and BUS objects
* A new structure of calibration parameters is defined for the accel offsets and gains, gyro offsets and magneto offsets without any internal conversions of the data
* Fixed IMU\_Packets not being loaded before S function generation in the srtartup script
* Fixed RGB color conversion in AR Drone Video Viewer and DownScaleToRGB blocks which had a blue shift

**Version 0.1**

* The toolbox is now placed inside a Simulink project
* The manual setup of compilers, loading configuration sets and generating s functions is now automated
* Example models are added
* Documentation is added
* Windows 10 is now supported by using built in MATLAB FTP functionality
* MATLAB support expanded from 2014b to 2016b
* Removed precompiled s functions
* Implemented old and new Simulink blocks in a library with user interface improvements
* Added a Simulink block to view the AR Drone cameras using external mode
* Added Simulink blocks to split and recombine large one dimension arrays into multiple packets to send via UDP
* Fixed IP adress setting to be user configurable and passed to external mode

# External resources

This section contains links to external resources documenting the hardware of the AR Drone, software implementations made by paparrazi and general quadcopter information.

## **General resources:**

[Parrot AR Drone 2.0 official specifications](http://www.parrot.com/products/ardrone-2)

The parrot website contains the specifications of the AR Drone 2.0 with details about the camera, motors, processing boards and construction

[IFIXIT AR.Drone 2.0 Repair Guide](https://www.ifixit.com/Device/Parrot_AR.Drone_2.0)

IFIXIT provides a full guide on how to repair the AR Drone 2.0. From replacing a single propellor to the entire navigation board.

## **TI Processor resources**

[TI AM35x-OMAP35x-PSP 03.00.00.05 UserGuide](http://processors.wiki.ti.com/index.php/AM35x-OMAP35x-PSP_03.00.00.05_UserGuide)

The AR Drone 2.0 uses the TI OMAP 35 as it's main processing unit. The user guide includes links to video capture and resizing drivers

[TI User guide for the TMS320C64x/C64x+ DSP](http://www.ti.com/lit/ug/spru732j/spru732j.pdf)

The AR Drone 2.0 uses the C64x digital signal processing unit.

[TI Programmers guide for the TMS320C6000](http://www.ti.com/lit/ug/spru198k/spru198k.pdf)

The AR Drone 2.0 uses the C64x digital signal processing unit which is based on the TMS320C6000

## **Sensor data sheets**

[Bosch BMA150 Digitial, triaxial accelration sensor Data sheet](http://odroid.com/dokuwiki/lib/exe/fetch.php?media=en:universal_motion_joypad:bma150.pdf)

The bosch BMA150 is the three axis accelerometer used in the IMU board of the AR Drone 2.0

[InvenSense Inc.: 3-Axis Gyroscope: IMU-3000](https://store.invensense.com/ProductDetail/IMU3000-InvenSense-Inc/415075/)

The product page of the 3 Axis gyroscope used in the AR Drone 2.0 with links to additional information.

[Prowave 400ST/R160 Air Ultrasonic Ceramic Transducers](http://www.prowave.com.tw/pdf/T400S16.PDF)

The Prowave ultrasonic transmitter and receiver are used to determine the altitude of the AR Drone 2.0.

## **Video resources**

[YCbCr Color space](https://en.wikipedia.org/wiki/YCbCr)

The AR Drone 2.0 captures video in the YCbCr (also known as YUV) colorspace. This means that for a number of applications the color space has to be converted to RGB. This Wikipedia link provides background on the YCbCr colorspace and conversions to other color spaces.

[4:2:2 subsampling explanation](https://en.wikipedia.org/wiki/Chroma_subsampling#Sampling_systems_and_ratios)

The front camera of the AR Drone 2.0 captures video at a resolution of 1280 \* 720 pixels. The horizontal resolution however is interlaced meaning that only the chroma value covers the full 1280 pixel width. Wikipedia provides insight on how this method of subsampling works.

[TI Display driver guide using Video4Linux](http://processors.wiki.ti.com/index.php/UserGuideDisplayDrivers_PSP_03.00.00.05)

The TI OMAP processor within the AR Drone 2.0 has built in support for video streaming using the Video 4 Linux 2 API. TI provides a guide on how to use it.

[Video4Linux2 Documentation](http://linuxtv.org/downloads/v4l-dvb-apis/index.html)

The Video 4 Linux 2 documentation contains a lot of information on what function calls to use to acces the camera output.

Frequently asked questions

Awnsers to frequently occuring questions and troubleshooting topics.

**What software versions are supported?**

* For the Drone firmware version 2.4.8 has been tested. On the computer side Windows 10 x64 is tested with MATLAB 2016a x64

**What C compiler is needed with MATLAB?**

* This project has been tested with the MinGW64 compiler available through the Add-On manager and with the Microsoft C++ 2015 Professional (C) compiler. You can type 'mex -setup c' to see which compiler you have currently set up to use in MATLAB.

**Which ARM compilers are supported?**

* The only tested ARM compiler is the Code Sourcery CodeBench Lite for ARM 2011.03-41. Currently Mentor no longer links to the used version on their website but it can still be downloaded using the [direct link](https://sourcery.mentor.com/sgpp/lite/arm/portal/package8738/public/arm-none-linux-gnueabi/arm-2011.03-41-arm-none-linux-gnueabi.exe) (as of May 2016).

**Do I need to modify the Parrot AR Drone 2.0 before using this package?**

* No Modifications are necessary, simply connect your PC to the WIFI network set up by the AR Drone 2.0 and you will be able to build and run the Simulink models on the AR Drone 2.0.

**Why am I getting a "ExtTargetPktPending() call failed" error?**

* Too much information is being send back to Simulink causing the drone lose connection. Reduce the amount of scopes or other real time sources or use a rate transition block to reduce their sampling rate.

**Why does initialization of the project path fail during the Simulink project startup?**

* The path is set up when the Simulink project is started and removed when the project is closed. If you close MATLAB without closing the Simulink project first the shutdown procedure is not ran meaning the path is not updated. While this will show as a failure to initialize the project path the path will be correct after opening the project.

**Will uploading any of the provided models to the Drone prohibit the factory flight options? (flying with the app?)**

* The process of uploading the custom Simulink model onto the drone kills the default Program.elf running on the drone responsible for factory flight, however when the Drone reboots it will execute it's default Program.elf again putting the drone back in factory flight mode.

**How can I acces the Drone filesystem?**

* Using your favorite Telnet protocol you can acces the Drone at it's default IP of 192.168.1.1 (e.g. type 'telnet 192.168.1.1' into a Windows command prompt). The drone uses a small version of BusyBox for shell commands limiting what is available to you. Usefull commands are 'reboot' in case the AR Drone becomes non responsive or checking out the various files on the drone with vi, such as the firmware version located in /firmware/version.txt

**How do I stop a program running on the AR Drone without using external mode?**

* Rebooting the AR Drone (either unplugging the battery typing reboot in a telnet session with the AR Drone 2.0) will kill any custom program. To specifically kill just a single program type 'top' in a telnet console to see all running programs and their PID (Process ID). Typing 'kill 'xxxx'' will kill the process with PID 'xxxx'

# AR Drone 2.0 Toolbox: Examples

The provided examples will guide you through the steps necessary to deploy code to the AR Drone 2.0

## **Example 1: Getting started**

Example 1 shows how to get a simple Simulink model running on the drone in external mode. The model will verify whether the firmware on your drone is supported.

## **Example 2: Streaming video in external mode**

This example will show you how to show the video feed from the AR Drone 2.0 within Simulink using External mode.

## **Example 3: Streaming video in Normal mode**

This example will show you how to run a model in normal and stream video data to the PC using UDP.

## **Example 4: Reading and logging sensor data**

In this example the data from the various sensors on the AR Drone 2.0 will be read into a program created from a Simulink model, allowing visualisation in real time and logging to the Simulink PC.

# Example 1: Getting Started

This example shows how to get a simple Simulink model running on the drone in external mode. The model will verify whether the firmware on your drone is supported.

## **Setting up the WIFI connection**

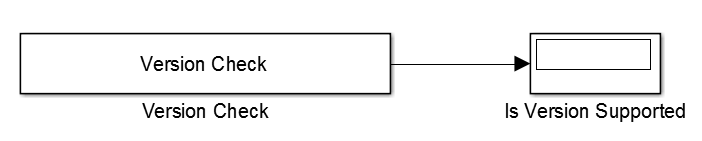
The AR Drone 2.0 creates it's own WIFI network on startup called 'ardrone2'. Connecting a PC to this unsecured network allows the uploading of code to the drone. By default the AR Drone 2.0 will give itself the IP Adress '192.168.1.1' and the PC will have '192.168.1.2'.

## **The first Simulink model**

The AR Drone 2.0 Hardware support package has been developed using multiple drones running on firmware version 2.4.8. In order to make sure that your drone is compatible with the AR Drone 2.0 Hardware support package the first model will consist of nothing more than a block which checks if the AR Drone is running this supported firmware version and then outputs that to a display. Open the model by executing the following command in your MATLAB workspace:

open('Example1\_Getting\_Started')

The model should now open and appear as below:



The Version Check block is created using the MATLAB Legacy Code Tool which can be used to integrate legacy C code into Simulink. If you wish to see what the underlying C code contains, execute the following MATLAB command in the workspace:

edit('versionCheck.c')

This C file contains three functions, versionCheckInit is called at when the model begins execution, in this case it reads the version.txt file which is by default located in the update folder on the AR Drone 2.0. The code then compares the found version with the known supported version, if they differ the static internal variable isSupported is set from 1 (supported) to 0 (not supported). The second function, versionCheckStep, is called during every execution step on the AR Drone. It sets the output of the Simulink block equal to the internal isSupported variable allowing us to see the result of the initialization function. The last function is the versionCheckClose function which is called when the model terminates. It is included for completeness of the implementation but as no memory is allocated no cleanup is necessary and the close function is kept empty.

The Simulink model is allready configured to run on the AR Drone 2.0 using external mode. Openening the Configuration Parameters can be done by pressing the C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654832742521.pngbutton in Simulink or by pressing 'ctrl + E'. This will bring you to a Configuration Reference window where the specific Configuration can be selected. For this example we will run the model in external mode allowing us to send commands and retrieve data from the drone while the model runs on the drone which is why the 'AR\_Drone\_ConfigSet' is selected, another configset could be set 'AR\_Drone\_ConfigSet\_NOEXT' if it is desired to run the model in normal mode. Without the use of external mode the model will still run on the drone however we would be unable to see the "Is Version Supported" display output.

## **Running custom firmware verification code on the AR Drone 2.0**

The following steps will take you through how to

* Make sure the model is set to external mode in the Simulink toolbar: C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654832742522.png
* Build the model using the C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654832742523.pngbutton or pressing 'ctrl + B'. The configuration set by default has the 'Build load and run' option selected meaning the software will be automatically transfered to the drone over FTP and made ready for a connection with Simulink.
* Click 'Connect to target' C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654832742524.pngin Simulink which will transfer any parameters that are tunable in real time (in this case the model has no tuneable parameters).
* Once this is done the 'Run' C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654832742525.pngbutton will turn green, press it to start the model on the drone. The "Is Version Supported" display will now display a value of either "0" (your drone does not run a supported firmware version) or "1" (your drones software version is supported).

If the display shows "0" this does not necessarily mean that your drone is incompatible with this Hardware support package but care should be taken when using the provided models as they are untested for your drones firmware version.

When you are done with this example you can stop it's execution using the 'Stop' C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654832742526.pngbutton in Simulink. Without having to reboot the drone you can run any desired model on the drone by following the steps above again.

# Example 2: Streaming video in external mode

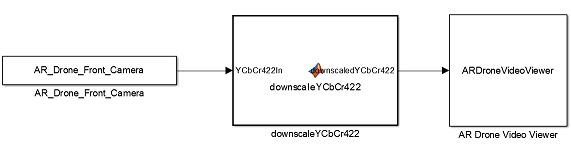
This example will show you how to show the video feed from the AR Drone 2.0 within Simulink using External mode.

## **Displaying video in External mode**

There are many different ways to display video in Simulink using various Simulink Video Display blocks for different target hardware platforms. The AR Drone 2.0 however does not support direct video output meaning that any code compiled for the AR Drone 2.0 can not assume the drone is capable of displaying video. Example 2 contains an implementation which is capable of streaming video from the drone to Simulink in external mode which you can open via:

open('AR\_Drone\_Models\Example\_Models\Example2\_Streaming\_Video\_In\_External\_Mode.slx');

This command will open the following Simulink model:



The AR\_Drone\_Front\_Camera block on the left is created using the Legacy Code Tool using the C code in video.c which contains initialization, step and termination functions for both the front and bottom camera. You can see how the video signal is retrieved exactly by executing:

edit('video.c');

The videoInit1 and videoInit2 functions query the AR Drone's front and bottom camera for video capture respectively. The correct camera parameters are set and memory is allocated to use as an image buffer. The videoGrabImage1 and videoGrabImage2 functions are used to retrieve a new frame from the AR Drone's video channel and memory copy it to an array. This array is accessed as the block output via a pointer. The videoClose1 and videoClose2 functions free up the memory which was allocted during initialization. As this example uses the AR\_Drone\_Front\_Camera block, only the videoInit1, videoGrabImage1 and videoClose1 functions are actually used.

As the drone runs in external mode the amount of bandwith necessary for the video stream should be limited to ensure sufficient bandwidth remains for external mode connectivity. The second block in the model, "downscaleYCbCr422" is responsible for limiting the bandwith by taking the original input images (Which use the YCbCr color space and 4:2:2 pixel subsampling) and down sampling by a user specified amount (in both horizontal and vertical direction). Taking the original 1280\*720 image and down sampling 8 times results in a video stream of 160x90 pixels, a reduction in bandwidth of 98.44%. Another way in wich the bandwidth is limited is by reducting the sampling time of "AR\_Drone\_Front\_Camera" to the default value of 10 Fps. Sending too much data will cause the model to be unable to receive any shutdown packets from Simulink meaning the AR Drone 2.0 needs to be reset by unplugging the battery.

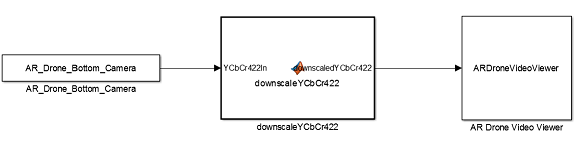
The ARDroneVideoViewer block is what enables the video stream to be displayed in Simulink. It is a level-2 MATLAB S-Function which has been configured to work as a Sim Viewing Device. This causes Embedded Coder to ignore all code within the ARDroneVideoViewer S-Function when compiling for the AR Drone 2.0 target. The signal which goes into the ARDroneVideoViewer is sent to Simulink using external mode. The transformation from YCbCr422 to RGB is then done on the Simulink PC to ease the computational burden on the drone. Displaying the video feed is simply done by giving the RGB image to the MATLAB function 'image' within the S-Function.

Building and running the model can be done in the exact same way as described in Example 1.

## **Viewing the bottom camera:**

A second model is available to view the video feed from the bottom facing camera on the AR Drone 2.0. This camera can output video with a resolution of 320x240 at 50 fps, allthough a lower Fps is used and down sampling is once again implemented to reduce the needed tcpip bandwidth. To open this second model evaluate the following MATLAB command:

open('AR\_Drone\_Models\Example\_Models\Example2\_Streaming\_Bottom\_Video\_In\_External\_Mode.slx');



The bottom camera uses a different aspect ratio (4:3) than the front camera (16:9) which is entered in the ARDroneVideoViewer block parameters diaglog box. Running the model will open a figure showing the bottom camera video feed. In the commercial application of the AR Drone 2.0 software the bottom camera is used to have a form of global positioning. If the floor beneath the drone has clearly distinguishable features/patterns the drone will be able to compensate for drift which it would otherwise not be able to detect using other sensors.

Example 3: Streaming video in Normal mode

This example will show you how to run a model in normal and stream video data to the PC using UDP.

**Running in Normal mode**

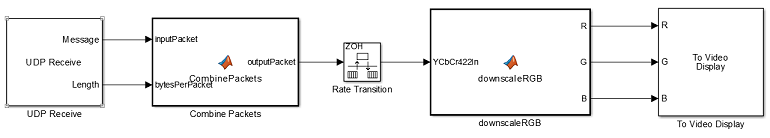
In the previous example the AR Drone 2.0 was run in External mode in order to receive the video stream in Simulink. The downside of External mode is the overhead introduced and the tcpip protocol which is not typipcally used for video streaming. This was dealt with by greatly reducing the resolution at which the video was streamed, in order to get a higher resolution video feed from the drone this example will show how to run the model in Normal mode and send the video feed to a second Simulink model using UDP.

**Receiving data using UDP**

UDP is a simpler method of sending data than tcpip and comes with it's own limitations. In the case of the AR Drone 2.0 the maximum packet size for a single UDP packet is 15000 bytes. A single uncompressed image shot at 1280 \* 720 pixels (interlaced) would therefor consist of over 122 maximum size UDP packets. While UDP is faster for this application than tcpip sending over 100 subsequent packages per frame will overburden the drone so down sampling will still be used to an extend for this example. As a single image will consist of many different packeges it is necessary to keep track of where each package belongs in the full image, to this end the full frame is devided into packages that are one byte smaller than the maximum size leaving room for a single byte to act as header. A model which can recombine these packages and show them as RGB video can be opened via :

open('Example3\_Receiving\_UDP\_Video\_In\_Normal\_Mode.slx');

Which will open up the model as seen below:



In this model we see a number of blocks:

**UDP Receive**

The UDP receive block is what allows the user to specify the network and UDP packet parameters desired for the UDP data transfer from the AR Drone 2.0 to this Simulink model. It has the following settings:

* Local IP Port: Set this to the port you wish to use for UDP trafic, this should be different from the ports used for any other protocols you wish to use in conjunction with this model (such as port 23 for telnet). The default value is 24000
* Remote IP adress: This is the IP adress of the AR Drone 2.0, by default set to '192.168.1.1'
* Receive buffer size (bytes): This buffer size is used to avoid data loss caused by buffer overflows. It is recommended to set this to no more than one packet size as that would cause addional delay.
* Maximum length for Message: This should be set equal to the size of the packet you wish to receive and no more than the amount of bytes permissible by the AR Drone 2.0 UDP packet size (15000)
* Data type for Message: This should be kept at uint8
* Output variable-size signal: This should be unchecked
* Sample time (seconds): This should be set at least as fast as the sample time with which the AR Drone 2.0 will be sending packages.

As the 'Ouput variable-size signal' option is unset the block will automatically provide an output for the constant 'length' variable which is then conveniently fed into the next block:

**Combine Packets**

The Combine Pakcets block takes packets from the UDP Receive block containing a single byte of header data and combines them into a single packet of user defined size. It has the setting:

* Full packet size: Set this to the number of bytes that the full input packet will contain. For example when using the front camera (1280 \* 720 interlaced resolution meaning 2 bytes per pixel) and down sampling 5 times (in horizontal and vertical direction meaning a reduction in data of 25x) this value should be set to 73728 bytes.

Note that UDP is subject to package loss meaning that part of an image can be lost during transfer. This is visible to the user by part of the image lagging behind untill the next frame.

**Rate Transition**

The rate transition block allows for different sampling times of the input and output ports. As multiple packets need to be processed before a single image can be reconstructed the UDP Receive side of the model needs to run at a higher sample frequency than the video display side. It has the settings:

* Ensure data integrity during data transfer: Leave this option checked to reduce the amount of packet loss.
* Ensure deterministic data transfer (maximum delay): Leave this option checked such that data is only send to the next block in line after the user defined sample time has passed.
* Initial conditions: Can be left at 0.
* Ouput port sample time options: Set to specify to allow for control of the update rate of the video feed update frequency
* Output port sample time: Set to the update speed with which the video feed is caputured on the AR Drone 2.0 (for example 1/10 if the drone captures video at 10 Fps)

**Downscale RGB**

The Downscale RGB block can handle both downscaling and colorspace conversion to RGB. In this case we only use it for the color space conversion. It has the following settings:

* Input Image width: The width of the images contained within a fully recombined packet (for example 1280/5 when using the front camera and 5x down sampling).
* Input Image height: The height of the images contained within a fully recombined packet (for example 720/5 when using the front camera and 5x down sampling).
* Down scaling factor: The desired amount of down scaling applied to the input image (Leave this to 1 as the down scaling is allready happening on the drone and no further down scaling is necessary)

**To Video Display**

This block provides a lightweight, high performance display, which accepts RGB and YcbCr formatted videos. This block contains the following settings:

* View: Select the desired window size and whether it should open at the start of the simulation according to your needs.
* Input Color Format: Chose between RGB and YCbCr 4:2:2. As the camera output is YCbCr 4:2:2 and the desired color space is RGB both of these are valid settings depending on how the data can be sensibly transformed before passing it to the Video Display.

In this case it would have been possible to feed the YCbCr image into the Video Viewer but this would have still required transferring the one-dimensional input array containing the seperate pixel values into matrixes with the chroma and hue values. As the Downscale RGB block also takes care of the transformation to seperate red, green and blue intensity matrixes the RGB input mode is used.

**Running the video receive model**

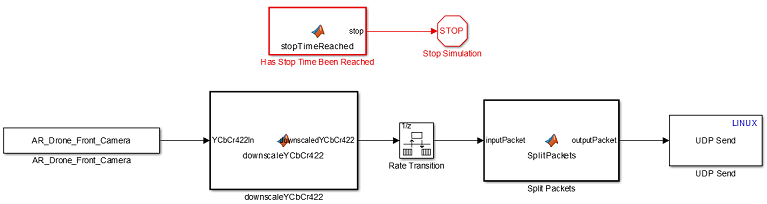
The model for receiving video is going to be running on the host PC and as such does not need to use the AR Drone configuration set and can be run using normal mode. Simply pless the 'Run' button in simulink and you are all set to receive video for which we will now set up the AR Drone 2.0.

**Streaming video using normal mode:**

A second model is provided which will be run on the AR Drone 2.0 in normal mode. This reduces the overhead from external mode and allows the implementation of UDP instead of tcpip for video transfer. Open the model by evaluating the following MATLAB command:

open('Example3\_Sending\_UDP\_Video\_In\_Normal\_Mode.slx');

The model will appear as below:



A number of blocks should be familiar from Example 2: Streaming video in external mode and from the previous model of this example. Some new blocks are introduced and will be explained below:

**Stop Time Reached**

This block will keep track of the run time of a model, if the run time is below the user specified stop time it will output '0', else it will output '1'. It has the following settings:

* Stop time [s]: This is the amount of seconds the model will run before this block outputs '1'
* Sample time [s]: This sets the sample time of this block, by setting it equal to the sample time of a different block they share a thread. As high accuracy is not required for this block the default is set to 0.1 [s], equal to the sample time of the front camera.

Note that if the AR Drone can not keep up with the model in real time due to the computational burden the 'stop' output will also be delayed.

**Stop Simulation**

When this block receives a non-zero input the simulation is stopped.

While the model will be executed in normal mode it will not be done by pressing the 'Run' button in Simulink but instead by using 'Build load and run' such that the model is running on the target hardware (AR Drone 2.0) instead of the Simulink PC. As such this is not a simulation and the simulation end time has no effect on the run time of the model, by sending a non-zero command to this block from the Stop Time Reached block the desired run time can be specified by the user.

**Split Packets**

This block splits an input array into packages of user specified maximum packet size, including a single byte as a header. It has one setting:

* Bytes per packet: The amount of bytes a single outgoing packet can contain. The default maximum value is 15000 bytes of which one byte is reserved as the header.

**UDP Send Linux**

This block will send a UDP packet to a target network adress and can be compiled to Linux targets, it has the following settings:

* Remote IP Adress: By default this is set to '192.168.1.2' as this is the default IP the PC receives when connecting to the AR Drone 2.0 wireless network.
* Remote IP Port: Set this to the port you wish to use for UDP trafic, this should be different from the ports used for any other protocols you wish to use in conjunction with this model (such as port 23 for telnet). The default value is 24000.

**Running the video send model**

As the video send model will be running on the AR Drone it requires an AR Drone compatible configset. In the case of this example a different configset is used than before as the model will need to be run on the drone in normal mode instead of external mode. This means that the source files necessary for external mode communication are not required and that the model needs to be executed on the drone without waiting for any incoming tcpip 'start' packets. The example is allready set up to use the correct configuration set which you can see by pressing control + E. The Referenced configuration set name should be set to 'AR\_Drone\_ConfigSet\_NOEXT'. Inside the Simulink window the simulation mode should allready be set to 'Normal'. With the 'Example3\_Receiving\_UDP\_Video\_In\_Normal\_Mode' model allready running you can simply press build or control + B to get the video streaming model running on the AR Drone 2.0 in normal mode. Once the process is complete you will see the video stream appear in the 'To Video Display' window of the 'Example3\_Receiving\_UDP\_Video\_In\_Normal\_Mode' model. Note that it is possible to stream video at a higher resolution or framerate due to the reduced overhead of normal mode and UDP compared to external mode and tcpip. After a certain amount of time the video stream will stop updating as the program on the AR Drone will terminate when the user specified Stop time is reached. You can then stop the 'Example3\_Receiving\_UDP\_Video\_In\_Normal\_Mode' model.

# Example 4: Reading and logging sensor data

In this example the data from the various sensors on the AR Drone 2.0 will be read into a program created from a Simulink model, allowing visualisation in real time and logging to the Simulink PC.

## **AR Drone sensor drivers in Simulink**

In order to retrieve the sensor data from the AR Drone 2.0 a driver interface is needed which can read the IMU data as it is published inside the drone filesystem at /dev/tty01. The Paparazzi open source community created a c code implementation which is able to read the data and write it into a navdata structure. A simple wrapper of the initialization, update and termination functions is therefor all that was required to be able to read the IMU data. Similairly the Paparazzi community created a c code implementation for the battery measurement which reads /dev/ic2-1, and is also wrapped to provide a Simulink block. If you wish to see how easy it is to wrap legacy c code for use in Simulink function blocks you can view the original Paparrazi and wrapper file via:

edit('BatteryMeasure.c');

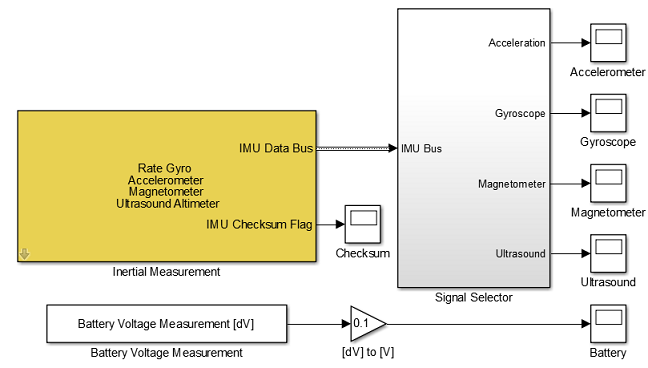
edit('BatteryMeasure\_Wrapper.c');

## **Using the driver blocks**

Both these blocks are available in the AR Drone 2 Library under the "Sensors" sub library. These library blocks are then used for the model which can visualise and save the data. To open this model evaluate the following MATLAB command:

open('Example4\_Reading\_And\_Logging\_Sensor\_Data.slx');

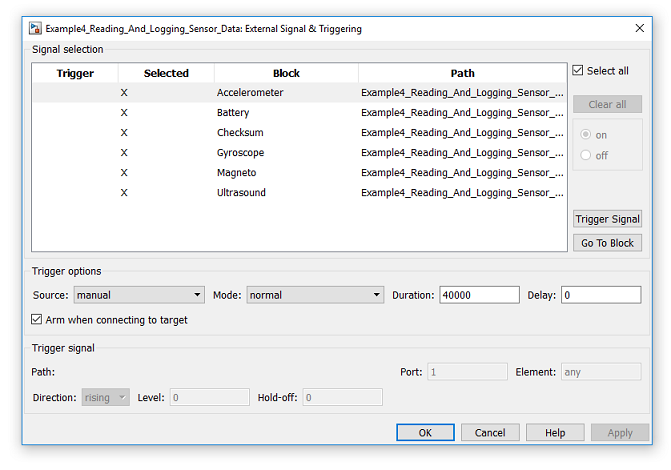
This will open the following model:



The 'IMU Data Bus' coming out of the 'Intertial Measurement' block contains the majority of the sensor information. As not all signals inside the bus are used in the Paparazii implementation nor interesting to look at it is connected to the 'Signal Selector' subsystem where a 'Bus Selector' is used to get the desired signals from the 'IMU Bus', and muxes are then used to combine the x,y and z components of the accelerometer, gyroscope and magnetometer into a single signal per sensor. The ultrasound signal and 'Battery Voltage Measurement' are single dimension signals by definition.

In order to see the signals while the model runs on the AR Drone 2.0 the model is run in external mode. This means that signals which are connected to a scope will be send to the Simulink PC and displayed. Logging the data is done by setting each scopes 'Configuration Properties >> Logging' menu such that 'Log data to workspace' is selected. The 'Limit data points to last' index is what determines how many data points are down inside the scope before clearing the scope. For example with a setting of 20000 and the 'Interial Measurement' block set to sample at 200Hz (which is the rate at which is publishes) the scope is reset every 10 seconds. When you run the model on the drone (via the same route as the previous examples in External mode) you can click on the scopes to open them and see the data as measured on the drone.

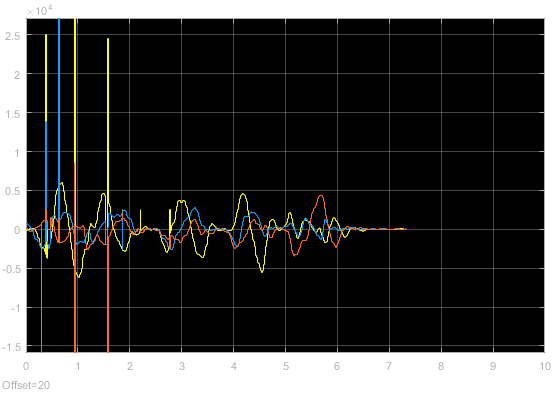
After the program is terminated the .mat files containing the sensor data will appear in the MATLAB workspace. The amount of data saved to a mat file by the scope when using External mode is goverend by a different parameter. To set this parameter open the 'External Mode Control Panel' found under the 'Code' menu entry in the Simulink Toolbar. Under 'Configuration >> Signal & Triggering ...' you can set the 'Trigger options >> duration' amount desired. As the scopes are set to save every signal as a structure with time, twice the duration should be set as twice the data is used. The image below shows this menu with the 'Duration' setting set to 40000 for all signals, meaning 100 seconds of data will be saved when sampling the IMU block at 200Hz.



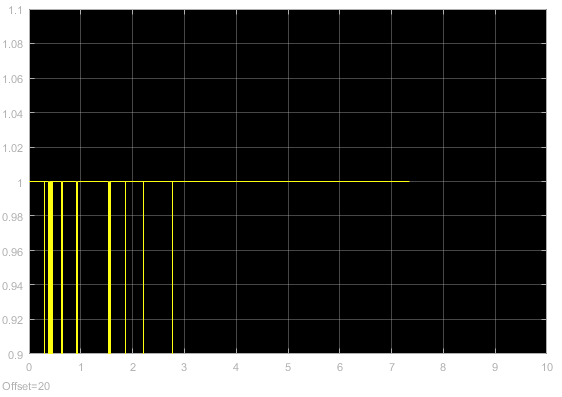
Note that the way data is stored is in intervals of the set duration, meaning that if a measurement interval has been completed all previous data will be discarded (for example using the above setting, if the model execution is stopped at 103 seconds only data from 100 till 103 seconds will be saved, with the data of the first interval from 0 to 100 seconds discarded).

## **Validating IMU data integrity**

Looking at the 'Inertial Measurement' Simulink block a second output is visible, the 'IMU Checksum Flag'. This output is '1' when the IMU data has passed a data integrity check, and set to '0' when the check fails. Knowing when the IMU data is valid is important for processing the data as large errors are introduced into the measurement when the 'IMU Checksum Flag' is '0'. This is illustrated in the figures below.



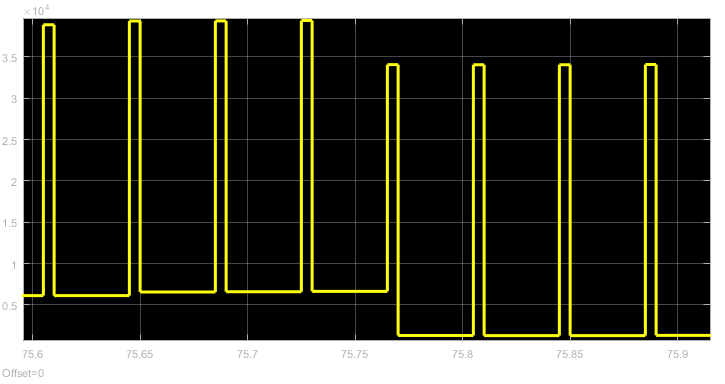
The signals in the above figure are the outputs from the three gyroscopes inside the AR Drone 2.0 IMU. The drone is moved around continuously to get significant data from the gyroscopes. A number of discontinuous peaks are visible in the data between 0 and 3 seconds, a number of which are significantly bigger than any actual data. By comparing the timestamp of these spikes to the timestamps of the 'IMU Checksum Flag' failures as seen in the figure below, the peaks are seen to be the result of corrupted navdata packages.



To this end any implementation reliant on the IMU data should be able to deal with these corrupted packages. The easiest recommended method is to hold the values of the last known good 'IMU Data Bus' sample if the current 'IMU Checksum Flag' is '0'.

## **Understanding the ultrasound data**

The ultrasound sensor positioned on the bottom of the AR Drone 2.0 faces downwards and can be used to measure the distance between the drone and a surface below it. For the best measurements the surface below the drone must be a flat horizontal surface to not scatter or misdirect the ultrasonic waves. The ultrasonic sensor polls at 25Hz and seperates the data by adding a constant value to a new measurement for the first 0.05 seconds of the sample. The figure below shows how the ultrasound measurement changes when the drone is moved such that it is positioned above a higher surface (meaning the distance from the drone to the first surface below it drops).



The value added as a measurement seperator is C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654832989695.png(or 32768). Dealing with these peaks requires detection of them (for example call anything larger than 30000 a peak) and then either holding the previous measurement, introducing a slight delay, or substracting C:\Users\sannem\AppData\Local\Temp\ConnectorClipboard2074629690423436440\image14654832989856.pngfrom that peak to reconstruct the actual data.