



Parrot
AR.Drone
THE FLYING VIDEO GAME

Developer Guide
SDK 2.0



Prepared Stephane Piskorski Nicolas Brulez Pierre Eline Frederic D'Haeyer	Title AR.Drone Developer Guide		
Approved	Date December 19, 2012	Revision SDK 2.0	File

Notations used in this document :

\$ This is a Linux shell command line (the dollar sign represents the shell prompt and should not be typed)
This is a console output (do not type this)

Here is a *file_name*.

Here is a **macro**.

iPhone® and iPod Touch® are registered trademarks of Apple Inc.

Wi-Fi® is a trademark of the Wi-Fi Alliance.

Visuals and technical specifications subject to change without notice. All Rights reserved.

The Parrot Trademarks appearing on this document are the sole and exclusive property of Parrot S.A. All the others Trademarks are the property of their respective owners.

Contents

A.R.Drone Developer Guide	1
Contents	i
I SDK documentation	1
1 Introduction	3
2 AR.Drone 2.0 Overview	5
2.1 Introduction to quadrotor UAV	5
2.2 Indoor and outdoor design configurations	7
2.3 Engines	7
2.4 LiPo batteries	7
2.5 Motion sensors	8
2.6 Assisted control of basic manoeuvres	8
2.7 Advanced manoeuvres using host tilt sensors	9
2.8 Video streaming, tags and roundel detection	10
2.9 Wifi network and connection	10
2.10 Communication services between the AR.Drone 2.0 and a client device	11
2.11 Differences between AR.Drone 2.0 and AR.Drone 1.0	11
3 AR.Drone 2.0 SDK Overview	13
3.1 Layered architecture	13
3.2 The AR.Drone 2.0 Library	14
3.3 The AR.Drone 2.0 Tool	15
3.4 The AR.Drone Engine - <i>only for Apple iOS devices</i>	16
4 ARDroneLIB and ARDroneTool functions	19
4.1 Drone control functions	19
ardrone_tool_set_ui_pad_start	19
ardrone_tool_set_ui_pad_select	19
ardrone_at_set_progress_cmd	20
ardrone_at_set_progress_cmd_with_magneto	21
5 Creating an application with ARDroneTool	23
5.1 Quick steps to create a custom AR.Drone 2.0 application	23
5.2 Customizing the client initialization	24
5.3 Using navigation data	25
5.4 Command line parsing for a particular application	27

5.5 Thread management in the application	27
5.6 Managing the video stream	28
6 AT Commands	31
6.1 AT Commands syntax	32
6.2 Commands sequencing	32
6.3 Floating-point parameters	33
6.4 AT Commands summary	34
6.5 Commands description	35
AT*REF	35
AT*PCMD / AT*PCMD_MAG	36
AT*FTRIM	37
AT*CALIB	37
AT*CONFIG	38
AT*CONFIG_IDS	38
AT*COMWDG	38
7 Incoming data streams	39
7.1 Navigation data	39
7.1.1 Navigation data stream	39
7.1.2 Initiating the reception of Navigation data	40
7.1.3 Augmented reality data stream	42
7.2 The AR.Drone 1.0 video stream	43
7.2.1 Image structure	43
7.2.2 UVLC codec overview	45
7.2.3 P264 codec overview	46
7.2.4 Specific block entropy-encoding	49
7.2.5 Transport layer	52
7.2.6 End of sequence (EOS) (22 bits)	57
7.2.7 Initiating the video stream	58
7.3 The AR.Drone 2.0 video stream	59
7.3.1 Video codecs	59
7.3.2 Video encapsulation on network	59
7.3.3 Network transmission of video stream	60
7.3.4 Latency reduction mechanism	61
7.3.5 Video record stream	61
8 Drone Configuration	63
8.1 Reading the drone configuration	63
8.1.1 With ARDroneTool	63
8.1.2 Without ARDroneTool	63
8.2 Setting the drone configuration	66
8.2.1 With ARDroneTool	66
8.2.2 From the Control Engine for iPhone	66
8.2.3 Without ARDroneTool	68
8.3 Multiconfiguration	69
8.3.1 With ARDroneTool	69
8.3.2 Multiconfiguration with Control Engine (iPhone only)	70
8.3.3 Without ARDroneTool	70
8.3.4 Common category (CAT_COMMON)	70
8.3.5 Application category (CAT_APPLI)	71
8.3.6 User category (CAT_USER) – also called "Profile" category	71

8.3.7 Session category (CAT_SESSION)	71
8.3.8 Technical details on id generation and descriptions	72
8.4 General configuration	73
GENERAL:num_version_config	73
GENERAL:num_version_mb	73
GENERAL:num_version_soft	73
GENERAL:drone_serial	73
GENERAL:soft_build_date	73
GENERAL:motor1_soft	73
GENERAL:motor1_hard	73
GENERAL:motor1_supplier	73
GENERAL:ardrone_name	73
GENERAL:flying_time	74
GENERAL:navdata_demo	74
GENERAL:navdata_options	74
GENERAL:com_watchdog	74
GENERAL:video_enable	75
GENERAL:vision_enable	75
GENERAL:vbat_min	75
8.5 Control configuration	76
CONTROL:accs_offset	76
CONTROL:accs_gains	76
CONTROL:gyros_offset	76
CONTROL:gyros_gains	76
CONTROL:gyros110_offset	76
CONTROL:gyros110_gains	76
CONTROL:magneto_offset	76
CONTROL:magneto_radius	76
CONTROL:gyro_offset_thr_x	76
CONTROL:pwm_ref_gyros	76
CONTROL:osctun_value	77
CONTROL:osctun_test	77
CONTROL:control_level	77
CONTROL:euler_angle_max	77
CONTROL:altitude_max	78
CONTROL:altitude_min	78
CONTROL:control_iphone_tilt	78
CONTROL:control_vz_max	78
CONTROL:control_yaw	79
CONTROL:outdoor	79
CONTROL:flight_without_shell	79
CONTROL:autonomous_flight	79
CONTROL:manual_trim	80
CONTROL:indoor_euler_angle_max	80
CONTROL:indoor_control_vz_max	80
CONTROL:indoor_control_yaw	80
CONTROL:outdoor_euler_angle_max	80
CONTROL:outdoor_control_vz_max	80
CONTROL:outdoor_control_yaw	80
CONTROL:flying_mode	80
CONTROL:hovering_range	81

CONTROL:flight_anim	81
8.6 Network configuration	82
NETWORK:ssid_single_player	82
NETWORK:ssid_multi_player	82
NETWORK:wifi_mode	82
NETWORK:wifi_rate	82
NETWORK:owner_mac	82
8.7 Nav-board configuration	83
PIC:ultrasound_freq	83
PIC:ultrasound_watchdog	83
PIC:pic_version	83
8.8 Video configuration	84
VIDEO:camif_fps	84
VIDEO:codec_fps	84
VIDEO:camif_buffers	84
VIDEO:num_trackers	84
VIDEO:video_codec	84
VIDEO:video_slices	85
VIDEO:video_live_socket	85
VIDEO:video_storage_space	85
VIDEO:bitrate	85
VIDEO:max_bitrate	85
VIDEO:bitrate_control_mode	86
VIDEO:bitrate_storage	86
VIDEO:videol_channel	86
VIDEO:video_on_usb	86
VIDEO:video_file_index	87
8.9 Leds configuration	88
LEDS:leds_anim	88
8.10 Detection configuration	89
DETECT:enemy_colors	89
DETECT:groundstripe_colors	89
DETECT:enemy_without_shell	89
DETECT:detect_type	89
DETECT:detections_select_h	90
DETECT:detections_select_v_hsync	90
DETECT:detections_select_v	90
8.11 SYSLOG section	92
8.12 USERBOX section	92
USERBOX:userbox_cmd	92
8.13 GPS section	93
GPS:latitude	93
GPS:longitude	93
GPS:altitude	93
8.14 CUSTOM section - Multiconfig support	94
CUSTOM:application_id	94
CUSTOM:application_desc	94
CUSTOM:profile_id	94
CUSTOM:profile_desc	94
CUSTOM:session_id	94
CUSTOM:session_desc	94

II Tutorials	95
9 Building the iOS Example	97
10 Building the Linux Examples	99
10.1 Set up your development environment	100
10.2 Compile linux examples	100
10.3 Run the SDK Demo program	101
10.4 Run the Video Demo program	102
10.5 Run the <i>Navigation</i> program	103
11 Android example	107
11.1 Set up your development environment	107
11.2 Building and installing the Android example	109
11.3 Modifying the Android example source code	109
11.3.1 Modifying the ARDroneLib part	109
11.3.2 Modifying the JNI part	109
11.3.3 Modifying the UI part	110

Part I

SDK documentation



Introduction

Welcome to the AR.Drone 2.0 Software Development Kit !

The AR.Drone 2.0 product and the provided host interface example have innovative and exciting features such as:

- intuitive touch and tilt flight controls
- live video streaming
- video recording and photo shooting
- updated Euler angles of the AR Drone
- embedded tag detection for augmented reality games

The AR.Drone 2.0 SDK allows third party developers to develop and distribute new games based on AR.Drone 2.0 product for Wifi, motion sensing mobile devices like the Apple iPhone, iPad, iPod touch, personal computers or Android devices.

To download the AR.Drone 2.0 SDK, third party developers will have to register and accept the AR.Drone 2.0 SDK License Agreement terms and conditions. Upon final approval from Parrot, they will have access to the AR.Drone 2.0 SDK download web page.

This SDK includes :

- this document explaining how to use the SDK, and describes the drone communications protocols;
- the AR.Drone 2.0 Library (**ARDroneLIB**), which provides the APIs needed to easily communicate and configure an AR.Drone 2.0 product;
- the AR.Drone 2.0 Tool (**ARDroneTool**) library, which provides a fully functionnal drone client where developers only have to insert their custom application specific code;
- the AR.Drone 2.0 Control Engine library which provides an intuitive control interface developed by Parrot for remotely controlling the AR.Drone 2.0 product from an iOS Device;
- several code examples that show how to control the drone from a Linux personal computer.
- source code for iOS and Android¹ AR.FreeFlight 2.0 applications

Where should I start ?

Please first read chapter [2](#) to get an overview of the drone abilities and a bit of vocabulary.

You then have the choice between :

- using the provided library [5](#) and modifying the provided examples ([9](#), [10](#)) to suit your needs
- trying to write your own software from scratch by following the specifications given in [6](#) and [7](#).

¹May not be available as part of the first release of the AR.Drone 2.0 SDK



2

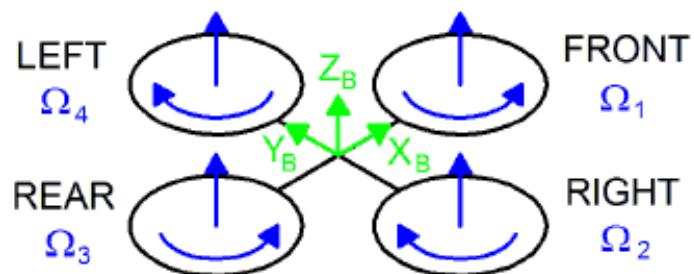
AR.Drone 2.0 Overview



2.1 Introduction to quadrotor UAV

AR.Drone 2.0 is a quadrotor. The mechanical structure comprises four rotors attached to the four ends of a crossing to which the battery and the RF hardware are attached.

Each pair of opposite rotors is turning the same way. One pair is turning clockwise and the other anti-clockwise.



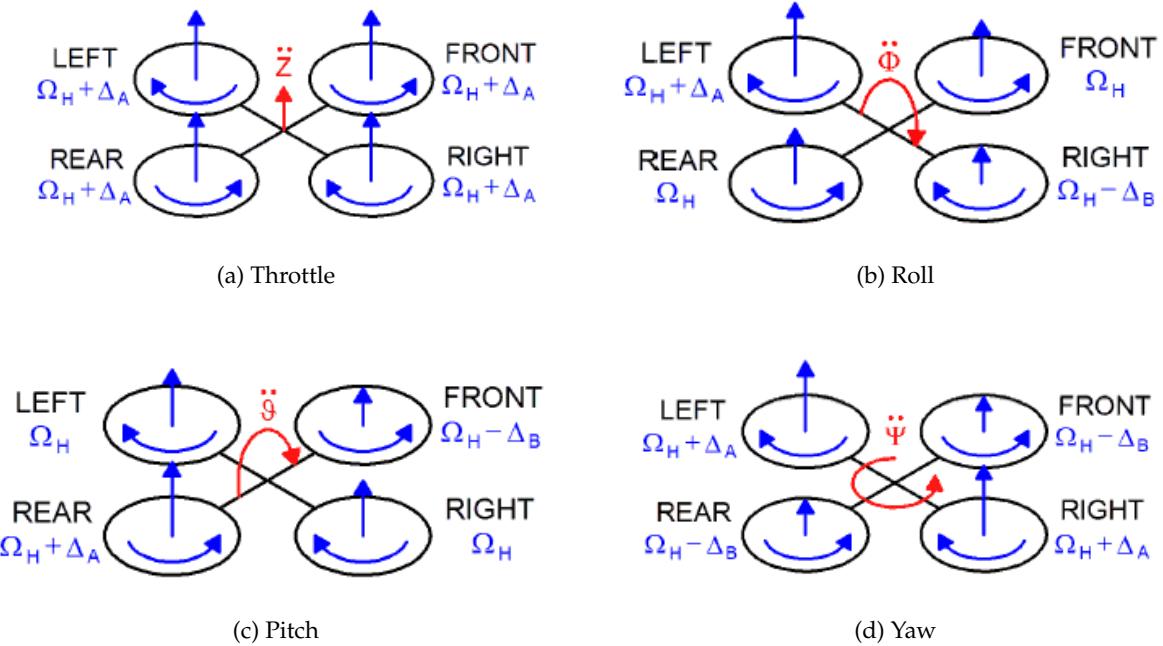
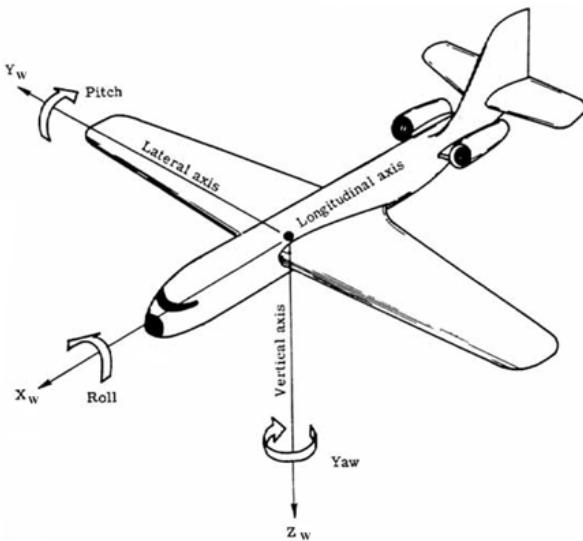


Figure 2.1: Drone movements

Manoeuvres are obtained by changing pitch, roll and yaw angles of the AR.Drone 2.0 .



Varying left and right rotors speeds the opposite way yields roll movement. This allows to go forth and back.

Varying front and rear rotors speeds the opposite way yields pitch movement.

Varying each rotor pair speed the opposite way yields yaw movement. This allows turning left and right.



Figure 2.2: Drone hulls

2.2 Indoor and outdoor design configurations

When flying outdoor the AR.Drone 2.0 can be set in a light and low wind drag configuration ([2.2b](#)). Flying indoor requires the drone to be protected by external bumpers ([2.2a](#)).

When flying indoor, tags can be added on the external hull to allow several drones to easily detect each others via their cameras.

2.3 Engines

The AR.Drone 2.0 is powered with brushless engines with three phases current controlled by a micro-controller

The AR.Drone 2.0 automatically detects the type of engines that are plugged and automatically adjusts engine controls. The AR.Drone 2.0 detects if all the engines are turning or are stopped. In case a rotating propeller encounters any obstacle, the AR.Drone 2.0 detects if any of the propeller is blocked and in such case stops all engines immediately. This protection system prevents repeated shocks.

2.4 LiPo batteries

The AR.Drone 2.0 uses a charged 1000mAh, 11.1V LiPo batteries to fly. While flying the battery voltage decreases from full charge (12.5 Volts) to low charge (9 Volts). The AR.Drone 2.0 monitors battery voltage and converts this voltage into a battery life percentage (100% if battery is full, 0% if battery is low). When the drone detects a low battery voltage, it first sends a warning message to the user, then automatically lands. If the voltage reaches a critical level, the whole system is shut down to prevent any unexpected behaviour.

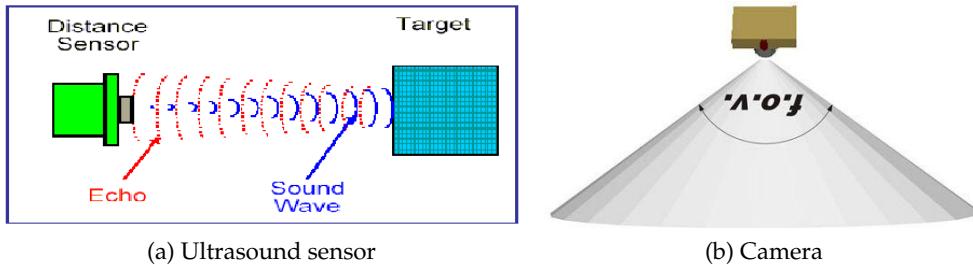


Figure 2.3: Drone Sensors

2.5 Motion sensors

The AR.Drone has many motion sensors. They are located below the central hull.

The AR.Drone 1.0 features a 6 DOF, MEMS-based, miniaturized inertial measurement unit. It provides the software with pitch, roll and yaw measurements.

Inertial measurements are used for automatic pitch, roll and yaw stabilization and assisted tilting control. They are needed for generating realistic augmented reality effects.

An ultrasound telemeter provides altitude measures for automatic altitude stabilization and assisted vertical speed control.

A camera aiming towards the ground provides with ground speed measures for automatic hovering and trimming.

The AR.Drone 2.0 Add 3 DOF to the IMU with a 3 axis magnetometer (mandatory for Absolute Control mode). It also adds a pressure sensor to allow altitude measurements at any height.

2.6 Assisted control of basic manoeuvres

Usually quadrotor remote controls feature levers and trims for controlling UAV pitch, roll, yaw and throttle. Basic manoeuvres include take-off, trimming, hovering with constant altitude, and landing. It generally takes hours to a beginner and many UAV crashes before executing safely these basic manoeuvres.

Thanks to the AR.Drone 2.0 onboard sensors take-off, hovering, trimming and landing are now completely automatic and all manoeuvres are completely assisted.

User interface for basics controls on host can now be greatly simplified :

- When landed push *take-off* button to automatically start engines, take-off and hover at a pre-determined altitude.
- When flying push *landing* button to automatically land and stop engines.
- Press *turn left* button to turn the AR Drone automatically to the left at a predetermined speed. Otherwise the AR Drone automatically keeps the same orientation.
- Press *turn right* button to turn the AR Drone automatically to the right. Otherwise the AR Drone automatically keeps the same orientation.
- Push *up* button to go upward automatically at a predetermined speed. Otherwise the AR Drone automatically stays at the same altitude.
- Push *down* to go downward automatically at a predetermined speed. Otherwise the AR Drone automatically stays at the same altitude.

A number of flight control parameters can be tuned:

- altitude limit
- yaw speed limit
- vertical speed limit
- AR.Drone 2.0 tilt angle limit
- host tilt angle limit

2.7 Advanced manoeuvres using host tilt sensors

Many hosts now include tilt motion sensors. Their output values can be sent to the AR.Drone 2.0 as the AR.Drone 2.0 tilting commands.

One *tilting* button on the host activates the sending of tilt sensor values to the AR.Drone 2.0 . Otherwise hovering is a default command when the user does not input any manoeuvre command. This dramatically simplifies the AR.Drone 2.0 control by the user.

The host tilt angle limit and trim parameters can be tuned.

2.8 Video streaming, tags and roundel detection

The frontal camera is a CMOS sensor with a 90 degrees angle lens.

The AR.Drone automatically encodes and streams the incoming images to the host device. The AR.Drone 1.0 use QCIF (176x144, bottom facing camera) or QVGA (320x240, front facing camera) image resolutions. The video stream frame rate is set to 15 FPS. The AR.Drone 2.0 use 360p (640x360) or 720p (1280x720) image resolutions for both cameras (with upscaling from bottom facing camera). The video stream frame rate can be adjusted between 15 and 30 FPS.

The AR.Drone provides detection of three different tags types shown below. After update of the AR.Drone 1.0 , cross detections with AR.Drone 2.0 is possible.

Users can download a printable version of the Oriented Roundel on [Parrot website](#).

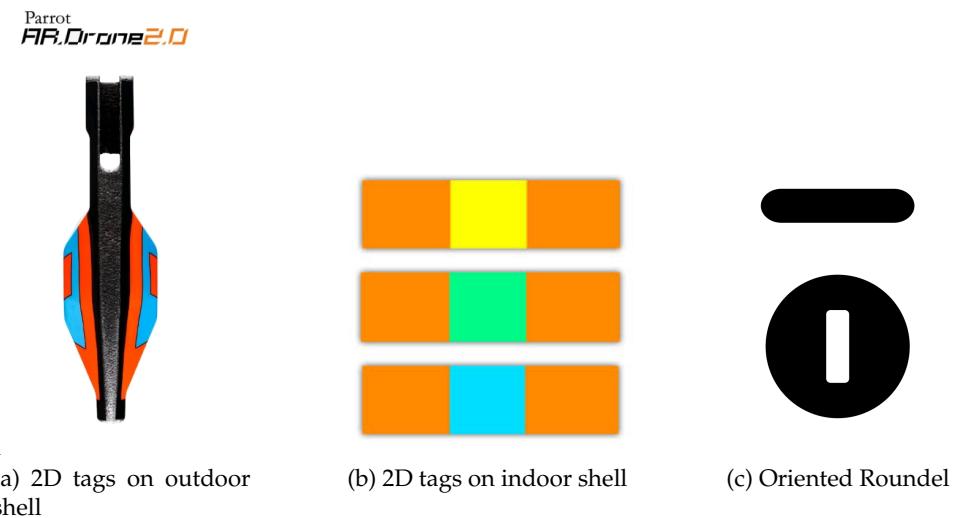


Figure 2.4: Detection tags

2.9 Wifi network and connection

The AR.Drone 2.0 can be controlled from any client device supporting Wifi. The following process is followed :

1. the AR.Drone creates a WIFI network with an ESSID usually called ***ardrone2_xxx*** (***ardrone_xxx*** for AR.Drone 1.0) and self allocates a free, odd IP address (typically 192.168.1.1).
2. the user connects the client device to this ESSID network.
3. the client device requests an IP address from the drone DHCP server.
4. the AR.Drone DHCP server grants the client with an IP address which is :
 - the drone own IP address plus 1 (for AR.Drone 1.0 prior to version 1.1.3)

- the drone own IP address plus a number between 1 and 4 (for AR.Drone 2.0 and AR.Drone 1.0 after 1.1.3 version)
5. the client device can start sending requests the AR.Drone IP address and its services ports.

2.10 Communication services between the AR.Drone 2.0 and a client device

Controlling the AR.Drone is done through 3 main communication services.

Controlling and configuring the drone is done by sending *AT commands* on UDP port 5556. The transmission latency of the control commands is critical to the user experience. Those commands are to be sent on a regular basis (usually 30 times per second). The list of available commands and their syntax is discussed in chapter [6](#).

Information about the drone (like its status, its position, speed, engine rotation speed, etc.), called *navdata*, are sent by the drone to its client on UDP port 5554. These *navdata* also include tags detection information that can be used to create augmented reality games. They are sent approximatively 15 times per second in demo mode, and 200 times per second in full (debug) mode.

A video stream is sent by the AR.Drone to the client device on port 5555 (UDP for AR.Drone 1.0 , TCP for AR.Drone 2.0). Images from this video stream can be decoded using the codec included in this SDK. Its encoding format is discussed in section [7.2](#).

A fourth communication channel, called *control port*, can be established on TCP port 5559 to transfer critical data, by opposition to the other data that can be lost with no dangerous effect. It is used to retrieve configuration data, and to acknowledge important information such as the sending of configuration information.

2.11 Differences between AR.Drone 2.0 and AR.Drone 1.0

Please note that this list is not an exhaustive list, but rather a reminder for developpers that want to support both generations of AR.Drone

Sensors

AR.Drone 2.0 includes new hardware sensors : a 3 axis magnetometer, and a pressure sensor. Other sensors were also changed, as the navigation boards are not the same. These magnetometer is mandatory for the Absolute Control feature. The pressure sensor allows the AR.Drone 2.0 to know its height regardless of the ultrasound performance (after 6 meters, the ultrasound can't measure the height)

Cameras

AR.Drone 2.0 use a HD (720p-30fps) front facing camera. This camera can be configured to stream both 360p (640*360) or 720p (1280*720) images.

AR.Drone 1.0 use a VGA (640*480) camera, which can only stream QVGA (320*240) pictures. Full resolution pictures are only available to detection algorithms, and photo shooting.

AR.Drone 2.0 bottom facing camera is a QVGA (320*240) 60fps camera. This camera pictures will be upscaled to 360p or 720p for video streaming.

AR.Drone 1.0 use a QCIF (176*144) 60fps camera, which is streamed at full resolution.

USB port

AR.Drone 2.0 has a master USB port, with a standard USB-A connector. This USB port is currently used for USB Key video recording.

Please note that the AR.Drone 2.0 only supports USB keys with a grounded USB connector casing, and formatted in FAT32 file format.



AR.Drone 2.0 SDK Overview

This SDK allows you to easily write your own applications to remotely control the drone :

- from any Linux personal computer with Wifi connectivity;
- from an Apple iOS device;
- from an Android device¹.

It also allows you, with a bit more effort, to remotely control the drone from any programmable device with a Wifi network card and a TCP/UDP/IP stack - for devices which are not supported by Parrot, a complete description of the communication protocol used by the drone is given in this document;

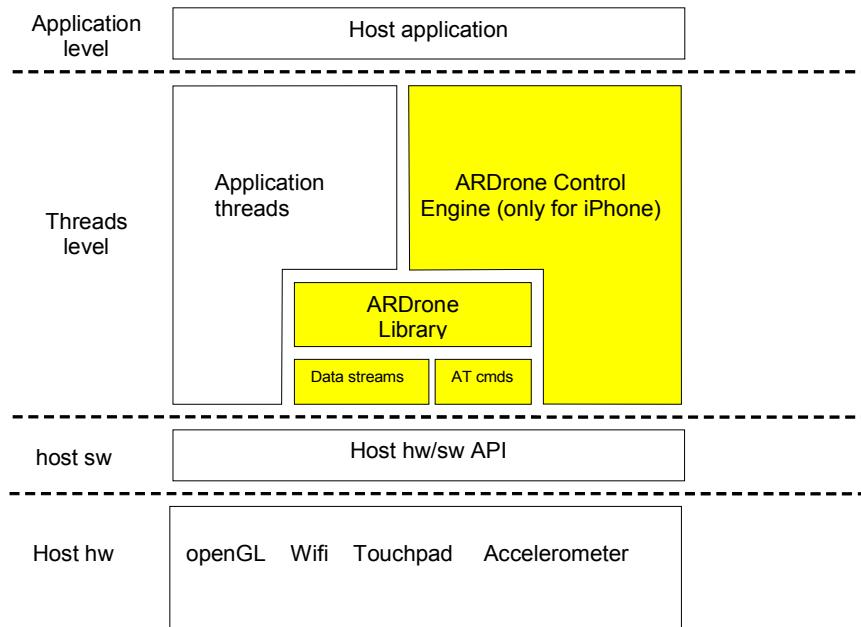
However, this SDK does NOT support :

- rewriting your own embedded software - no direct access to the drone hardware (sensors, engines) is allowed.

3.1 Layered architecture

Here is an overview of the layered architecture of a host application built upon the AR.Drone 2.0 SDK.

¹May not be available as part of the first release of the AR.Drone 2.0 SDK



3.2 The AR.Drone 2.0 Library

The AR.Drone 2.0 Library is currently provided as an open-source library with high level APIs to access the drone.

Let's review its content :

- **SOFT** : the drone-specific code, including :
 - **COMMON** : header (.h) files describing the communication structures used by the drone (make sure you pack the C structures when compiling them)
 - **Lib/ardrone_tool** : a set of tools to easily manage the drone, like an AT command sending loop and thread, a navdata receiving thread, a ready to use video pipeline, and a ready to use **main** function
 - **Lib/utils** : a set of utilities for writing applications around the AR.Drone
- **VLIB** : the AR.Drone 1.0 video processing library. It contains the functions to receive and decode the video stream
- **FFMPEG** : a complete snapshot of FFMPEG library, with associated build scripts for the AR.Drone 2.0 applications
- **ITTIAM** : a prebuilt, highly optimized (ARMv7 + NEON), video decoding library for iOS and Android applications
- **VPSDK** : a set of general purpose libraries, including
 - **VPSTAGES** : video processing pieces, which you can assemble to build a video processing pipeline

- **VPOS** : multiplatform (Linux/Windows/Parrot proprietary platforms) wrappers for system-level functions (memory allocation, thread management, etc.)
- **VPCOM** : multiplatform wrappers for communication functions (over Wifi, Bluetooth, etc.)
- **VPAPI** : helpers to manage video pipelines and threads

Let's now detail the **ARDroneTool** part :

- *ardrone_tool.c* : contains a ready-to-use *ardrone_tool_main* C function which initialises the Wifi network and initiates all the communications with the drone
- **UI** : contains a ready-to-use gamepad management code
- **AT** : contains all the functions you can call to actually control the drone. Most of them directly refer to an AT command which is then automatically built with the right syntax and sequencing number, and forwarded to the AT management thread.
- **NAVDATA** : contains a ready-to-use Navdata receiving and decoding system
- **ACADEMY** : contains a ready-to-use downloading and uploading system for the upcoming AR.Drone Academy . The downloading system also manage the AR.Drone photo shooting.
- **VIDEO** : contains all the function related to video receiving, decoding and recording, for both AR.Drone 1.0 and AR.Drone 2.0 .
- **CONTROL** : contains a ready-to-use AR.Drone configuration management tool

3.3 The AR.Drone 2.0 Tool

Part of the AR.Drone 2.0 Library is the **ARDroneTool** .

The **ARDroneTool** is a library which implements in an efficient way the four services described in section [2.10](#).

In particular, it provides :

- an AT command management thread, which collects commands sent by all the other threads, and send them in an ordered manner with correct [sequence numbers](#)
- a *navdata* management thread which automatically receives the *navdata* stream, decodes it, and provides the client application with ready-to-use navigation data through a callback function
- a video management thread, which automatically receives the video stream and provides the client application with ready-to-use video data through a callback function. This thread also manages AR.Drone 1.0 recording
- a video recorder thread (only for AR.Drone 2.0), which manages the recording of the HD Stream, and the .mp4/.mov encapsulation.

- a *control* thread which handles requests from other threads for sending reliable commands from the drone, and automatically checks for the drone acknowledgements.
- a set of threads for AR.Drone Academy , which automatically receives the photo shooting (.jpg format) by ftp protocol. A thread manage userbox binary data receiving and uploading to the AR.Drone Academy server².

All those threads take care of connecting to the drone at their creation, and do so by using the *vp_com* library which takes charge of reconnecting to the drone when necessary.

These threads, and the required initialization, are created and managed by a *ardrone_tool_main* function, also provided by the **ARDroneTool** in the *ardrone_tool.c* file.

All a programmer has to do is then fill the desired callback functions with some application specific code. Navdata can be processed as described in section [5.3](#). The video frames can be retrieved as mentionned in [5.6](#).

3.4 The AR.Drone Engine - *only for Apple iOS devices*

The AR.Drone Engine (*ControlEngine* folder) provides all the AR.Drone 2.0 applications for iOS Device with common methods for managing the drone, managing touch/tilt controls and special events on the iOS Device , and provide decoded video datas for display.

It is meant to be a common base for all iOS applications, in order to provide a common drone API and user interface (common controls, setting menus, etc.). The Control Engine API is the only interface to the drone from the iOS application. It is the Control Engine task to acces the **ARDroneLIB** .

The AR.Drone Engine automatically opens, receives and decodes video stream coming from AR.Drone . The AR.Drone Engine does not render the decoded frames on screen. The application is in charge of displaying frames.

The AR.Drone Engine also provide an HUD containing different input buttons and informations about the AR.Drone state:

The input buttons are :

- Back : return to the application home screen, and pause the **ARDroneTool** . (can be disabled in HUD configuration)
- Settings : display the AR.Drone settings screen.
- Emergency : Stop the AR.Drone engines, regardless of its current state.
- Switch : Switch between the different cameras of the AR.Drone . (can be disabled in HUD configuration)
- Record : Start/stop the video recording. (can be disabled in HUD configuration)
- Screenshot : Take a photo from the AR.Drone front facing camera. (can be disabled in HUD configuration)
- Take-Off / Landing : Take off AR.Drone (if landed) or Land AR.Drone (if flying). Take-Off button also reset emergency state if needed.

²The server won't be available until the release of AR.FreeFlight 2.1

The AR.Drone state informations are:

- Battery level : Provide information about the AR.Drone battery level. (numeric percentage can be disabled in HUD configuration)
- Wifi indicator : Provide a quality estimation of the WiFi link with the AR.Drone
- USB indicator (only on AR.Drone 2.0) : If an USB key is plugged, display the remaining record time on the USB drive.
- Warning message label : Display various warning/emergency messages described below.
- PopUp view : display various temporary messages to the user (can be closed).

Possible warning messages are :

- *CONTROL LINK NOT AVAILABLE* : WiFi connection lost.
- *START NOT RECEIVED* : AR.Drone didn't receive the take off command.
- *CUT OUT EMERGENCY* : One or more motor(s) was stopped by environment.
- *MOTORS EMERGENCY* : One or more motor(s) is not responding.
- *CAMERA EMERGENCY* : One or more camera(s) is not responding.
- *PIC WATCHDOG EMERGENCY* : Navboard is not responding.
- *PIC VERSION EMERGENCY* : Navboard was not correctly updated.
- *TOO MUCH ANGLE EMERGENCY* : AR.Drone euler angles went too high. Motors shutdown to prevent bad behaviors.
- *BATTERY LOW EMERGENCY* : Battery too low (automatic landing).
- *USER EMERGENCY* : User pressed the Emergency button.
- *ULTRASOUND EMERGENCY* : Ultrasound sensor is not responding.
- *UNKNOWN EMERGENCY* : The reason of the emergency state is not known by the application (should not happen if the application is up-to-date)
- *VIDEO CONNECTION ALERT* : The video stream can't be retrieved.
- *BATTERY LOW ALERT* : Battery is too low to take-off again. No changes if already flying.
- *ULTRASOUND ALERT* : Ultrasound sensor can't determine AR.Drone altitude.
- *VISION ALERT* : No speed estimation from bottom facing camera.



4

ARDroneLIB and ARDroneTool functions

Here are discussed the functions provided by the **ARDroneLIB** to manage and control the drone.

Important

Those functions are meant to be used along with the whole **ARDroneLIB** and **ARDroneTool** framework.

You can use them when building your own application as described in chapter 5 or when modifying the examples.

They cannot be used when writing an application from scratch; you will then have to reimplement your own framework by following the specifications of the *AT commands* (chapter 6), navigation data (section 7.1), and video stream (section 7.2).

Most of them are declared in file *ardrone_api.h* of the SDK.

4.1 Drone control functions

`ardrone_tool_set_ui_pad_start`

Summary : Take off - Land

Corresponding AT command : [AT*REF](#)

Args : (int *value* : take off flag)

Description :

Makes the drone take-off (if *value*=1) or land (if *value*=0).

When entering an emergency mode, the client program should call this function with a zero argument to prevent the drone from taking-off once the emergency has finished.

ardrone_tool_set_ui_pad_select

Summary : Send emergency signal / recover from emergency

Corresponding AT command : [AT*REF](#)

Args : (**int value** : emergency flag)

Description :

When the drone is in a normal flying or ready-to-fly state, use this command with value=1 to start sending an emergency signal to the drone, i.e. make it stop its engines and fall.

When the drone is in an emergency state, use this command with value=1 to make the drone try to resume to a normal state.

Once you sent the emergency signal, you must check the drone state in the *navdata* and wait until its state is actually changed. You can then call this command with value=0 to stop sending the emergency signal.

ardrone_at_set_progress_cmd

Summary : Moves the drone

Corresponding AT command : [AT*PCMD](#)

Args : (**int flags** : Flag enabling the use of progressive commands and the new Combined Yaw control mode
float phi : Left/right angle $\in [-1.0; +1.0]$
float theta : Front/back angle $\in [-1.0; +1.0]$
float gaz : Vertical speed $\in [-1.0; +1.0]$
float yaw : Angular speed $\in [-1.0; +1.0]$)

Description :

This function makes the drone move in the air. It has no effect when the drone lies on the ground.

The drone is controlled by giving as a command a set of four parameters :

- a left/right bending angle, with 0 being the horizontal plane, and negative values bending leftward
- a front/back bending angle, with 0 being the horizontal plane, and negative values bending frontward
- a vertical speed
- an angular speed around the yaw-axis

In order to allow the user to choose between smooth or dynamic moves, the arguments of this function are not directly the control parameters values, but a percentage of the maximum corresponding values as set in the drone parameters. All parameters must thus be floating-point values between -1.0 and 1.0.

The flags argument is a bitfields containing the following informations :

- Bit 0 : when Bit0=0 the drone will enter the *hovering* mode, i.e. try to stay on top of a fixed point on the ground, else it will follow the commands set as parameters.
- Bit 1 : when Bit1=1 AND CONTROL:control_level configuration Bit1=1, the new Combined Yaw mode is activated. This mode includes a complex hybridation of the phi parameter to generate complete turns (phi+yaw).

ardrone_at_set_progress_cmd_with_magneto

Summary : Moves the drone (allow Absolute Control mode)

Corresponding AT command : [AT*PCMD_MAG](#)

Args : (int <i>flags</i> :	Flag enabling the use of progressive commands and the new Combined Yaw control mode
	float <i>phi</i> :	Left/right angle $\in [-1.0; +1.0]$
	float <i>theta</i> :	Front/back angle $\in [-1.0; +1.0]$
	float <i>gaz</i> :	Vertical speed $\in [-1.0; +1.0]$
	float <i>yaw</i> :	Angular speed $\in [-1.0; +1.0]$
	float <i>magneto_psi</i> :	Angle of controlling device from north $\in [-1.0; +1.0]$
	float <i>magneto_psi_accuracy</i> :	Accuracy of the magneto_psi value $\in [-1.0; +1.0]$

Description :

This function makes the drone move in the air. It has no effect when the drone lies on the ground.

The drone is controlled by giving as a command a set of four parameters :

- a left/right bending angle, with 0 being the horizontal plane, and negative values bending leftward
- a front/back bending angle, with 0 being the horizontal plane, and negative values bending frontward
- a vertical speed
- an angular speed around the yaw-axis

In order to allow the user to choose between smooth or dynamic moves, the arguments of this function are not directly the control parameters values, but a percentage of the maximum corresponding values as set in the drone parameters. All parameters must thus be floating-point values between -1.0 and 1.0.

The flags argument is a bitfields containing the following informations :

- Bit 0 : when Bit0=0 the drone will enter the *hovering* mode, i.e. try to stay on top of a fixed point on the ground, else it will follow the commands set as parameters.
- Bit 1 : when Bit1=1 AND CONTROL:control_level configuration Bit1=1, the new Combined Yaw mode is activated. This mode includes a complex hybridation of the phi parameter to generate complete turns (phi+yaw).

- Bit 2 : when Bit2=1, the Absolute Control mode is activated (only for AR.Drone 2.0). All the commands will be considered in the controller frame instead of the AR.Drone 2.0 frame (e.g. front/back commands are relative to the user front/back, and not the AR.Drone 2.0 front/back)



5

Creating an application with ARDroneTool

The **ARDroneTool** library includes all the code needed to start your application. All you have to do is writing your application specific code, and compile it along with the **ARDroneLIB** library to get a fully functional drone client application which will connect to the drone and start interacting with it.

This chapter shows you how to quickly get a customized application that suits your needs.

You can try to immediately start customizing your own application by reading section [5.1](#), but it is recommended you read the whole chapter to understand what is going on inside.

5.1 Quick steps to create a custom AR.Drone 2.0 application

The fastest way to get an up and running application is to copy the SDK Demo application folder and bring the following modifications to suit your needs :

- create a new thread and add it to the *THREAD_TABLE* structure to send commands independently from the above-mentioned events (more details in [5.5](#))
- call the *ardrone_tool_main* function from your application.
- create any needed navdata handler and add it to the *ardrone_navdata_handler_table*.

To compile your customized demo, please refer to the tutorials.

5.2 Customizing the client initialization

The **ARDroneTool** library includes a custom *ardrone_tool_main* function with all the code needed to start your application. All you have to do is writing your application specific code, and compile it along with the **ARDroneLIB** library and call this function to get a fully functional drone client application.

Listing 5.1 shows the *ardrone_tool_main* function for the ARDrone application. It is located in the file *ardrone_tool.c* and should not require any modification. Every application you create will have a *ardrone_tool_main* function that is almost identical to this one.

This function performs the following tasks :

- Configures WIFI network.
- Initializes the communication ports (AT commands, Navdata, Video and Control).
- Calls the *ardrone_tool_init_custom* function. Its prototype is defined in *ardrone_tool.h* file, and must be defined and customized by the developer (see example 5.2). In this function we can find:
 - the local initialization for your own application.
 - the initialization of input devices, with the *ardrone_tool_input_init* function
 - the starting off all threads except the *navdata_update* and *ardrone_control* that are started by the *ardrone_tool_main* function.
- Starts the thread *navdata_update* that is located in *ardrone_navdata_client.c* file. To run properly this routine, the user must declare a table *ardrone_navdata_handler_table*. Listing 3 shows how to declare an *ardrone_navdata_handler* table. The MACRO is located in *ardrone_navdata_client.h* file.
- Starts the thread *ardrone_control* that is located in *ardrone_control.c* file.
- Acknowledge the Drone to indicate that we are ready to receive the Navdata.
- At last call *ardrone_tool_update* function in loop. The application does not return from this function until it quits. This function retrieves the device information to send to the Drone. The user can declare *ardrone_tool_update_custom* function, that will be called by the *ardrone_tool_update* function.

Listing 5.1: Application initialization with ARDroneLIB

```

int ardrone_tool_main(int argc, char *argv[])
{
    ...
    ardrone_tool__setup__com( NULL );
    ardrone_tool_init(argc, argv);
    while( SUCCEED(res) && ardrone_tool_exit() == FALSE )
    {
        res = ardrone_tool_update();
    }
    res = ardrone_tool_shutdown();
}

```

Listing 5.2: Custom application initialization example

```

C_RESULT ardrone_tool_init_custom(int argc, char **argv)
{
    gtk_init(&argc, &argv);
    /// Init specific code for application
    ardrone_navdata_handler_table[NAVDATA_IHM_PROCESS_INDEX].data = &cfg;
    // Add inputs
    ardrone_tool_input_add( &gamepad );
    // Sample run thread with ARDrone API.
    START_THREAD(ihm, &cfg);
    return C_OK;
}

```

5.3 Using navigation data

During the application lifetime, the **ARDroneTool** library automatically calls a set of user-defined callback functions every time some navdata arrive from the drone.

Declaring such a callback function is done by adding it to the **NAVDATA_HANDLER_TABLE** table. In code example 5.3, a *navdata_ihm_process* function, written by the user, is declared.

Note : the callback function prototype must be the one used in code example 5.3.

Listing 5.3: Declare a *navdata* management function

```
BEGIN_NAVDATA_HANDLER_TABLE //Mandatory
    NAVDATA_HANDLER_TABLE_ENTRY(navdata_ihm_init, navdata_ihm_process,
        navdata_ihm_release,
NULL)
END_NAVDATA_HANDLER_TABLE //Mandatory
//Definition for init, process and release functions.
C_RESULT navdata_ihm_init( mobile_config_t* cfg )
{ ... }

C_RESULT navdata_ihm_process( const navdata_unpacked_t* const pnd )
{ ... }

C_RESULT navdata_ihm_release( void )
{ ... }
```

Listing 5.4: Example of *navdata* management function

```
/* Receiving navdata during the event loop */
inline C_RESULT demo_navdata_client_process( const navdata_unpacked_t* const
    navdata )
{
    const navdata_demo_t* const nd = &navdata->navdata_demo;

    printf("Navdata for flight demonstrations\n");

    printf("Control state : %s\n",ctrl_state_str(nd->ctrl_state));
    printf("Battery level : %i/100\n",nd->vbat_flying_percentage);
    printf("Orientation   : [Theta] %f [Phi] %f [Psi] %f\n",nd->theta,nd->phi,nd->
        psi);
    printf("Altitude      : %i\n",nd->altitude);
    printf("Speed         : [vX] %f [vY] %f\n",nd->vx,nd->vy);

    printf("\033[6A"); // Ansi escape code to go up 6 lines

    return C_OK;
}
```

5.4 Command line parsing for a particular application

The user can implement functions to add arguments to the default command line. Functions are defined in `<ardrone_tool/ardrone_tool.h>` file :

- `ardrone_tool_display_cmd_line_custom` (Not mandatory): Displays help for particular commands.
- `ardrone_tool_check_argc_custom` (Not mandatory) : Checks the number of arguments.
- `ardrone_tool_parse_cmd_line_custom` (Not mandatory): Checks a particular line command.

5.5 Thread management in the application

In the preceding section, we showed how the ARDrone application was initialized and how it manages the Navdata and control events. In addition to those aspects of the application creation, there are also smaller details that need to be considered before building a final application.

It's the responsibility of the user to manage the threads. To do so, we must declare a thread table with MACRO defined in `vp_api_thread_helper.h` file. Listing 5.5 shows how to declare a threads table.

The threads `navdata_update` and `ardrone_control` do not need to be launched and released; this is done by the ARDroneMain for all other threads, you must use the MACRO named `START_THREAD` and `JOIN_THREAD`.

In the preceding sections, we have seen that the user must declare functions and tables (`ardrone_tool_init_custom`, `ardrone_tool_update_custom`, `ardrone_navdata_handler_table` and `threads` table), other objects can be defined by the user but it is not mandatory :

- `ardrone_tool_exit` function, which should return true to exit the main loop
- `ardrone_tool_shutdown` function where you can release the resources.

These functions are defined in `ardrone_tool.h`.

Listing 5.5: Declaration of a threads table

```
BEGIN_THREAD_TABLE //Mandatory
THREAD_TABLE_ENTRY( ihm, 20 ) // For your own application
THREAD_TABLE_ENTRY( navdata_update, 20 ) //Mandatory
THREAD_TABLE_ENTRY( ardrone_control, 20 ) //Mandatory
THREAD_TABLE_ENTRY( video_stage, 20 ) //Mandatory
THREAD_TABLE_ENTRY( video_recorder, 20 ) //Mandatory for AR.Drone 2.0
END_THREAD_TABLE //Mandatory
```

5.6 Managing the video stream

This SDK includes methods to manage the video stream. The whole process is managed by a *video pipeline*, built as a sequence of *stages* which perform basic steps, such as receiving the video data from a socket, decoding the frames, and displaying them.

It is strongly recommended to have a look at the *video_stage.c* file in the code examples to see how this works, and to modify it to suit your needs. In the examples a fully functionnal pipeline is already created, and you will probably just want to modify the displaying part.

A stage is embedded in a structure named *vp_api_io_stage_t* that is defined in the file *<VP_Api/vp_api.h>*.

Listing 5.6 shows how to add custom stages to the default video pipeline. This must be done by your application before calling the *START_THREAD* function for *video_stage*.

Listing 5.6: Adding custom stages to the live video pipeline

```

#define STREAM_WIDTH 320
#define STREAM_HEIGHT 240
#define NB_PRE_STAGES 0
#define NB_POST_STAGES 1

//Alloc structs
specific_parameters_t * params      = (specific_parameters_t *)vp_os_calloc(1,
                                         sizeof(specific_parameters_t));
specific_stages_t * pre_stages      = (specific_stages_t *)vp_os_calloc(1,
                                         sizeof(specific_stages_t));
specific_stages_t * post_stages      = (specific_stages_t *)vp_os_calloc(1,
                                         sizeof(specific_stages_t));
vp_api_picture_t * in_picture       = (vp_api_picture_t *) vp_os_calloc(1,
                                         sizeof(vp_api_picture_t));
vp_api_picture_t * out_picture       = (vp_api_picture_t *) vp_os_calloc(1,
                                         sizeof(vp_api_picture_t));

// Mandatory for both AR.Drone 1.0 and 2.0
out_picture->format      = PIX_FMT_RGB565;

// Mandatory for AR.Drone 1.0
in_picture->width        = STREAM_WIDTH;
in_picture->height        = STREAM_HEIGHT;

out_picture->framerate     = 20;
out_picture->width         = in_picture->width;
out_picture->height        = in_picture->height;

out_picture->y_buf          = vp_os_malloc( out_picture->width * out_picture->
                                             height * 2 ); // RGB565 needs 2 bytes per pixel
out_picture->cr_buf         = NULL;
out_picture->cb_buf         = NULL;

out_picture->y_line_size    = out_picture->width * 2; // RGB565 needs 2 bytes per
                                                         pixel
out_picture->cb_line_size   = 0;
out_picture->cr_line_size   = 0;

//Define the list of stages size
pre_stages->length  = NB_PRE_STAGES;
post_stages->length  = NB_POST_STAGES;

//Alloc the lists
pre_stages->stages_list  = NULL;
post_stages->stages_list = (vp_api_io_stage_t *)vp_os_calloc(post_stages->length,
                                         sizeof(vp_api_io_stage_t));

//Fill the POST-stages-----
int postStageNumber = 0;

custom_video_display_stage_config_t cvdsc;
vp_os_memset (&cvdsc, 0x0, sizeof(cvdsc));
/* Initialize your display stage config here */
post_stages->stages_list[postStageNumber].type  = VP_API_OUTPUT_LCD;
post_stages->stages_list[postStageNumber].cfg    = (void *)&cvdsc;
post_stages->stages_list[postStageNumber++].funcs =
    custom_video_display_stage_funcs;

params->in_pic = in_picture;
params->out_pic = out_picture;
params->pre_processing_stages_list = pre_stages;
params->post_processing_stages_list = post_stages;
params->needSetPriority = 0;
params->priority = 0;

START_THREAD(video_stage, params);

```




AT commands are text strings sent to the drone to control its actions.

Those strings are generated by the **ARDroneLIB** and **ARDroneTool** libraries, provided in the SDK. Most developers should not have to deal with them. Advanced developers who would like to rewrite their own AR.Drone middle ware can nevertheless send directly those commands to the drone inside UDP packets on port UDP-5556, from their local UDP-port 5556 (using the same port numbers on both sides of the UDP/IP link is a requirement in the current SDK).

Note : According to tests, a satisfying control of the AR.Drone 2.0 is reached by sending the AT-commands every 30 ms for smooth drone movements. To prevent the drone from considering the WIFI connection as lost, two consecutive commands must be sent within less than 2 seconds.

6.1 AT Commands syntax

Strings are encoded as 8-bit ASCII characters, with a *Carriage Return* character (byte value $0D_{(16)}$), noted <CR>hereafter, as a newline delimiter.

One command consists in the three characters **AT*** (i.e. three 8-bit words with values $41_{(16)}, 54_{(16)}, 2a_{(16)}$) followed by a command name, and equal sign, a sequence number, and optionally a list of comma-separated arguments whose meaning depends on the command.

A single UDP packet can contain one or more commands, separated by newlines (byte value $0A_{(16)}$). An AT command must reside in a single UDP packet. Splitting an AT command in two or more UDP packets is not possible.

Example :

```
AT*PCMD_MAG=21625,1,0,0,0,0,0<CR>AT*REF=21626,290717696<CR>
```

The maximum length of the total command cannot exceed 1024 characters; otherwise the entire command line is rejected. This limit is hard coded in the drone software.

Note : Incorrect AT commands should be ignored by the drone. Nevertheless, the client should always make sure it sends correctly formed UDP packets.

Most commands will accept arguments, which can be of three different type :

- A signed integer, stored in the command string with a decimal representation (ex: the sequence number)
- A string value stored between double quotes (ex: the arguments of AT*CONFIG)
- A single-precision IEEE-754 floating-point value (aka. *float*). Those are never directly stored in the command string. Instead, the 32-bit word containing the *float* will be considered as a 32-bit signed integer and printed in the AT command (an example is given below).

6.2 Commands sequencing

In order to avoid the drone from processing old commands, a sequence number is associated to each sent AT command, and is stored as the first number after the "equal" sign. The drone will not execute any command whose sequence number is less than the last valid received AT-Command sequence number. This sequence number is reset to 1 inside the drone every time a client disconnects from the AT-Command UDP port (currently this disconnection is done by not sending any command during more than 2 seconds), and when a command is received with a sequence number set to 1.

A client MUST thus respect the following rule in order to successfully execute commands on the drone :

- Always send 1 as the sequence number of the first sent command.
- Always send commands with an increasing sequence number. If several software threads send commands to the drone, generating the sequence number and sending UDP packets should be done by a single dedicated function protected by a mutual exclusion mechanism.

6.3 Floating-point parameters

Let's see an example of using a *float* argument and consider that a progressive command is to be sent with an argument of -0.8 for the pitch. The number -0.8 is stored in memory as a 32-bit word whose value is $BF4CCCCD_{(16)}$, according to the IEEE-754 format. This 32-bit word can be considered as holding the 32-bit integer value $-1085485875_{(10)}$. So the command to send will be **AT*PCMD_MAG=xx,xx,-1085485875,xx,xx,xx,xx**.

Listing 6.1: Example of AT command with floating-point arguments

```
assert(sizeof(int)==sizeof(float));
sprintf(my_buffer, "AT*PCMD_MAG,%d,%d,%d,%d,%d,%d\r",
    sequence_number,
    *(int*)(&my_floating_point_variable_1),
    *(int*)(&my_floating_point_variable_2),
    *(int*)(&my_floating_point_variable_3),
    *(int*)(&my_floating_point_variable_4),
    *(int*)(&my_floating_point_variable_5),
    *(int*)(&my_floating_point_variable_6) );
```

The **ARDroneLIB** provides a C union to ease this conversion. You can use the **_float_or_int_t** to store a float or an int in the same memory space, and use it as any of the two types.

6.4 AT Commands summary

AT command	Arguments ¹	Description
AT*REF	input	Takeoff/Landing/Emergency stop command
AT*PCMD	flag, roll, pitch, gaz, yaw	Move the drone
AT*PCMD_MAG	flag, roll, pitch, gaz, yaw, psi, psi accuracy	Move the drone (with Absolute Control support)
AT*FTRIM	-	Sets the reference for the horizontal plane (must be on ground)
AT*CONFIG	key, value	Configuration of the AR.Drone 2.0
AT*CONFIG_IDS	session, user, ap- plication ids	Identifiers for AT*CONFIG commands
AT*COMWDG	-	Reset the communication watchdog
AT*CALIB	device number	Ask the drone to calibrate the magnetometer (must be flying)

¹apart from the sequence number

6.5 Commands description

AT*REF

Summary : Controls the basic behaviour of the drone (take-off/landing, emergency stop/reset)

Corresponding API function : [ardrone_tool_set_ui_pad_start](#)

Corresponding API function : [ardrone_tool_set_ui_pad_select](#)

Syntax : AT*REF=%d,%d<CR>

Argument 1 : the sequence number

Argument 2 : an integer value in [0.. $2^{32} - 1$], representing a 32 bit-wide bit-field controlling the drone.

Description :

Send this command to control the basic behaviour of the drone. With SDK version 1.5, only bits 8 and 9 are used in the control bit-field. Bits 18, 20, 22, 24 and 28 should be set to 1. Other bits should be set to 0.

Bits	31 .. 10	9	8	7 .. 0
Usage	Do not use	Takeoff/Land (aka. "start bit")	Emergency (aka. "select bit")	Do not use

Bit 9 usages :

Send a command with this bit set to 1 to make the drone take-off. This command should be repeated until the drone state in the navdata shows that drone actually took off. If no other command is supplied, the drone enters a hovering mode and stays still at approximately 1 meter above ground.

Send a command with this bit set to 0 to make the drone land. This command should be repeated until the drone state in the navdata shows that drone actually landed, and should be sent as a safety whenever an abnormal situation is detected.

After the first *start* AT-Command, the drone is in the *taking-Off* state, but still accepts other commands. It means that while the drone is rising in the air to the "1-meter-high-hovering state", the user can send orders to move or rotate it.

Bit 8 usages :

When the drone is a "normal" state (flying or waiting on the ground), sending a command with this bit set to 1 (ie. sending an "emergency order") makes the drone enter an emergency mode. Engines are cut-off no matter the drone state. (ie. the drone crashes, potentially violently).

When the drone is in an emergency mode (following a previous emergency order or a crash), sending a command with this bit set to 1 (ie. sending an "emergency order") makes the drone resume to a normal state (allowing it to take-off again), at the condition the cause of the emergency was solved.

Send an AT*REF command with this bit set to 0 to make the drone consider following "emergency orders" commands (this prevents consecutive "emergency orders" from flip-flopping the drone state between emergency and normal states).

Note :

The names "start" and "select" come from previous versions of the SDK when take-off and landing were directly managed by pressing the select and start buttons of a game pad.

Example :

The following commands sent in a standalone UDP packet will send an emergency signal :

AT*REF=1,290717696<CR>AT*REF=2,290717952<CR>AT*REF=3,290717696<CR>

AT*PCMD / AT*PCMD_MAG

Summary : *Send progressive commands* - makes the drone move (translate/rotate).

Corresponding API function : *ardrone_at_set_progress_cmd*

Corresponding API function : *ardrone_at_set_progress_cmd_with_magneto*

Syntax : **AT*PCMD=%d,%d,%d,%d,%d,%d<CR>**

Syntax : **AT*PCMD_MAG=%d,%d,%d,%d,%d,%d,%d,<CR>**

Argument 1 : the sequence number

Argument 2 : flag enabling the use of progressive commands and/or the Combined Yaw mode (bitfield)

Argument 3 : drone left-right tilt - floating-point value in range [-1..1]

Argument 4 : drone front-back tilt - floating-point value in range [-1..1]

Argument 5 : drone vertical speed - floating-point value in range [-1..1]

Argument 6 : drone angular speed - floating-point value in range [-1..1]

Argument 7 : magneto psi (only for AT*PCMD_MAG) - floating-point value in range [-1..1]

Argument 8 : magneto psi accuracy (only for AT*PCMD_MAG) - floating-point value in range [-1..1]

Description :

This command controls the drone flight motions.

Always set the flag (argument 2) bit zero to one to make the drone consider the other arguments. Setting it to zero makes the drone enter *hovering* mode (staying on top of the same point on the ground).

Bits	31 .. 3	2	1	0
Usage	Do not use	Absolute Control enable	Combined yaw enable	Progressive commands enable

The left-right tilt (aka. "drone roll" or phi angle) argument is a percentage of the maximum inclination as configured here. A negative value makes the drone tilt to its left, thus flying leftward. A positive value makes the drone tilt to its right, thus flying rightward.

The front-back tilt (aka. "drone pitch" or theta angle) argument is a percentage of the maximum inclination as configured here. A negative value makes the drone lower its nose, thus flying forward. A positive value makes the drone raise its nose, thus flying backward.

The drone translation speed in the horizontal plane depends on the environment and cannot be determined. With roll or pitch values set to 0, the drone will stay horizontal but continue sliding in the air because of its inertia. Only the air resistance will then make it stop.

The vertical speed (aka. "gaz") argument is a percentage of the maximum vertical speed as defined here. A positive value makes the drone rise in the air. A negative value makes it go down.

The angular speed argument is a percentage of the maximum angular speed as defined here. A positive value makes the drone spin right; a negative value makes it spin left.

The psi argument is a normalized psi angle from north provided by magnetometer sensor as defined here. An angle value of 0 means that the controller is facing north. A positive value means that the controller is oriented to the east and a negative value is orienting to the west. 1 and -1 value are the same orientation. (only for AT*PCMD_MAG)

The psi accuracy argument is an accuracy of the magnetometer sensor. This value represents the maximum deviation of where the magnetic heading may differ from the actual geomagnetic heading in degrees. Negative values indicates the invalid heading. (only for AT*PCMD_MAG)

AT*FTRIM

Summary : *Flat trims* - Tells the drone it is lying horizontally

Corresponding API function : *ardrone_at_set_flat_trim*

Syntax : AT*FTRIM=%d,<CR>

Argument 1 : the sequence number

Description :

This command sets a reference of the horizontal plane for the drone internal control system.

It must be called after each drone start up, while making sure the drone actually sits on a horizontal ground. Not doing so before taking-off will result in the drone not being able to stabilize itself when flying, as it would not be able to know its actual tilt. This command *MUST NOT* be sent when the AR.Drone is flying.

When receiving this command, the drone will automatically adjust the **trim** on pitch and roll controls.

AT*CALIB

Summary : *Magnetometer calibration* - Tells the drone to calibrate its magnetometer

Corresponding API function : *ardrone_at_set_calibration*

Syntax : AT*CALIB=%d,%d,<CR>

Argument 1 : the sequence number

Argument 2 : Identifier of the device to calibrate - Choose this identifier from **ardrone_calibration_device_t**.

Description :

This command asks the drone to calibrate the drone magnetometer. This command *MUST* be sent when the AR.Drone is flying.

When receiving this command, the drone will automatically calibrate its magnetometer by spinning around itself for a few time.

AT*CONFIG

Summary : Sets an configurable option on the drone

Corresponding API function : *ardrone_at_set_toy_configuration*

Syntax : AT*CONFIG=%d,%s,%s<CR>

Argument 1 : the sequence number

Argument 2 : the name of the option to set, between double quotes (byte with hex.value 22h)

Argument 3 : the option value, between double quotes

Description :

Most options that can be configured are set using this command. The list of configuration options can be found in chapter [8](#).

AT*CONFIG_IDS

Summary : Identifiers for the next AT*CONFIG command

Corresponding API function : *ardrone_at_set_toy_configuration*

Syntax : AT*CONFIG_IDS=%d,%s,%s,%s<CR>

Argument 1 : the sequence number

Argument 2 : Current session id

Argument 3 : Current user id

Argument 4 : Current application id

Description :

While in multiconfiguration, you must send this command before every AT*CONFIG. The config will only be applied if the ids must match the current ids on the AR.Drone.

ARDroneTool does this automatically.

AT*COMWDG

Summary : reset communication watchdog



7

Incoming data streams

The drone provides its clients with two main data streams : the navigation data (aka. *navdata*) stream, and the video stream.

This chapter explains their format. This is useful for developers writing their own middleware. Developers using **ARDroneTool** can skip this part and directly access these data from the callback function triggered by **ARDroneTool** when receiving incoming data from the drone (see [5.3](#), [7.2](#) and [7.3](#)).

7.1 Navigation data

The navigation data (or *navdata*) is a mean given to a client application to receive periodically (< 5ms) information on the drone status (angles, altitude, camera, velocity, tag detection results ...).

This section shows how to retrieve them and decode them. Do not hesitate to use network traffic analysers like Wireshark to see how they look like.

7.1.1 Navigation data stream

The *navdata* are sent by the drone from and to the UDP port 5554. Information are stored in a binary format and consist in several sections blocks of data called *options*.

Each option consists in a header (2 bytes) identifying the kind of information contained in it, a 16-bit integer storing the size of the block, and several information stored as 32-bit integers, 32-bit single precision floating-point numbers, or arrays. All those data are stored with little-endianess.

Header 0x55667788	Drone state	Sequence number	Vision flag	Option 1			...	Checksum block		
32-bit int.	32-bit int.	32-bit int.	32-bit int.	id	size	data	...	cks id	size	cks data
				16-bit int.	16-bit int.	16-bit int.	16-bit int.	32-bit int.

All the blocks share this common structure :

Listing 7.1: Navdata option structure

```
typedef struct _navdata_option_t {
    uint16_t tag; /* Tag for a specific option */
    uint16_t size; /* Length of the struct */
    uint8_t data[]; /* Structure complete with the special tag */
} navdata_option_t;
```

The most important *options* are *navdata_demo_t*, *navdata_cks_t*, *navdata_host_angles_t* and *navdata_vision_detect_t*. Their content can be found in the C structure, mainly in the *navdata_common.h*.

7.1.2 Initiating the reception of Navigation data

To receive Navdata, you must send a packet of some bytes on the port **NAVDATA_PORT** of host.

Two cases :

- the drone starts in bootstrap mode, only the status and the sequence counter are sent.
- the Drone is always started, Navdata demo are send.

To exit BOOTSTRAP mode, the client must send an AT command in order to modify configuration on the Drone. Send AT command: "AT*CONFIG=\\"general:navdata_demo\\", \\"TRUE\\"\\r". Ack control command, send AT command: "AT*CTRL=0. The drone is now initialized and sends Navdata demo. This mechanism is summarized by figure 7.1.

How do the client and the drone synchronize ?

The client application can verify that the sequence counter contained in the header structure of NavData is growing.

There are two cases when the local (client side) sequence counter should be reset :

- the drone does not receive any traffic for more than 50ms; it will then set its **ARDRONE_COM_WATCHDOG_MASK** bit in the *ardrone_state* field (2nd field) of the *navdata* packet. To exit this mode, the client must send the AT Command **AT*COMWDG**.
- The drone does not receive any traffic for more than 2000ms; it will then stop all communication with the client, and internally set the **ARDRONE_COM_LOST_MASK** bit in its state variable. The client must then reinitialize the network communication with the drone.

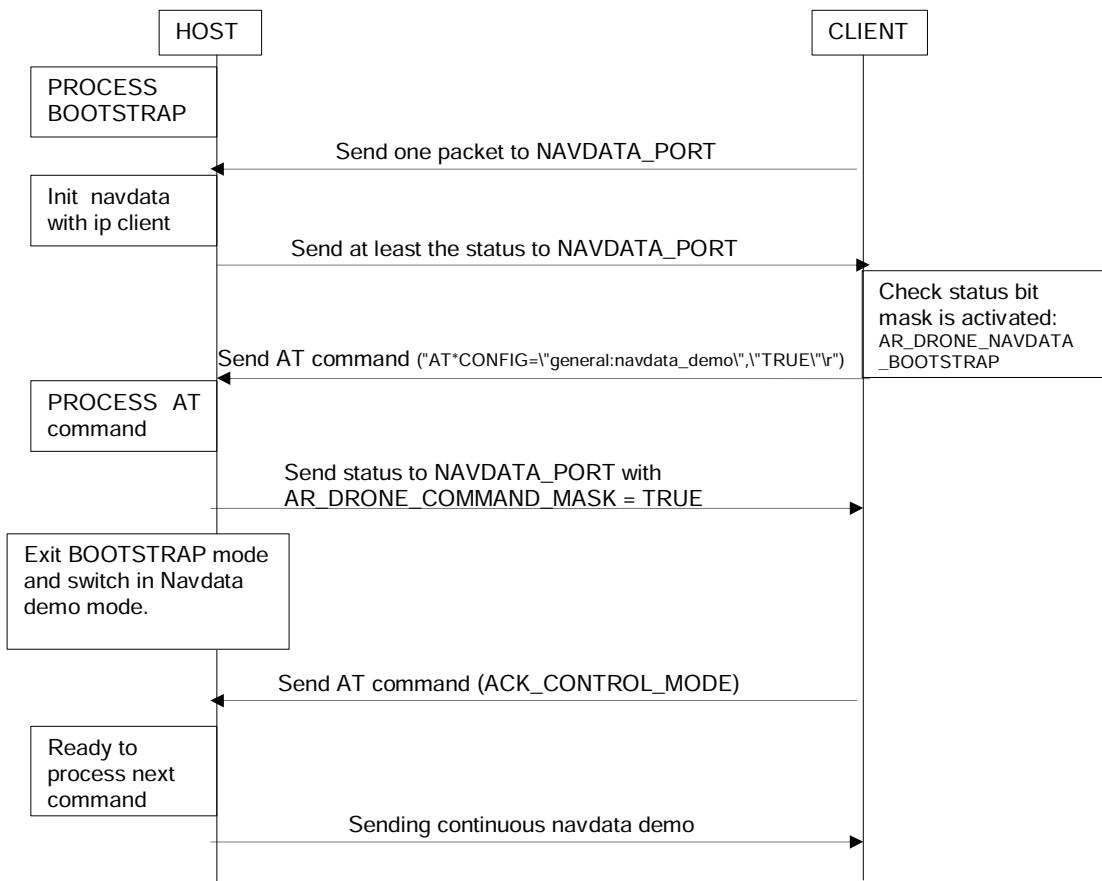


Figure 7.1: Navdata stream initiation

How to check the integrity of NavData ?

Compute a checksum of data and compare them with the value contained in the structure [*navdata_cks_t*]. The checksum is always the last *option* (data block) in the *navdata* packet.

Note : this checksum is already computed by **ARDroneLIB** .

7.1.3 Augmented reality data stream

In the previously described NavData, there are informations about vision-detected tags. The goal is to permit to the host to add some functionalities, like augmented reality features. The principle is that the AR.Drone sends informations on recognized [pre-defined tags](#), like type and position.

Listing 7.2: Navdata option for vision detection

```
typedef struct _navdata_vision_detect_t {
    uint16_t tag;
    uint16_t size;
    uint32_t nb_detected;
    uint32_t type[NB_NAVDATA_DETECTION_RESULTS];
    uint32_t xc[NB_NAVDATA_DETECTION_RESULTS];
    uint32_t yc[NB_NAVDATA_DETECTION_RESULTS];
    uint32_t width[NB_NAVDATA_DETECTION_RESULTS];
    uint32_t height[NB_NAVDATA_DETECTION_RESULTS];
    uint32_t dist[NB_NAVDATA_DETECTION_RESULTS];
    float32_t orientation_angle[NB_NAVDATA_DETECTION_RESULTS];
    matrix33_t rotation[NB_NAVDATA_DETECTION_RESULTS];
    vector31_t translation[NB_NAVDATA_DETECTION_RESULTS];
    uint32_t camera_source[NB_NAVDATA_DETECTION_RESULTS];
} __attribute__ ((packed)) navdata_vision_detect_t;
```

The drone can detect up to four tags or oriented roundel. The kind of detected tag, and which camera to use, can be set by using the configuration parameter [detect_type](#).

Let's detail the values in this block :

- ***nb_detected***: number of detected tags or oriented roundel.
- ***type[i]***: Type of the detected tag or oriented roundel #i ; see the **CAD_TYPE** enumeration.
- ***xc[i], yc[i]***: X and Y coordinates of detected tag or oriented roundel #i inside the picture, with (0, 0) being the top-left corner, and (1000, 1000) the right-bottom corner regardless the picture resolution or the source camera.
- ***width[i], height[i]***: Width and height of the detection bounding-box (tag or oriented roundel #i), when applicable.
- ***dist[i]***: Distance from camera to detected tag or oriented roundel #i in centimeters, when applicable.
- ***orientation_angle[i]*** : Angle of the oriented roundel #i in degrees in the screen, when applicable.
- ***rotation[i]*** : Reserved for future use.
- ***translation[i]*** : Reserved for future use.
- ***camera_source[i]*** : Camera Source which detected tag or oriented roundel #i.

7.2 The AR.Drone 1.0 video stream

Two codecs are available UVLC (MJPEG-like) and P264 (H.264-like).

UVLC features :

colorspace	YUV 4:2:0
transform	8x8 dct
entropy coding	RLE+UVLC

P264 vs H264 :

Feature	P264	H264 baseline profile
transform	pseudo dct 4x4 4x4 luma DC transform 2x2 Chroma DC transform	pseudo dct 4x4 4x4 luma DC transform 2x2 Chroma DC transform
frame type	I/P	I/P
intra 4x4 prediction	mode 0-8	mode 0-8
intra 16x16 prediction	mode 0-3	mode 0-3
intra 8x8 chroma prediction	mode 0-3	mode 0-3
macroblock partition for motion compensation	16x16 only	16x16,16x8,8x16,8x8,8x4,4x8,4x4
motion compensation precision	1 pixel	1/4 pixel
entropy	RLE+UVLC	CAVLC

7.2.1 Image structure

An image is split in groups of blocks (GOB), which correspond to 16-lines-height parts of the image, split as shown below :

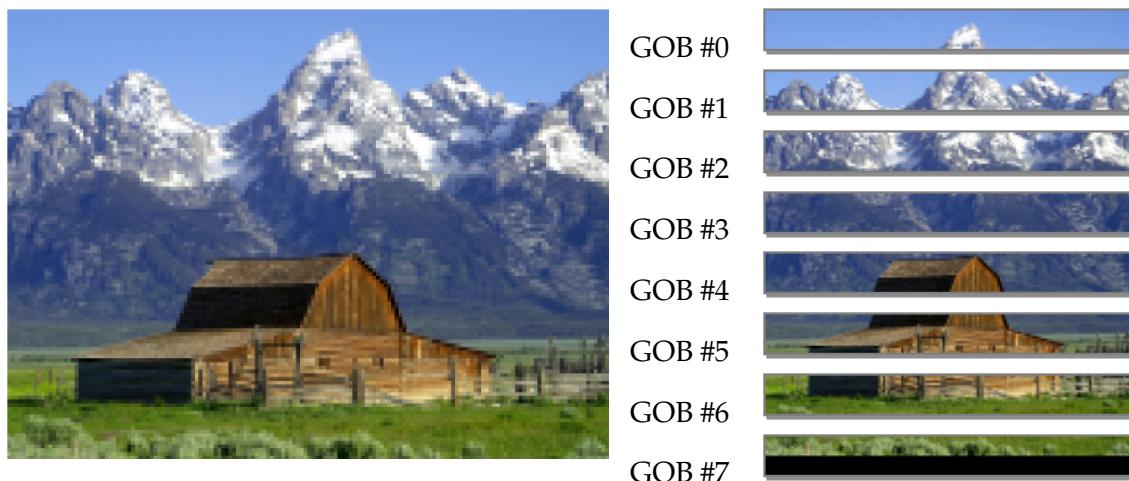
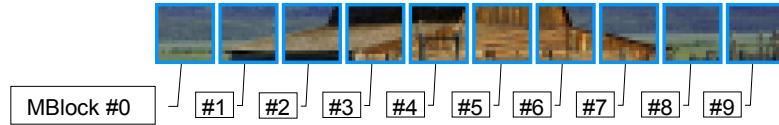


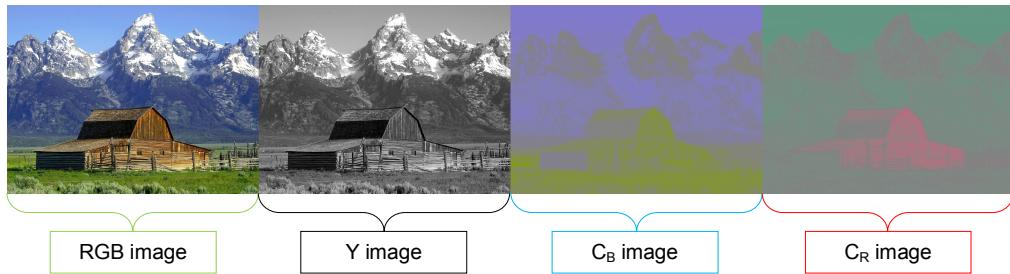
Figure 7.2: Example image (source : <http://en.wikipedia.org/wiki/YCbCr>)

Each GOB is split in Macroblocks, which represents a 16x16 image.



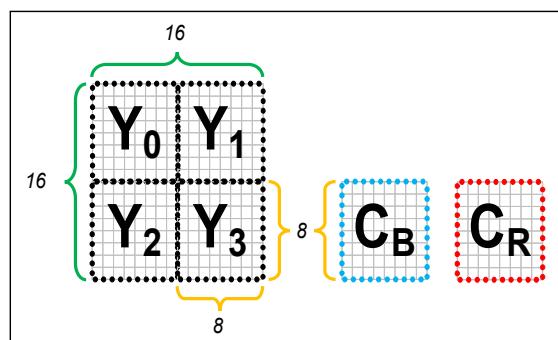
Each macroblock contains informations of a 16x16 image, in $Y\text{C}_B\text{C}_R$ format, type 4:2:0.

(see <http://en.wikipedia.org/wiki/YCbCr>,
http://en.wikipedia.org/wiki/Chroma_subsampling,
<http://www.ripp-it.com/glossaire/mot-420-146-lettre-tous-Categorie-toutes.html>)



The 16x16 image is finally stored in the memory as 6 blocks of 8x8 values:

- 4 blocks (Y_0 , Y_1 , Y_2 and Y_3) to form the 16x16 pixels Y image of the luma component (corresponding to a greyscale version of the original 16x16 RGB image).
- 2 blocks of down-sampled chroma components (computed from the original 16x16 RGB image):
 - C_B : blue-difference component (8x8 values)
 - C_R : red-difference component (8x8 values)



7.2.2 UVLC codec overview

UVLC codec is very closed to JPEG please refer to <http://en.wikipedia.org/wiki/JPEG> for more details.

Step 1: each 8x8 block of the current macroblock is transformed by DCT.

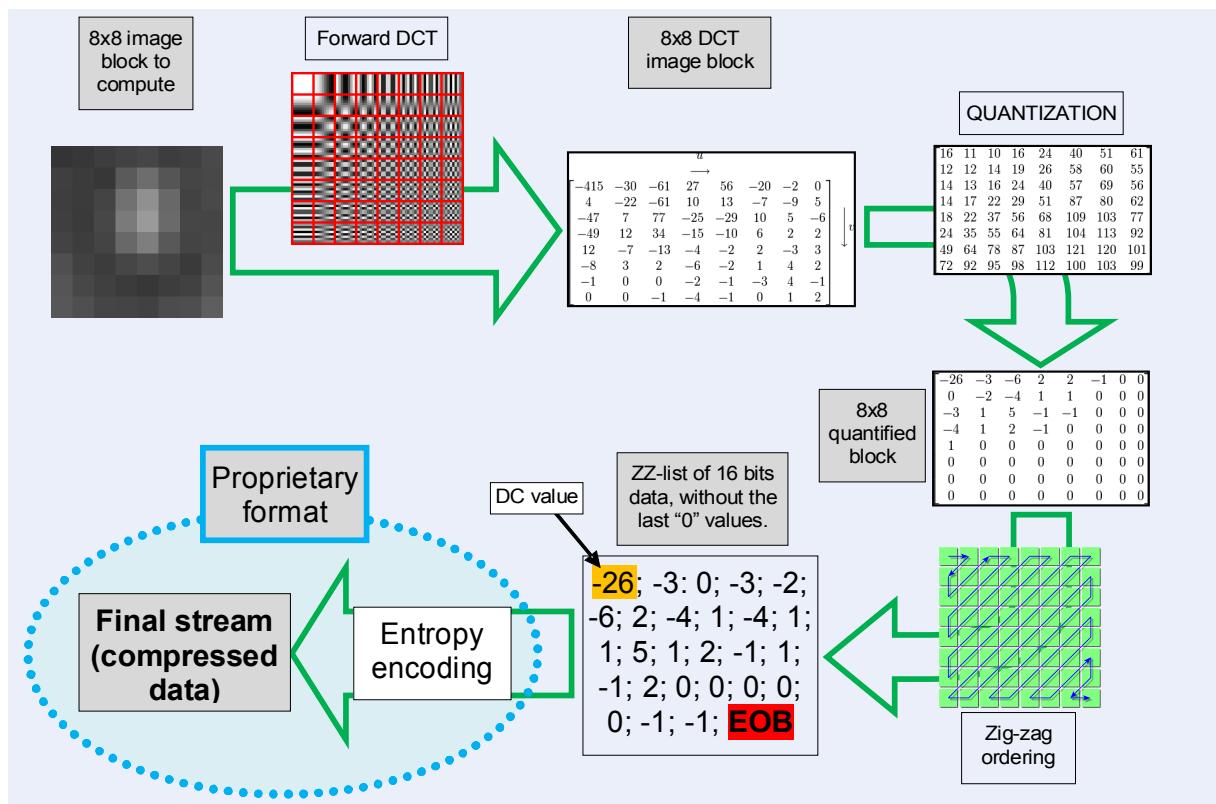
Step 2: each element of the transformed 8x8 block is divided by a quantization coefficient. The quantization matrix used in UVLC codec is defined by:

$$QUANT_IJ(i, j, q) = (1 + (1 + (i) + (j)) * (q))$$

where i,j are the index of current element in the 8x8 block, and q is a number between 1 and 30. A low q, produces a better image but more bytes are used to encode it

Step 3: the block 8x8 is then zig zag reordered.

Step 4: the block 8x8 is then encoded using UVLC method (see entropy coding section)



7.2.3 P264 codec overview

Since the encoding concepts involved comes from the H264 specification, this part is only a brief description of P264. Please refer to the *Recommendation ITU-T H.264* for further details.

7.2.3.1 I Frame

An I frame is a complete frame. No reference to any previous frame is needed to decode it. Like H264, P264 makes a spatial prediction for each macroblock based on the neighbouring pixels.

Several mode are available for I macroblock:

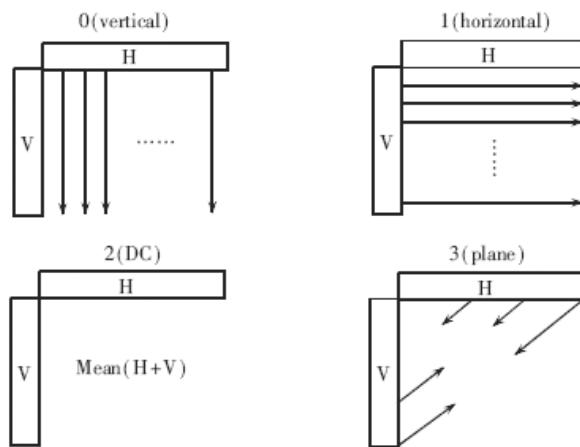
- intra 16x16 - a prediction is made over the all 16x16 macroblock.
- intra 4x4 - the macroblock is divided into 16 4x4 blocks. Each 4x4 block has its own intra prediction

Once the intra prediction is done, it is subtracted from the current macroblock. The residual data is then processed with classical steps : transform, quantization, entropy coding. (see residual data section)

7.2.3.1.1 Luma intra 16x16 prediction

4 modes are available:

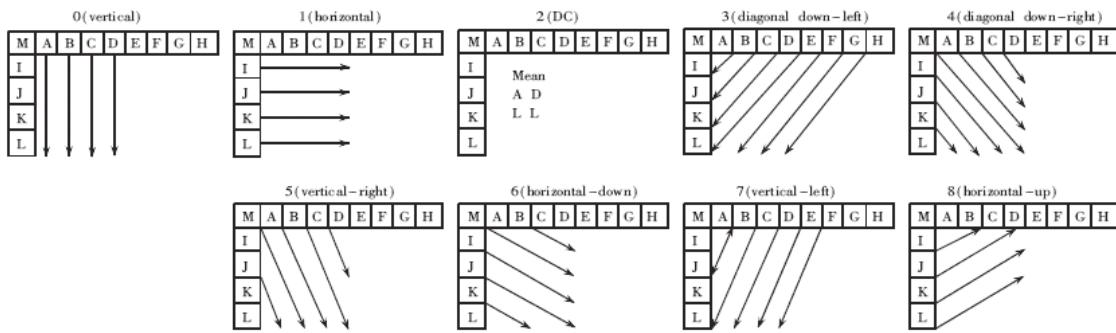
- VERTICAL - extends the 16 upper neighbour pixels over the all 16x16 macroblock
- HORIZONTAL - extends the 16 left neighbour pixels over the all 16x16 macroblock
- DC - fills the 16x16 block with the mean of the 16 upper and the 16 left neighbour pixels
- PLANE - makes interpolation of the 16 upper and the 16 left neighbour pixels



7.2.3.1.2 Luma intra 4x4 prediction

9 modes are available :

- VERTICAL_4x4_MODE
- HORIZONTAL_4x4_MODE
- DC_4x4_MODE
- DIAGONAL_DL_4x4_MODE
- DIAGONAL_DR_4x4_MODE
- VERTICAL_RIGHT_4x4_MODE
- HORIZONTAL_DOWN_4x4_MODE
- VERTICAL_LEFT_4x4_MODE
- HORIZONTAL_UP_4x4_MODE



7.2.3.1.3 Chroma 8x8 prediction

For chroma prediction, 4 modes are available :

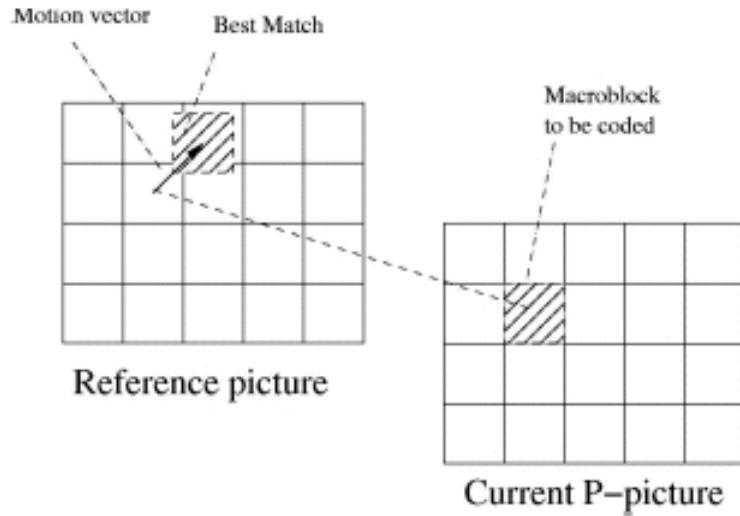
- DC
- HORIZONTAL
- VERTICAL
- PLANE

Those modes are equivalent to luma intra 16x16 except for DC which is slightly different. Please refer to H.264 specification for more details.

7.2.3.2 P Frame

While I frame performs a spatial prediction, P frames make predictions based on the previous encoded frames.

For each macroblock, a reference is found in the previous frame by looking around the current position. The motion vector is the distance between the reference in the previous picture and the current macroblock to be encoded. The best reference is subtracted from the current macroblock to form the residual data. The motion vector will be transmitted in the data stream so that decoder could rebuild the frame.



The motion vector has a pixelic precision for luma component and half pixel precision for chroma component due to chroma subsampling. Therefore Chroma needs to be interpolated to access sub pixels (refer to h.264 specification). Today, P264 doesn't allow macroblock fragmentation for motion estimation. Only one motion vector is computed for the entire 16x16 macroblock. The reference frame is always the previous encoded/decoded frame.

7.2.3.3 Residual data

Once intra/inter prediction is done, it is subtracted from the current macroblock. The residual data is then processed with the next scheme :

step 1: split the residual macroblock into 16 4x4 luma blocks and 4 4x4 chroma block for each chroma component

step 2: apply pseudo dct 4x4 on each 4x4 block

step 3: quantize all 4x4 blocks

step 4: if current macroblock is encoded using a luma 16x16 prediction, collect all DC coefficients of each 4x4 luma block and apply an hadamard transformation (see h.264 spec)

step 5: for each chroma component collect the 4 chroma DC values and performs an 2x2 hadamard transform (see h.264 spec)

step 6: zigzag all AC blocks

step 7: entropy encoding

Note: step 1-6 are exactly the same for P264 and H264.

In fact in intra 4x4 coding, for each 4x4 block, the intra prediction is determined first then the residual 4x4 block is processed from step 1 to step 3. Then the 4x4 block is reconstructed in order to have the correct neighbouring pixels for the next 4x4 block intra prediction.

The order for luma (Y) and chroma (C) 4x4 block encoding is resume here :

Y0	Y1	Y4	Y5	C0	C1
Y2	Y3	Y6	Y7	C2	C3
Y8	Y9	Y12	Y13		
Y10	Y11	Y14	Y15		

7.2.4 Specific block entropy-encoding

The proprietary format used to encode blocks is based on a mix of RLE and Huffman coding (cf. http://en.wikipedia.org/wiki/Run-length_encoding and http://en.wikipedia.org/wiki/Huffman_coding).

To resume, the RLE encoding is used to optimize the many zero values of the list, and the Huffman encoding is used to optimize the non-zero values.

Below will be shown the pre-defined sets of codewords ("dictionaries"), for RLE and Huffman coding. Then, the process description and an example.

Note: For UVLC codec, the first value of the list (the "DC value") is not compressed, but 16 to 10 bits encoded. That's not the case in P264.

Coarse	Additionnal	Size	Value of run	Range	Length of run
1		1	0	0	1
0 1		2	1	1	1
0 0 1	x	4	(x) + 2	2 : 3	2
0 0 0 1	xx	6	(xx) + 4	4 : 7	3
0 0 0 0 1	xxx	8	(xxx) + 8	8 : 15	4
0 0 0 0 0 1	xxxx	10	(xxxx) + 16	16 : 31	5
0 0 0 0 0 0 1	xxxxx	12	(xxxxx) + 32	32 : 63	6

Coarse	Additionnal	Size	Value of run	Range	Length of run
1	s	2	1		1
0 1		2	0 or EOB		
0 0 1	xs	5	(x) + 2	±2 : 3	2
0 0 0 1	xxs	7	(xx) + 4	±4 : 7	3
0 0 0 0 1	xxx s	9	(xxx) + 8	±8 : 15	4
0 0 0 0 0 1	xxxx s	11	(xxxx) + 16	±16 : 31	5
0 0 0 0 0 0 1	xxxxx s	13	(xxxxx) + 32	±32 : 63	6
0 0 0 0 0 0 1	xxxxx s	15	(xxxxx) + 64	±64 : 127	7

Note: s is the sign value (0 if datum is positive, 0 otherwise.)

7.2.4.1 Entropy-encoding process

Encoding principle :

The main principle to compress the values is to form a list of pairs of encoded-data. The first datum indicates the number of successive zero values (from 0 to 63 times). The second one corresponds to a non-zero Huffman-encoded value (from 1 to 127), with its sign.

Compression process :

The process to compress the "ZZ-list" (cf. Figure 13) in the output stream could be resumed in few steps:

- Direct copy of the 10-significant bits of the first 16-bits datum ("DC value") (**only for UVLC codec**)

- Initialize the counter of successive zero-values at zero.
- For each of the remaining 16-bits values of the list:
 - If the current value is zero:
 - * Increment the zero-counter
 - Else:
 - * Encode the zero-counter value as explained below :
 - Use the RLE dictionary (cf. Figure 14) to find the corresponding range of the value (ex: 6 is in the 4 : 7 range).
 - Subtract the low value of the range (ex: $6 - 4 = 2$)
 - Set this temporary value in binary format (ex: $2_{(10)} = 10_{(2)}$)
 - Get the corresponding "coarse" binary value (ex: $6_{(10)} \rightarrow 0001_{(2)}$)
 - Merge it with the temporary previously computed value (ex: $0001_{(2)} + 10_{(2)} \rightarrow 000110_{(2)}$)
 - * Add this value to the output stream
 - * Set the zero-counter to zero
 - * Encode the non-zero value as explain below :
 - Separate the value in temporary absolute part a , and sign part s . ($s = 0$ if datum is positive, 1 otherwise). Ex: for $d = -13 \rightarrow a = 13$ and $s = 1$.
 - Use the Huffman dictionary (cf. Figure 15) to find the corresponding range of a (ex: 13 is in the 8 : 15 range).
 - Subtract the lower bound (ex : $13 - 8 = 5$)
 - Set this temporary value in binary format (ex : $5_{(10)} = 101_{(2)}$)
 - Get the corresponding *coarse* binary value (ex : $5 \rightarrow 00001_{(2)}$)
 - Merge it with the temporary previously computed value, and the sign (ex : $00001_{(2)} + 101_{(2)} + 1_{(2)} \rightarrow 000011011_{(2)}$)
 - * Add this value to the output stream
 - Get to the next value of the list
- (End of "For")

7.2.4.2 Entropy-decoding process

The process to retrieve the "ZZ-list" from the compressed binary data is detailed here :

- Direct copy of the first 10 bits in a 16-bits datum ("DC value"), and add it to the output list. **(only for UVLC codec)**
- While there remains compressed data (till the "EOB" code):
 - Reading of the zero-counter value as explain below:
 - * Read the *coarseS* pattern part (bit-per-bit, till there is 1 value).
 - * On the corresponding line (cf. Figure 14), get the number of complementary bits to read. (Ex: $000001_{(2)} \rightarrow xxxx \rightarrow 4$ more bits to read.)
 - * If there is no 0 before the 1 (first case in the RLE table): \Rightarrow Resulting value (zero-counter) is equal to 0.

- * Else: \Rightarrow Resulting value (zero-counter) is equal to the direct decimal conversion of the merged read binary values. Ex: if $xxxx = 1101_{(2)} \rightarrow 000001_{(2)} + 1101_{(2)} = 0000011101_{(2)} = 29_{(10)}$
- Add "0" to the output list, as many times indicated by the zero-counter.
- Reading of the non-zero value as explained below:
 - * Read the *coarse* pattern part (bit-per-bit, till there is 1 value).
 - * On the corresponding line (cf. Figure 15), get the number of complementary bits to read. Ex: $0001_{(2)} \rightarrow xxs \rightarrow$ 2 more bits to read (then the sign bit.)
 - * If there is no 0 before the 1 (*coarse* pattern part = 1, in the first case of the Huffman table): \Rightarrow Temporary value is equal to 1.
 - * Else if the *coarse* pattern part = $01_{(2)}$ (second case of the Huffman table) : \Rightarrow Temporary value is equal to *End Of Bloc* code (**EOB**).
 - * Else \Rightarrow Temporary value is equal to the direct decimal conversion of the merged read binary values. Ex: if $xx = 11 \rightarrow 00001_{(2)} + 11_{(2)} = 0000111_{(2)} = 7_{(10)}$ Read the next bit, to get the sign (*s*).
 - * If *s* = 0: \Rightarrow Resulting non-zero value = temporary value
 - * Else (*s* = 1): \Rightarrow
 - * Resulting non-zero value = temporary value $\times (-1)$
- Add the resulting non-zero value to the output list.
- (End of "while")

7.2.4.3 Example

Encoding :

- Initial data list :
-26; -3; 0; 0; 0; 7; -5; EOB
- Step 1 :
-26; 0x"0"; -3; 4x"0"; 7; 0x"0"; -5; 0x"0"; EOB
- Step 2 (binary form):
111111111100110; 1; 001 11; 0001 00; 0001 110; 1; 0001 011; 1; 01
- Final stream :
1111100110100111000100000111010001011101

Decoding :

- Initial bit-data stream :
{11110001110111000110001010010100001010001101}
- Step 1 (first 10 bits split) :
{1111000111}; {0111000110001010010100001010001101}
- Step 2 (16-bits conversion of DC value) :
{1111111111000111}; {0111000110001010010100001010001101}

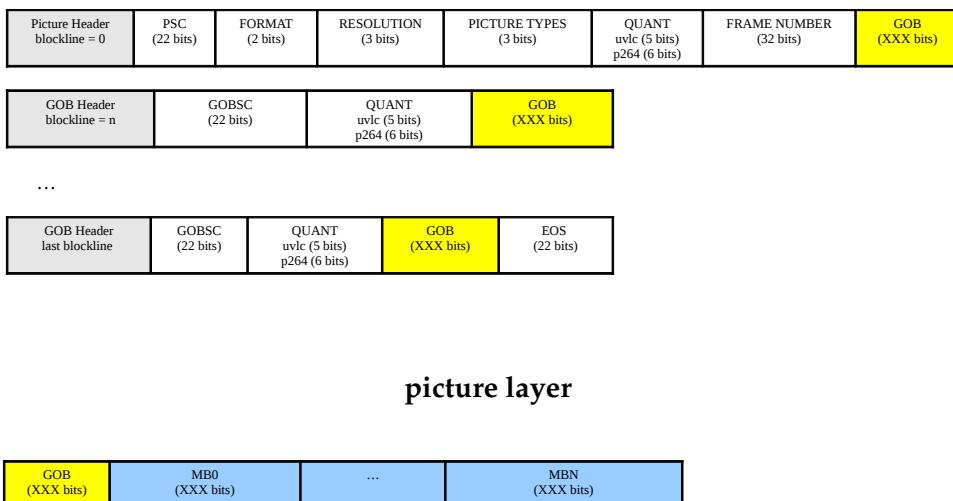
- Step 3, remaining data (DC value is done) :
 $\{-57\}; \{011100011000101001010001100110101\}$
- Step 4, first couple of values:
 $\{-57\}; [\{01.\}; \{1.1\}]; \{00011000101001010001100110101\}$
 $\{-57\}; ["0"; "-1"]; \{00011000101001010001100110101\}$
- Step 5, second couple of values :
 $\{-57"; "0"; "-1"; [\{0001.10\}; \{001.01\}]; \{001010001100110101\}$
 $\{-57"; "0"; "-1"; ["000000"; "-2"]; \{001010001100110101\}$
- Step 6, third couple of values :
 $\{-57"; "0"; "-1"; "000000"; "-2"; [\{001.0\}; \{1.0\}]; \{001100110101\}$
 $\{-57"; "0"; "-1"; "000000"; "-2"; "+1"; ["000"; "+3"]; \{101\}$
- Step 7, fourth couple of values :
 $\{-57"; "0"; "-1"; "000000"; "-2"; "+1"; [\{001.1\}; \{001.10\}]; \{101\}$
 $\{-57"; "0"; "-1"; "000000"; "-2"; "+1"; ["000"; "+3"]; \{101\}$
- Step 8, last couple of values (no "0" and "EOB" value):
 $\{-57"; "0"; "-1"; "000000"; "-2"; "+1"; "000"; "+3"; [\{1.\}; \{01\}]$
 $\{-57"; "0"; "-1"; "000000"; "-2"; "+1"; "000"; "+3"; [""; "EOB"]$
- Final data list :
 $\{-57"; "0"; "-1"; "0"; "0"; "0"; "0"; "0"; "0"; "-2"; "+1"; "0"; "0"; "+3"; "EOB"\}$

7.2.5 Transport layer

This section describes how the final data stream is generated.

For each picture, data correspond to an image header followed by data blocks groups and an ending code (EOS, end of sequence).

The composition of each block-layer is resumed here:



gob layer

MBx (XXX bits)	MBC (1 bit)	MBDES (8 bits)	MBDIFF (2 bits) (reserved)	Y0 (XXX bits)	Y1 (XXX bits)	Y2 (XXX bits)	Y3 (XXX bits)	U0 (XXX bits)	V0 (XXX bits)
-------------------	----------------	-------------------	----------------------------------	------------------	------------------	------------------	------------------	------------------	------------------

uvlc macroblock layer

Intra MBx (XXX bits)	INTRA LUMA TYPE (1 bit)	INTRA CHROMA TYPE (2 bits)	INTRABR 4x4	INTRABR 16x16	INTRABR 4x4 MODE (16*XXX bits)	Y0 (XXX bits)	...	Y15 (XXX bits)	CHROMA DATA
-------------------------	----------------------------	-------------------------------	----------------	------------------	-----------------------------------	------------------	-----	-------------------	-------------

Inter MBx (XXX bits)	PARTITION LIST (3 bits)	MOTION VECTOR LIST (XXX bits)	Y0 (XXX bits)	...	Y15 (XXX bits)	CHROMA DATA
-------------------------	----------------------------	----------------------------------	------------------	-----	-------------------	-------------

CHROMA DATA	DC U (XXX bits)	AC U0 (XXX bits)	...	AC U3 (XXX bits)	DC V (XXX bits)	AC V0 (XXX bits)	...	AC V3 (XXX bits)
-------------	--------------------	---------------------	-----	---------------------	--------------------	---------------------	-----	---------------------

p264 macroblock layer

7.2.5.1 Picture start code (PSC) (22 bits)

UVLC start with a PSC (Picture start code) which is 22 bits long:

0000 0000 0000 0000 1 00000

P264 PSC is:

0000 0000 0000 0001 0 00000

A PSC is always byte aligned.

7.2.5.2 Picture format (PFORMAT) (2 bits)

The second information is the picture format which can be one of the following : CIF or VGA

- 00 : forbidden
- 01 : CIF
- 10 : VGA

7.2.5.3 Picture resolution (PRESOLUTION) (3 bits)

Picture resolution which is used in combination with the picture format (3 bits)

- 000 : forbidden
- 001 : for CIF it means sub-QCIF
- 010 : for CIF it means QCIF
- 011 : for CIF it means CIF

- 100 : for CIF it means 4-CIF
- 101 : for CIF it means 16-CIF

7.2.5.4 Picture type (PTYPE) (3 bits)

Picture type:

- 000 : INTRA picture
- 001 : INTER picture

7.2.5.5 Picture quantizer (PQUANT) (5/6 bits)

UVLC codec: The PQUANT code is a 5-bits-long word. The quantizer's reference for the picture that range from 1 to 30.

P264 codec: The PQUANT code is a 6-bits-long word and range from 0 to 63;

7.2.5.6 Picture frame (PFRAME) (32 bits)

The frame number (32 bits).

7.2.5.7 Group of block start code (GOBSC) (22 bits)

Each GOB starts with a GOBSC (Group of block start code) which is 22 bits long:

uvlc codec :

0000 0000 0000 0000 1xxx xx

p264 codec :

0000 0000 0000 0001 0xxx xx

A GOBSC is always a byte aligned. The least significant bytes represent the blockline's number. We can see that PSC means first GOB too. So for the first GOB, GOB's header is always omitted.

7.2.5.8 Group of block quantizer (GOBQUANT) (5/6 bits)

Equivalent to PQUANT for the current GOB.

7.2.5.9 UVLC Macroblocks Layer

Data for each macroblock corresponding to an header of macroblock followed by data of macroblock.

MBx (XXX bits)	MBC (1 bit)	MBDES (8 bits)	MBDIFF (2 bits) (reserved)	Y0 (XXX bits)	Y1 (XXX bits)	Y2 (XXX bits)	Y3 (XXX bits)	U0 (XXX bits)	V0 (XXX bits)
-------------------	----------------	-------------------	----------------------------------	------------------	------------------	------------------	------------------	------------------	------------------

uvlc macroblock layer

MBC - Coded macroblock bit:

- Bit 0 : '1' means there's a macroblock / '0' means macroblock is all zero.
- If MBC is 0, the following fields are omitted.

MBDES - Macroblock description code:

- Bit 0 : '1' means there's non dc coefficients for block y0.
- Bit 1 : '1' means there's non dc coefficients for block y1.
- Bit 2 : '1' means there's non dc coefficients for block y2.
- Bit 3 : '1' means there's non dc coefficients for block y3.
- Bit 4 : '1' means there's non dc coefficients for block cb.
- Bit 5 : '1' means there's non dc coefficients for block cr.
- Bit 6 : '1' means there's a differential quantization (MBDIFF) value following this code.
Not implemented, always 0
- Bit 7 : Always '1' to avoid a zero byte.

MBDIFF – differential quantization: Not implemented

Y0-Y3 U0 V0: Each block Yi, U and V are encoded using the method described in section 4.x.x Block Entropy-encoding process.

7.2.5.10 P264 Macroblock Layer

Intra MBx (XXX bits)	INTRA LUMA TYPE (1 bit)	INTRA CHROMA TYPE (2 bits)	INTRA 4x4	INTRA LUMA 4x4 MODE (16*XXX bits)	Y0 (XXX bits)	...	Y15 (XXX bits)		CHROMA DATA
			INTRA 16x16	INTRA LUMA 16x16 MODE (2 bits)	DC Y (XXX bits)	AC Y0 (XXX bits)	...	AC Y15 (XXX bits)	

Inter MBx (XXX bits)	PARTITION LIST (3 bits)	MOTION VECTOR LIST (XXX bits)	Y0 (XXX bits)	...	Y15 (XXX bits)	CHROMA DATA
-------------------------	----------------------------	----------------------------------	------------------	-----	-------------------	-------------

CHROMA DATA	DC U (XXX bits)	AC U0 (XXX bits)	...	AC U3 (XXX bits)	DC V (XXX bits)	AC V0 (XXX bits)	..	AC V3 (XXX bits)
-------------	--------------------	---------------------	-----	---------------------	--------------------	---------------------	----	---------------------

p264 macroblock layer

There are 3 types of Macroblock in the transport layer :

- I frame with intra 16x16 prediction for the current macroblock
- I frame with intra 4x4 prediction for the current macroblock
- P frame

Macroblock intra 16x16:

INTRA LUMA TYPE – fragmentation used for intra prediction:

Bit 0 : '0' means intra 4x4, '1' means intra 16x16. Thus INTRA LUMA TYPE is set to 1 for an intra 16x16 Macroblock

INTRA CHROMA TYPE – intra mode for chroma component:

One of the four available intra chroma predictions coded over 2 (bits).

INTRA LUMA 16x16 MODE – 16x16 intra mode for luma component:

One of the four available intra chroma predictions coded over 2 (bits).

Y0 – Y15 – luma 4x4 blocks:

Each block (16 elements) is encoded using the method described in section *Block Entropy-encoding* process.

CHROMA DATA – U and V blocks:

This segment is common to all types of macroblock. See description below.

Macroblock intra 4x4:

INTRA LUMA TYPE – fragmentation used for intra prediction:

Bit 0 : '0' means intra 4x4, '1' means intra 16x16. Thus INTRA LUMA TYPE is set to 0 for a intra 4x4 Macroblock

INTRA CHROMA TYPE – intra mode for chroma component:

One of the four available intra chroma prediction coded over 2 (bits).

INTRA LUMA 4x4 MODE - list of 16 intra 4x4 prediction:

Each intra 4x4 is one of the nine available intra 4x4 luma prediction (horizontal, vertical, vertical up, ...)

Each element of the list is coded using a prediction based on the neighbouring predictions. If the prediction is correct, the element is coded using only 1 bit. If the prediction is false 4 bits are used. Please refer to h264 specification for details about how the prediction is done on intra 4x4 mode.

DC Y – list of 16 DC value:

DC Y is a list of 16 elements which gather DC values from the 16 4x4 blocks of the current macroblock. This list is written in the data stream using the block-encoding method.

AC Yx – block of AC coeff:

Each AC block (15 elements) is encoded with the block-encoding method.

CHROMA DATA – U and V blocks:

This segment is common to all type of macroblocks. See description below.

Inter Macroblock :

PARTITION LIST – list of mb subdivision for motion estimation:
Always read as '000' because P264 doesn't support macroblock partition.

MOTION VECTOR LIST – list of motion vector associated to each partition:

There is only one motion vector per macroblock. The vector is not put in the stream directly. A predicted motion vector for the current macroblock is determined with the already transmitted neighboring motion vector. The difference between the prediction and the real motion vector is written in the data stream.

The x component is transmitted before the y. Each component is written with the level-encoding method (see block-encoding). For further details about the way prediction is determined please refer to h.264 specification.

Y0 – Y15 – luma 4x4 blocks:

Each block (16 elements) is encoded using the method described in section *Block Entropy-encoding* process.

CHROMA DATA – U and V blocks:

This segment is common to all type of macroblocks. See description below.

Chroma Data:

DC U – list of 4 DC value:

DC U is a list which contains the DC values from each chroma 4x4 block. This list is encoded with the block-encoding method.

AC UX - block of AC coeff:

Each AC block (15 elements) is encoded with the block-encoding method.

DC V:

Same as DC U

AC VX:

Same as AC UX

7.2.6 End of sequence (EOS) (22 bits)

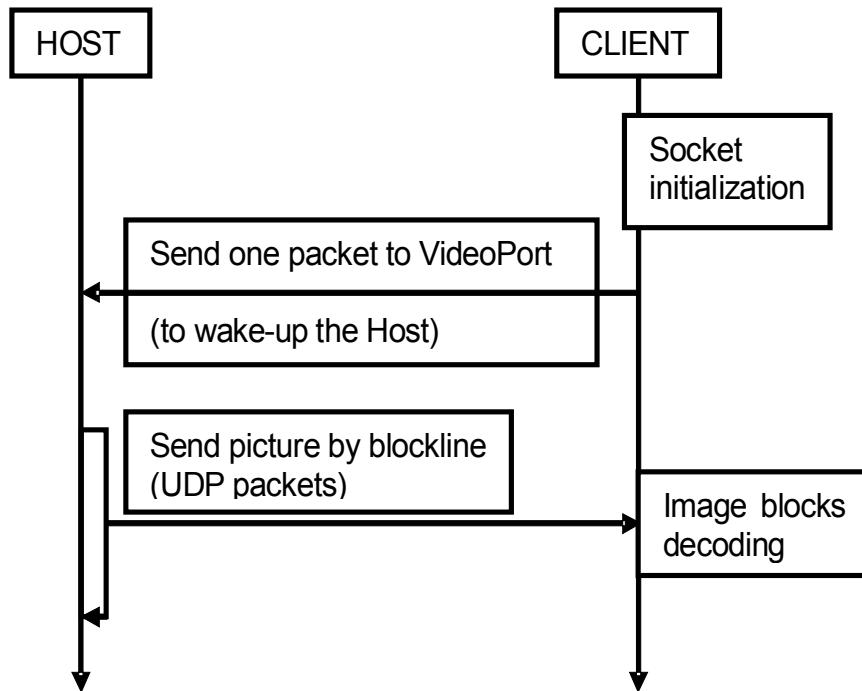
The end of sequence (EOS) which is 22 bits long :

0000 0000 0000 0001 0111 11

7.2.7 Initiating the video stream

To start receiving the video stream, a client just needs to send a UDP packet on the drone video port.

The drone will stop sending data if it cannot detect any network activity from its client.



7.3 The AR.Drone 2.0 video stream

The AR.Drone 2.0 use standard video codecs, with a custom encapsulation for managing the network stream

7.3.1 Video codecs

AR.Drone 2.0 use H264 (MPEG4.10 AVC) baseline profile for high quality video streaming and video recording.

The following parameters can be adjusted for the live H264 stream :

- FPS : Between 15 and 30
- Bitrate : Between 250kbps and 4Mbps
- Resolution : 360p (640x360) or 720p (1280*720)

Typical values for some Apple devices are :

- iPhone 4S : 360p, 30FPS, 4Mbps
- iPhone 4 : 360p, 25FPS, 1.5Mbps
- iPhone 3GS : 360p, 15FPS, 500kbps

These parameters are fixed to 720p, 30FPS, 4Mbps for the record stream, regardless of the device.

While recording, the hardware H264 encoder is no longer available to the live stream, thus the AR.Drone 2.0 uses a software MPEG4.2 Visual encoder for the live stream.

The following parameters can be adjusted for the live MPEG4.2 stream :

- FPS : Between 15 and 30
- Bitrate : Between 250kbps and 1Mbps

For further details about the video codecs, see [Wikipedia page](#) about MPEG-4 standard.

7.3.2 Video encapsulation on network

For network transmission, video frames are sent with custom headers, which contains many informations about the frame.

The headers are called PaVE (Parrot Video Encapsulation), and are described in listing 7.3

Listing 7.3: PaVE definition

```

typedef struct {
    uint8_t signature[4];                                /* "PaVE" - used to identify the start of
    frame */
    uint8_t version;                                    /* Version code */
    uint8_t video_codec;                               /* Codec of the following frame */
    uint16_t header_size;                             /* Size of the parrot_video_encapsulation_t
    */
    uint32_t payload_size;                            /* Amount of data following this PaVE */
    uint16_t encoded_stream_width;                   /* ex: 640 */
    uint16_t encoded_stream_height;                  /* ex: 368 */
    uint16_t display_width;                           /* ex: 640 */
    uint16_t display_height;                          /* ex: 360 */
    uint32_t frame_number;                            /* Frame position inside the current stream
    */
    uint32_t timestamp;                              /* In milliseconds */
    uint8_t total_chuncks;                           /* Number of UDP packets containing the
    current decodable payload - currently unused */
    uint8_t chunck_index;                            /* Position of the packet - first chunk is #0
    - currently unused*/
    uint8_t frame_type;                             /* I-frame, P-frame -
    parrot_video_encapsulation_frametypes_t */
    uint8_t control;                                /* Special commands like end-of-stream or
    advertised frames */
    uint32_t stream_byte_position_lw;               /* Byte position of the current payload in
    the encoded stream - lower 32-bit word */
    uint32_t stream_byte_position_uw;               /* Byte position of the current payload in
    the encoded stream - upper 32-bit word */
    uint16_t stream_id;                            /* This ID indentifies packets that should be
    recorded together */
    uint8_t total_slices;                           /* number of slices composing the current
    frame */
    uint8_t slice_index;                            /* position of the current slice in the frame
    */
    uint8_t header1_size;                           /* H.264 only : size of SPS inside payload -
    no SPS present if value is zero */
    uint8_t header2_size;                           /* H.264 only : size of PPS inside payload -
    no PPS present if value is zero */
    uint8_t reserved2[2];                           /* Padding to align on 48 bytes */
    uint32_t advertised_size;                      /* Size of frames announced as advertised
    frames */
    uint8_t reserved3[12];                          /* Padding to align on 64 bytes */
} __attribute__ ((packed)) parrot_video_encapsulation_t;

```

7.3.3 Network transmission of video stream

AR.Drone 2.0 video stream is transmitted on TCP socket 5555. AR.Drone 2.0 will start sending frame immediatly when a client connects to the socket.

A frame can be sent in multiple TCP packets, and thus should be reassembled by the application before feeding the video decoder. In **ARDroneTool**, this is done within the *Video/video_stage_tcp.c* file.

7.3.4 Latency reduction mechanism

The TCP transmission allow the application to receive all frames from the live stream, but this can introduce latency.

A latency reduction mechanism is implemented inside the *Video/video_stage_tcp.c* file. This mechanism will automatically select the most recent decodable frame to send to the decoder, and discard any older frame.

The same algorithm is also implemented on AR.Drone 2.0 side, before transmitting data on the network.

Listing 7.4 describes the system as a pseudo-code algorithm.

Listing 7.4: Latency reduction system

```
if (I-Frame available on buffer)
{
    send most recent I-Frame
}
else
{
    send next P-Frame
}
```

This mechanism is disabled on record stream, and the AR.Drone 2.0 buffer is larger, allowing approx. 30 seconds of buffering. This avoid frame loss on record stream, where latency is not an issue.

7.3.5 Video record stream

Video recording uses TCP socket 5553 to transmit H264-720p frames. This stream is disabled when the application is not recording.

This stream uses the same transmission encapsulation (PaVE) as the live stream.

The conversion between the raw H264 stream and the .mov/.mp4 file is done by the *Video/video_stage_encoded_recorder.c* file. With use of the *utils/ardrone_video_atoms* and *utils/ardrone_video_encapsuler* utilities.



The drone behaviour depends on many parameters which can be modified by using the [AT*CONFIG](#) AT command, or by using the appropriate **ARDroneTool** macro **ARDRONE_TOOL_CONFIGURATION_ADDEVENT**.

This chapter shows how to read/write a configuration parameter, and gives the list of parameters you can use in your application.

8.1 Reading the drone configuration

8.1.1 With **ARDroneTool**

ARDroneTool implements a 'control' thread which automatically retrieves the drone configuration at startup.

Include the `<ardrone_tool/ardrone_tool_configuration.h>` file in your C code to access the `ardrone_control_config` structure which contains the current drone configuration. Its most interesting fields are described in the next section.

If your application is structured as recommended in chapter [5](#) or you are modifying one of the examples, the configuration should be retrieved by **ARDroneTool** before the threads containing your code get started by **ARDroneTool**.

8.1.2 Without **ARDroneTool**

The drone configuration parameters can be retrieved by sending the AT*CTRL command with a mode parameter equaling 4 (**CFG_GET_CONTROL_MODE**).

The drone then sends the content of its configuration file, containing all the available configuration parameters, on the control communication port (TCP port 5559). Parameters are sent as ASCII strings, with the format *Parameter_name = Parameter_value*.

Here is an example of the sent configuration :

Listing 8.1: Example of configuration file as sent on the control TCP port

```

general:num_version_config          = 1
general:num_version_mb             = 33
general:num_version_soft           = 2.1.18
general:drone_serial               = XXXXXXXXXXXX
general:soft_build_date            = 2012-04-06 12:09
general:motor1_soft                = 1.41
general:motor1_hard                = 5.0
general:motor1_supplier             = 1.1
general:motor2_soft                = 1.41
general:motor2_hard                = 5.0
general:motor2_supplier             = 1.1
general:motor3_soft                = 1.41
general:motor3_hard                = 5.0
general:motor3_supplier             = 1.1
general:motor4_soft                = 1.41
general:motor4_hard                = 5.0
general:motor4_supplier             = 1.1
general:ardrone_name               = My ARDrone
general:flying_time                 = 758
general:navdata_demo                = TRUE
general:navdata_options              = 105971713
general:com_watchdog                  = 2
general:video_enable                  = TRUE
general:vision_enable                 = TRUE
general:vbat_min                     = 9000
control:accs_offset
    +03 }                                = { -2.0952554e+03 2.0413781e+03 2.0569382e
control:accs_gains
    -02 -2.0475569e-03 -9.9886459e-01 -9.5556228e-04 2.9887848e-03 -1.9088354e-02
    -9.8093420
    e-01 }
control:gyros_offset
    e+00 }                                = { -3.8548752e+01 -1.0268125e+02 -4.3712502
control:gyros_gains
    -03 }                                = { 1.0711575e-03 -1.0726772e-03 -1.0692523e
control:gyros110_offset
control:gyros110_gains
control:magneto_offset
    +02 }                                = { 1.6625000e+03 1.6625000e+03 }
control:magneto_radius
control:gyro_offset_thr_x
control:gyro_offset_thr_y
control:gyro_offset_thr_z
control:pwm_ref_gyros
control:osctun_value
control:osctun_test
control:altitude_max
control:altitude_min
control:control_level
control:euler_angle_max
control:control_iphone_tilt
control:control_vz_max
control:control_yaw
control:outdoor
control:flight_without_shell
control:autonomous_flight
control:manual_trim
control:indoor_euler_angle_max
control:indoor_control_vz_max
control:indoor_control_yaw
control:outdoor_euler_angle_max
    = 1.3417094e+02
    = 4.0000000e+00
    = 4.0000000e+00
    = 5.0000000e-01
    = 500
    = 63
    = TRUE
    = 3000
    = 50
    = 0
    = 2.0943952e-01
    = 3.4906584e-01
    = 7.0000000e+02
    = 1.7453293e+00
    = FALSE
    = FALSE
    = FALSE
    = FALSE
    = 2.0943952e-01
    = 7.0000000e+02
    = 1.7453293e+00
    = 3.4906584e-01

```

```

control:outdoor_control_vz_max          = 1.0000000e+03
control:outdoor_control_yaw           = 3.4906585e+00
control:flying_mode                   = 0
control:hovering_range               = 1000
control:flight_anim                 = 0,0
network:ssid_single_player          = ardrone2_XXXX
network:ssid_multi_player           = ardrone2_XXXX
network:wifi_mode                   = 0
network:wifi_rate                  = 0
network:owner_mac                  = 00:00:00:00:00:00
pic:ultrasound_freq                = 8
pic:ultrasound_watchdog           = 3
pic:pic_version                   = 184877088
video:camif_fps                    = 30
video:codec_fps                     = 30
video:camif_buffers                = 2
video:num_trackers                = 12
video:video_codec                 = 0
video:video_slices                 = 0
video:video_live_socket            = 0
video:video_storage_space          = 15360
video:bitrate                       = 1000
video:max_bitrate                  = 4000
video:bitrate_ctrl_mode            = 0
video:bitrate_storage              = 4000
video:video_channel                = 0
video:video_on_usb                 = TRUE
video:video_file_index             = 1
leds:leds_anim                     = 0,0,0
detect:enemy_colors                = 1
detect:groundstripe_colors         = 16
detect:enemy_without_shell          = 0
detect:detect_type                 = 3
detect:detections_select_h         = 0
detect:detections_select_v_hsync   = 0
detect:detections_select_v         = 0
syslog:output                      = 7
syslog:max_size                    = 102400
syslog:nb_files                     = 5
userbox:userbox_cmd                = 0
gps:latitude                        = 5.000000000000000e+02
gps:longitude                       = 5.000000000000000e+02
gps:altitude                         = 0.000000000000000e+00
custom:application_id              = 00000000
custom:application_desc             = Default application configuration
custom:profile_id                  = 00000000
custom:profile_desc                 = Default profile configuration
custom:session_id                  = 00000000
custom:session_desc                 = Default session configuration

```

8.2 Setting the drone configuration

8.2.1 With ARDroneTool

Use the **ARDRONE_TOOL_CONFIGURATION_ADDEVENT** macro to set any of the configuration parameters.

This macro makes **ARDroneTool** queue the new value in an internal buffer, send it to the drone, and wait for an acknowledgment from the drone.

It takes as a first parameter the name of the parameter (see next sections to get a list or look at the *config_keys.h* file in the SDK). The second parameter is a pointer to the new value to send to the drone. The third parameter is a callback function that will be called upon completion.

The callback function type is *void (*callback)(unsigned int success)*. The configuration tool will call the callback function after any attempt to set the configuration, with zero as the parameter in case of failure, and one in case of success. In case of failure, the tool will automatically retry after an amount of time.

Your program must not launch a new **ARDRONE_TOOL_CONFIGURATION_ADDEVENT** call at each time the callback function is called with zero as its parameter.

The callback function pointer can be a NULL pointer if your application don't need any acknowledgment.

Listing 8.2: Example of setting a config. parameter

```
// Set SSID name of AR.Drone
char ssid_name[] = "My_ARDron";
ARDRONE_TOOL_CONFIGURATION_ADDEVENT(ssid_single_player, \emph{ssid_name}, NULL); //  
WARNING : Don't put pointer of string, but just string.

// Set color of enemy hull.
int enemy_color;
enemy_color = ORANGE_GREEN;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT(enemy_colors, \emph{&enemy_color}, NULL);
```

8.2.2 From the Control Engine for iPhone

Using the Control Engine for iPhone, the ARDrone instance of your application can accept messages to control some parts of the configuration.

The message is -(void) executeCommandIn:(ARDRONE_COMMAND_IN_WITH_PARAM)commandIn fromSender:(id)sender refreshSettings:(BOOL)refresh.

The **ARDRONE_COMMAND_IN_WITH_PARAM** structure (declared in *ARDroneTypes.h*) is composed of the following members :

- **ARDRONE_COMMAND_IN** command : This is the same parameter as the legacy "exe-

cuteCommandIn" message (see below for details).

- command_in_configuration_callback callback : This is a callback that will be called upon configuration completion. If your application does not need the callback, you may put this field to *NULL*.
- void *parameter : Parameter to the command (described below).

ARDRONE_COMMAND_IN enum contains the following members (In parenthesis, the pointer type of the parameter associated) :

- ARDRONE_COMMAND_ISCLIENT : For multiplayer games, set different ultrasound frequencies for server and client (integer casted to (void *)).
- ARDRONE_COMMAND_DRONE_ANIM : Play a flight anim (ARDRONE_ANIMATION_PARAM pointer, see *ARDroneTypes.h* and *config.h*).
- ARDRONE_COMMAND_DRONE_LED_ANIM : Play a LED animation (ARDRONE_LED_ANIMATION_PARAM pointer, see *ARDroneTypes.h* and *led_animations.h*).
- ARDRONE_COMMAND_SET_CONFIG : Set any config key value (ARDRONE_CONFIG_PARAM pointer, see *ARDroneTypes.h*).
- ARDRONE_COMMAND_ENABLE_COMBINED_YAW : Enable (Disable) the combined yaw mode. (boolean casted to (void *)).

The second parameter is currently unused (passing *nil* is ok).

The third parameter is a boolean that tells the control engine to refresh the settings menu upon configuration completion. This can take nearly a second (request of configuration file of the AR.Drone), so you should not activate this when not needed.

Listing 8.3: Example of setting outdoor flight with Control Engine

```
ARDRONE\COMMAND\_IN\_WITH\_PARAM command;
ARDRONE\CONFIG\_PARAM param;
int value;

value = 1; // Will be treated as a boolean

param.config\_key = ARDRONE\CONFIG\_KEY\_OUTDOOR;
param.value = (void *)&value;

command.command = ARDRONE\COMMAND\_SET\_CONFIG;
command.callback = NULL;
command.parameter = (void *)&param;

[ardrone executeCommandIn:command fromSender:nil refreshSettings:NO];
```

Note : Using multiconfiguration, the AR.Drone will ignore any configuration request when not setted to the correct appli/user/session profiles. To reflect that, we've added an *isInit* field to the ARDroneNavigationData structure. Your application must ensure that the *isInit* flag is set (value will be 1) before sending any configuration. (Note that this flag CAN go back to zero during flight when a WiFi disconnection is detected).

Note : The legacy (1.6 SDK) *executeCommandIn* message is still supported in 2.0 SDK, but is flagged as deprecated and may be removed in future versions.

8.2.3 Without **ARDroneTool**

Use the **AT*CONFIG** command to send a configuration value to the drone. The command must be sent with a correct sequence number, the parameter note between double-quotes, and the parameter value between double-quotes.

8.3 Multiconfiguration

Starting with firmware 1.6.4, the AR.Drone supports different configurations depending on the application, user and session connected.

This allow different applications and users to share the same AR.Drone with different settings without any user action nor any "send all configurations" at application startup.

Configuration keys are split into 4 categories :

- CAT_COMMON : This is the default category, common to all applications.
- CAT_APPLI : This setting will be saved for the current application (regardless of the device it is running on).
- CAT_USER : This setting will be saved for the current user (regardless of the application). For more information about users, see the [user](#) section of this documentation. (User category is also called "profile" category).
- CAT_SESSION : This setting will be saved for the current flight session (regardless of application/user).

The legacy */data/config.ini* file contains the configuration for unidentified applications, and all CAT_COMMON settings. All identified applications settings are stored into */data/custom.config/* folder, under subfolders *applis*, *profiles* and *sessions*, with the name *config.<id>.ini*.

Note : When the AR.Drone 1.0 detects a disconnection, it will automatically switch back to the default configuration, and get back in bootstrap mode (no navdata are sent). This behaviour is only on AR.Drone 1.0 .

Note : As the session holds the application and user identifiers (CAT_SESSION settings), switching back to the session id will automatically put the drone back into the correct application/user. (See the [session](#) documentation for further informations).

8.3.1 With **ARDroneTool**

The *ardrone_tool_init* function now takes two string pointers to the application name and the user name that will be used as the application defaults.

The session name is a random generated number, and can not be specified from the application.

This function set an *ardrone_config_t* structure called *ardrone_application_default_config* which will hold the default configuration needed by your application. These configuration will be sent to the AR.Drone when the associate configuration are created. (Note : CAT_COMMON configs are never sent).

This allow users to overwrite your default settings, because your application will only send them once, and not at each startup.

Note : All description above refers to the *NO_ARDRONE_MAINLOOP* version of the *ardrone_tool_init* function. This is the reference mode, and the legacy **ARDroneTool** main loop should

not be used anymore. The legacy function only sets the application name according to `argv[0]` and init the `ardrone_application_default_config` structure.

8.3.2 Multiconfiguration with Control Engine (iPhone only)

The Control Engine init function (ARDrone object init) calls the `ardrone_tool_init_custom` passing the bundle name as the application name (e.g. "com.parrot.freeflight" for AR.FreeFlight), and the device type/UDID as the user.

(e.g. ".iPhone:0123456789ABCDEF", or ".iPod Touch:ABCDEF0123456789")

Note : The dot ('.') at the beginning of the username is explained in the [user section documentation](#).

To set the default configuration, the ARDrone class supports a new message called `-(void) setDefaultConfigurationForKey:(ARDRONE_CONFIG_KEYS)key withValue:(void *)value`.

The first parameter is the name of the configuration key (found in *ARDroneGeneratedTypes.h* file). The second parameter is a valid pointer to the value that you want to set as default. Pointer types for each keys can be found in the *config_keys.h* file.

Note : The default configuration must be set AFTER creating the ARDrone instance (after connecting to the AR.Drone), but BEFORE changing its state to "inGame".

8.3.3 Without ARDroneTool

When running on the default configuration, the drone use the standard behaviour, and does not require any aditionnal informations.

When running inside of any new configuration (application, user, session or any combination of theses), the AR.Drone expect to receive a new `AT*CONFIG_IDS` command before each `AT*CONFIG`. The AR.Drone will only set the configuration if the `AT*CONFIG_IDS` identifiers match the currently loaded configuration. In any case, the configuration will be acknowledged to the application.

8.3.4 Common category (CAT_COMMON)

This category holds the config keys that are common to all applications and users. These keys should not be initialized by your application, as they are shared between all.

This category holds, for example, all the version numbers, network configuration, and indoor/outdoor settings.

8.3.5 Application category (CAT_APPLI)

This category hold all the application specific configuration. Currently, this is limited to the video encoding options and the navdata_options required by your application.

8.3.6 User category (CAT_USER) – also called "Profile" category

This is the most complex category as different applications may want to access the same user profiles, so application must be able to switch their active user at runtime.

The **ARDroneTool** provides a few functions to do that. These functions are described in the *ardrone_api.h* file.

By convention, we use the dot ('.') as an "hidden user" identifier (like hidden files in UNIX systems). These user are default users that should not be shown to the consumer (e.g. default user for each iDevice, default user for a predefined flying mode ...), in regard to the "real" users that people will create on their drone to share their settings between many applications (including games). This convention is used inside the *ardrone_get_user_list* function to build a "visible user list" rather than a complete list.

Note : Known issues : As 1.8 version of AR.FreeFlight, user switch is not implemented, and thus partially untested. Changing the user (using *ardrone_api.h* functions) will cause an immediate change of the application identifier, but the queuing of the associated **AT*CONFIG**. In result, any queued config at this time may be refused by the AR.Drone before the effective config switch (see [here](#) for details).

8.3.7 Session category (CAT_SESSION)

The session category holds the "current flight session" settings. These settings includes the active video camera, the active video detection ...

This allow your application to get back to the exact same state after a drone reboot (battery change), or WiFi disconnection. In either cases, the drone is put back to the default configuration, and in bootstrap mode (no navdatas). To reconnect, your application just need to send back its session id (this will automatically put back the AR.Drone into the good application/user profiles), then ask again for navdata_demo and navdata_options keys.

See *ardrone_general_navdata.c* for an example.

Note : All application are expected to clean all sessions file at startup (see *ardrone_general_navdata.c*), so you may not rely on sessions when your application was shut down for any reason.

8.3.8 Technical details on id generation and descriptions

The **ARDroneTool** generates the different ids the following way :

The application id is generated as a CRC32 of a string composed of the application name and the SDK current version (e.g. for AR.FreeFlight 1.8, the complete string is "com.parrot.freeflight:1.7"). The application description is set as the application name only (e.g. "com.parrot.freeflight" for AR.FreeFlight 1.8)

The user id is generated as a CRC32 of the username provided to the **ARDroneTool**. The description is set as the username.

The session id is randomly generated. The session description is set as "Session <number>" with <number> the generated id.

Note : The CRC32 function used to generate the ids is NOT collision free. It's impossible for the AR.Drone to make the difference between two applications if their ids are the same.

8.4 General configuration

All the configurations are given accordingly to the *config_keys.h* file order.

In the API examples, the *myCallback* argument can be a valid pointer to a callback function, or a NULL pointer if no callback is needed by the application.

GENERAL:num_version_config _____ CAT_COMMON | Read only

Description :

Version of the configuration subsystem (Currently 1).

GENERAL:num_version_mb _____ CAT_COMMON | Read only

Description :

Hardware version of the drone motherboard.

GENERAL:num_version_soft _____ CAT_COMMON | Read only

Description :

Firmware version of the drone.

GENERAL:drone_serial _____ CAT_COMMON | Read only

Description :

Serial number of the drone.

GENERAL:soft_build_date _____ CAT_COMMON | Read only

Description :

Date of drone firmware compilation.

GENERAL:motor1_soft _____ CAT_COMMON | Read only

Description :

Software version of the motor 1 board. Also exists for motor2, motor3 and motor4.

GENERAL:motor1_hard _____ CAT_COMMON | Read only

Description :

Harware version of the motor 1 board. Also exists for motor2, motor3 and motor4.

GENERAL:motor1_supplier _____ CAT_COMMON | Read only

Description :

Supplier version of the motor 1 board. Also exists for motor2, motor3 and motor4.

GENERAL:ardrone_name

CAT_COMMON | Read/Write

Description :

Name of the AR.Drone. This name may be used by video games developper to assign a default name to a player, and is not related to the Wi-Fi SSID name.

AT command example : AT***CONFIG=605,"general:ardrone_name","My ARDrone Name"**

API use example :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (ardrone_name, "My ARDrone Name", myCallback);
```

GENERAL:flying_time

CAT_COMMON | Read only

Description :

Numbers of seconds spent by the drone in a flying state in its whole lifetime.

GENERAL:navdata_demo

CAT_COMMON | Read/Write

Description :

The drone can either send a reduced set of navigation data (*navdata*) to its clients, or send all the available information which contain many debugging information that are useless for everyday flights.

If this parameter is set to TRUE, the reduced set is sent by the drone (this is the case in the AR.FreeFlight iPhone application).

If this parameter is set to FALSE, all the available data are sent (this is the case in the Linux example *ardrone-navigation*).

AT command example : AT***CONFIG=605,"general:navdata_demo","TRUE"**

API use example :

```
bool_t value = TRUE;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (navdata_demo, &value, myCallback);
```

GENERAL:navdata_options

CAT_APPLI | Read/Write

Description :

When using navdata_demo, this configuration allow the application to ask for others *navdata* packets. Most common example is the *default_navdata_options* macro defined in the *config_key.h* file. The full list of the possible *navdata* packets can be found in the *navdata_common.h* file.

AT command example : AT***CONFIG=605,"general:navdata_options","105971713"**

API use example :

```
uint32_t ndOptions = (NAVDATA_OPTION_MASK (NAVDATA_DEMO_TAG) |
NAVDATA_OPTION_MASK (NAVDATA_VISION_DETECT_TAG) |
NAVDATA_OPTION_MASK (NAVDATA_GAMES_TAG) |
NAVDATA_OPTION_MASK (NAVDATA_MAGNETO_TAG) |
NAVDATA_OPTION_MASK (NAVDATA_HDVIDEO_STREAM_TAG) |
NAVDATA_OPTION_MASK (NAVDATA_WIFI_TAG));
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (navdata_options, &ndOptions, myCallback);
```

GENERAL:com_watchdog

CAT_COMMON | Read/Write

Description :

Time the drone can wait without receiving any command from a client program. Beyond this delay, the drone will enter in a 'Com Watchdog triggered' state and hover on top a fixed point.

Note : This setting is currently disabled. The drone uses a fixed delay of 250 ms.

GENERAL:video_enable _____ *CAT_COMMON | Read/Write*

Description :

Reserved for future use. The default value is TRUE, setting it to FALSE can lead to unexpected behaviour.

GENERAL:vision_enable _____ *CAT_COMMON | Read/Write*

Description :

Reserved for future use. The default value is TRUE, setting it to FALSE can lead to unexpected behaviour.

Note : This setting is not related to the tag detection algorithms

GENERAL:vbat_min _____ *CAT_COMMON | Read only*

Description :

Minimum battery level before shutting down automatically the AR.Drone.

8.5 Control configuration

CONTROL:accs_offset _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations. AR.Drone accelerometers offsets.

CONTROL:accs_gains _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations. AR.Drone accelerometers gains.

CONTROL:gyros_offset _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations. AR.Drone gyrometers offsets.

CONTROL:gyros_gains _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations. AR.Drone gyrometers gains.

CONTROL:gyros110_offset _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations.

CONTROL:gyros110_gains _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations.

CONTROL:magneto_offset _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations.

CONTROL:magneto_radius _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations.

CONTROL:gyro_offset_thr_x _____ *CAT_COMMON | Read only*

Description :

Parrot internal debug informations.

Note : Also exists for the *y* and *z* axis.

CONTROL:pwm_ref_gyros _____ *CAT_COMMON | Read only***Description :**

Parrot internal debug informations.

CONTROL:osctun_value _____ *CAT_COMMON | Read only***Description :**

Parrot internal debug informations.

CONTROL:osctun_test _____ *CAT_COMMON | Read only***Description :**

Parrot internal debug informations.

CONTROL:control_level _____ *CAT_APPLI | Read/Write***Description :**

This configuration describes how the drone will interprete the progressive commands from the user.

Bit 0 is a global enable bit, and should be left active.

Bit 1 refers to a *combined yaw* mode, where the roll commands are used to generates roll+yaw based turns. This is intended to be an easier control mode for racing games.

Note : This configuration and the flags parameter of the *ardrone_at_set_progress_commands* function will be compared on the drone. To activate the combined yaw mode, you must set both the bit 1 of the control_level configuration, and the bit 1 of the function parameters.

The *ardrone_at_set_progress_commands* function parameter reflects the current user commands, and must be set only when the *combined_yaw* controls are activated (e.g. both buttons pressed)

This configuration should be left active on the AR.Drone if your application makes use of the *combined_yaw* functionnality.

Note : This configuration does not need to be changed to use the new Absolute Control mode. This mode is only enabled by the flag of the progressive command.

AT command example : **AT*CONFIG=605,"control:control_level","3"**

API use example :

```
ardrone_control_config.control_level |= (1 << CONTROL_LEVEL_COMBINED_YAW);
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (control_level, &(ardrone_control_config.control_level), myCallback);
```

CONTROL:euler_angle_max _____ *CAT_USER | Read/Write***Description :**

Maximum bending angle for the drone in radians, for both **pitch** and **roll** angles.

The **progressive command function** and its associated **AT command** refer to a percentage of this value. **Note :** For AR.Drone 2.0 , the new **progressive command function** is preferred (with the corresponding **AT command**).

This parameter is a positive floating-point value between 0 and 0.52 (ie. 30 deg). Higher values might be available on a specific drone but are not reliable and might not allow the drone to stay at the same altitude.

This value will be saved to **indoor/outdoor_euler_angle_max**, according to the **CONFIG:outdoor** setting.

AT command example : **AT*CONFIG=605,"control:euler_angle_max","0.25"**

API use example :

```
float eulerMax = 0.25;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (euler_angle_max, &eulerMax, myCallback);
```

CONTROL:altitude_max

CAT_COMMON | Read/Write

Description :

Maximum drone altitude in millimeters.

On AR.Drone 1.0 : Give an integer value between 500 and 5000 to prevent the drone from flying above this limit, or set it to 10000 to let the drone fly as high as desired. On AR.Drone 2.0 : Any value will be set as a maximum altitude, as the pressure sensor allow altitude measurement at any height. Typical value for "unlimited" altitude will be 100000 (100 meters from the ground)

AT command example : AT***CONFIG=605,"control:altitude_max","3000"**

API use example :

```
uint32_t altMax = 3000;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (altitude_max, &altMax, myCallback);
```

CONTROL:altitude_min

CAT_COMMON | Read/Write

Description :

Minimum drone altitude in millimeters.

Should be left to default value, for control stabilities issues.

AT command example : AT***CONFIG=605,"control:altitude_min","50"**

API use example :

```
uint32_t altMin = 50;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (altitude_min, &altMin, myCallback);
```

CONTROL:control_iphone_tilt

CAT_USER | Read/Write

Description :

The angle in radians for a full iPhone accelerometer command. This setting is stored and computed on the AR.Drone so an iPhone application can send progressive commands without taking this in account.

On AR.FreeFlight, the progressive command sent is between 0 and 1 for angles going from 0 to 90. With a *control_iphone_tilt* of 0.5 (approx 30), the drone will saturate the command at 0.33

Note : This settings corresponds to the *iPhone tilt max* setting of AR.FreeFlight

AT command example : AT***CONFIG=605,"control:control_iphone_tilt","0.25"**

API use example :

```
float iTiltMax = 0.25;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (control_iphone_tilt, &iTiltMax, myCallback);
```

CONTROL:control_vz_max

CAT_USER | Read/Write

Description :

Maximum vertical speed of the AR.Drone, in milimeters per second.

Recommended values goes from 200 to 2000. Others values may cause instability.

This value will be saved to indoor/outdoor_control_vz_max, according to the *CONFIG:outdoor* setting.

AT command example : AT***CONFIG=605,"control:control_vz_max","1000"**

API use example :

```
uint32_t vzMax = 1000;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (control_vz_max, &vzMax, myCallback);
```

CONTROL:control_yaw

CAT_USER | Read/Write

Description :

Maximum yaw speed of the AR.Drone, in radians per second.

Recommended values goes from 40/s to 350/s (approx 0.7rad/s to 6.11rad/s). Others values may cause instability.

This value will be saved to indoor/outdoor_control_yaw, according to the CONFIG:outdoor setting.

AT command example : AT*CONFIG=605,"control:control_yaw","3.0"

API use example :

```
float yawSpeed = 3.0;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (control_yaw, &yawSpeed, myCallback);
```

CONTROL:outdoor

CAT_COMMON | Read/Write

Description :

This settings tells the control loop that the AR.Drone is flying outside.

Setting the indoor/outdoor flight will load the corresponding indoor/outdoor_control_yaw, indoor/outdoor_euler_angle_max and indoor/outdoor_control_vz_max.

Note : This settings enables the wind estimator of the AR.Drone 2.0 , and thus should always be enabled when flying outside. *Note :* This settings corresponds to the *Outdoor flight* setting of AR.FreeFlight

AT command example : AT*CONFIG=605,"control:outdoor","TRUE"

API use example :

```
bool_t isOutdoor = TRUE;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (outdoor, &isOutdoor, myCallback);
```

CONTROL:flight_without_shell

CAT_COMMON | Read/Write

Description :

This settings tells the control loop that the AR.Drone is currently using the outdoor hull. Deactivate it when flying with the indoor hull

Note : This settings corresponds to the *outdoor hull* setting of AR.FreeFlight.

Note : This setting is not linked with the CONTROL:outdoor setting. They have different effects on the control loop.

AT command example : AT*CONFIG=605,"control:flight_without_shell","TRUE"

API use example :

```
bool_t withoutShell = TRUE;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (flight_without_shell, &withoutShell, myCallback);
```

CONTROL:autonomous_flight

CAT_COMMON | Read/Write

Description :

Deprecated : This setting enables the autonomous flight mode on the AR.Drone . This mode was developped for 2010 CES and is no longer maintained.

Enabling this can cause unexpected behaviour on commercial AR.Drone .

CONTROL:manual_trim _____ CAT_USER | Read only

Description :

This setting will be active if the drone is using manual trims. Manual trims should not be used on commercial AR.Drone , and this field should always be FALSE.

CONTROL:indoor_euler_angle_max _____ CAT_USER | Read/Write

Description :

This setting is used when *CONTROL:outdoor* is false. See the *CONTROL:euler_angle_max* description for further informations.

CONTROL:indoor_control_vz_max _____ CAT_USER | Read/Write

Description :

This setting is used when *CONTROL:outdoor* is false. See the *CONTROL:control_vz_max* description for further informations.

CONTROL:indoor_control_yaw _____ CAT_USER | Read/Write

Description :

This setting is used when *CONTROL:outdoor* is false. See the *CONTROL:control_yaw* description for further informations.

CONTROL:outdoor_euler_angle_max _____ CAT_USER | Read/Write

Description :

This setting is used when *CONTROL:outdoor* is true. See the *CONTROL:euler_angle_max* description for further informations.

CONTROL:outdoor_control_vz_max _____ CAT_USER | Read/Write

Description :

This setting is used when *CONTROL:outdoor* is true. See the *CONTROL:control_vz_max* description for further informations.

CONTROL:outdoor_control_yaw _____ CAT_USER | Read/Write

Description :

This setting is used when *CONTROL:outdoor* is true. See the *CONTROL:control_yaw* description for further informations.

CONTROL:flying_mode _____ CAT_SESSION | Read/Write

Description :

Since 1.5.1 firmware, the AR.Drone has two different flight modes. The first is the legacy FreeFlight mode, where the user controls the drone, an a new semi-autonomous mode, called "HOVER_ON_TOP_OF_ROUNDEL", where the drones will hover on top of a ground tag. This new flying mode was developped for 2011 CES autonomous demonstration. Since 2.0 and 1.10 firmwares, a third mode, called "HOVER_ON_TOP_OF_ORIENTED_ROUNDNEL", was added. This mode is the same as the previous one, except that the AR.Drone will always face the same direction.

For all modes, progressive commands are possible.

Note : Oriented Black&White Roundel detection must be activated with the *DETECT:detect_type* setting if you want to use the "HOVER_ON_TOP_OF_(ORIENTED_)ROUNDEL" mode.

Note : Enum with modes can be found in the *ardrone_api.h* file.

AT command example : AT***CONFIG=605**,"control:flying_mode","0"

API use example :

```
FLYING_MODE fMode = FLYING_MODE_FREE_FLIGHT;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (flying_mode, &fMode, myCallback);
```

CONTROL:hovering_range

CAT_SESSION | Read/Write

Description :

This setting is used when *CONTROL:flying_mode* is set to "HOVER_ON_TOP_OF_(ORIENTED_)ROUNDEL". It gives the AR.Drone the maximum distance (in millimeters) allowed between the AR.Drone and the oriented roundel.

CONTROL:flight_anim

CAT_COMMON | Read/Write

Description :

Use this setting to launch drone animations.

The parameter is a string containing the animation number and its duration, separated with a comma. Animation numbers can be found in the *config.h* file.

Note : The *MAYDAY_TIMEOUT* array contains defaults durations for each flight animations. **Note :** The *FLIP* animations are only available on AR.Drone 2.0

AT command example : AT***CONFIG=605**,"control:flight_anim","3,2"

API use example :

```
char param[20];
snprintf (param, sizeof (param), "%d,%d", ARDRONE_ANIMATION_FLIP_LEFT, MAYDAY_TIMEOUT[ARDRONE_ANIMATION_FLIP_LEFT]);
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (flight_anim, param, myCallback);
```

8.6 Network configuration

NETWORK:ssid_single_player

CAT_COMMON | Read/Write

Description :

The AR.Drone SSID. Changes are applied on reboot

AT command example : AT***CONFIG=605**,"network:ssid_single_player","myArdroneNetwork"

API use example :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (ssid_single_player, "myArdroneNetwork", myCallback);
```

NETWORK:ssid_multi_player

CAT_COMMON | Read/Write

Description :

Currently unused.

NETWORK:wifi_mode

CAT_COMMON | Read/Write

Description :

Mode of the Wi-Fi network. Possible values are :

- 0 : The drone is the access point of the network
- 1 : The drone creates (or join) the network in Ad-Hoc mode
- 2 : The drone tries to join the network as a station

Note : This value should not be changed for users applications.

NETWORK:wifi_rate

CAT_COMMON | Read/Write

Description :

Debug configuration that should not be modified.

NETWORK:owner_mac

CAT_COMMON | Read/Write

Description :

Mac address paired with the AR.Drone. Set to "00:00:00:00:00:00" to unpair the AR.Drone.

AT command example : AT***CONFIG=605**,"network:owner_mac","01:23:45:67:89:ab"

API use example :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (owner_mac, "cd:ef:01:23:45:67", myCallback);
```

8.7 Nav-board configuration

PIC:ultrasound_freq

CAT_COMMON | Read/Write

Description :

Frequency of the ultrasound measures for altitude. Using two different frequencies can reduce significantly the ultrasound perturbations between two AR.Drones.

Only two frequencies are available : 22.22 and 25 Hz.

The enum containing the values are found in the *ardrone_common_config.h* file.

AT command example : AT*CONFIG=605,"pic:ultrasound_freq","7"

API use example :

```
ADC_COMMANDS uFreq = ADC_CMD_SELECT_ULTRASOUND_22Hz;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (ultrasound_freq, &uFreq, myCallback);
```

PIC:ultrasound_watchdog

CAT_COMMON | Read/Write

Description :

Debug configuration that should not be modified.

PIC:pic_version

CAT_COMMON | Read only

Description :

The software version of the Nav-board.

8.8 Video configuration

VIDEO:camif_fps *CAT_COMMON | Read only*

Description :

Current FPS of the video interface. This may be different than the actual framerate.

VIDEO:codec_fps *CAT_SESSION | Read/Write*

Description :

Current FPS of the live video codec. Maximum value is 30. *Note :* Only effective on AR.Drone 2.0 .

AT command example : AT***CONFIG=605,"video:codec_fps","30"**

API use example :

```
uint32_t codecFps = 30;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (codec_fps, &codecFps, myCallback);
```

VIDEO:camif_buffers *CAT_COMMON | Read only*

Description :

Buffer depth for the video interface.

VIDEO:num_trackers *CAT_COMMON | Read only*

Description :

Number of tracking point for the speed estimation.

VIDEO:video_codec *CAT_SESSION | Default : Read only | Multiconfig : Read/Write*

Description :

Current video codec of the AR.Drone . Values differs for AR.Drone 1.0 and AR.Drone 2.0 .

Note : On AR.Drone 2.0 , this key controls the start/stop of the record stream.

Possible codec values for AR.Drone 2.0 are :

- MP4_360P_CODEC : Live stream with MPEG4.2 soft encoder. No record stream.
- H264_360P_CODEC : Live stream with H264 hardware encoder configured in 360p mode. No record stream.
- MP4_360P_H264_720P_CODEC : Live stream with MPEG4.2 soft encoder. Record stream with H264 hardware encoder in 720p mode.
- H264_720P_CODEC : Live stream with H264 hardware encoder configured in 720p mode. No record stream.
- MP4_360P_H264_360P_CODEC : Live stream with MPEG4.2 soft encoder. Record stream with H264 hardware encoder in 360p mode.

Possible codec values for AR.Drone 1.0 are :

- UVLC_CODEC : MJPEG-like codec.
- P264_CODEC : H264-like codec.

Note : Other codec values can lead to unexpected behaviour. *Note :* Enum with codec values can be found in the **VLIB/video_codec.h** file of the **ARDroneLIB** .

AT command example : AT***CONFIG=605,"video:video_codec","129"**

API use example :

```
codec_type_t newCodec = H264_360P_CODEC;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (video_codec, &newCodec, myCallback);
```

VIDEO:video_slices

CAT_SESSION | Read/Write

Description :*Debug configuration that should not be modified.***VIDEO:video_live_socket**

CAT_SESSION | Read/Write

Description :*Debug configuration that should not be modified.***VIDEO:video_storage_space**

CAT_COMMON | Read only

Description :

Size of the wifi video record buffer.

VIDEO:bitrate

CAT_APPLI | Default : Read only | Multiconfig : Read/Write

Description :

For AR.Drone 1.0 : When using the bitrate control mode in "VBC_MANUAL", sets the bitrate of the video transmission (size, in octets, of each frame).

Recommended values are 20000 for VLIB codec, and 15000 for P264 codec.

For AR.Drone 2.0 : When using the bitrate control mode in "VBC_MANUAL", sets the bitrate of the video transmission (kilobits per second).

Typical values range from 500 to 4000 kbps.

Note : This value is dynamically changed when bitrate_control_mode is set to VBC_MODE_DYNAMIC. **Note :** This configuration can't be changed on default configuration.

AT command example : AT*CONFIG=605,"video:bitrate","1000"**API use example :**

```
uint32_t newBitrate = 4000;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (bitrate, &newBitrate, myCallback);
```

VIDEO:max_bitrate

CAT_SESSION | Default : Read only | Multiconfig : Read/Write

Description :

AR.Drone 2.0 only.

Maximum bitrate that the device can decode. This is set as the upper bound for drone bitrate values.

Typical values for Apple iOS Device are :

- iPhone 4S : 4000 kbps
- iPhone 4 : 1500 kbps
- iPhone 3GS : 500 kbps

Note : When using the bitrate control mode in "VBC_MANUAL", this maximum bitrate is ignored. **Note :** When using the bitrate control mode in "VBC_MODE_DISABLED", the bitrate is fixed to this maximum bitrate.

AT command example : AT*CONFIG=605,"video:max_bitrate","1000"

API use example :

```
uint32_t newMaxBitrate = 4000;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (max_bitrate, &newMaxBitrate, myCallback);
```

VIDEO:bitrate_control_mode _____ CAT_APPLI | Default : Read only | Multiconfig : Read/Write**Description :**

Enables the automatic bitrate control of the video stream. Enabling this configuration will reduce the bandwidth used by the video stream under bad Wi-Fi conditions, reducing the commands latency.

Possible values are (see *ardrone_api.h*) :

	<i>AR.Drone 1.0</i>	<i>AR.Drone 2.0</i>
VBC_MODE_DISABLED	Don't use	Bitrate set to <i>video:max_bitrate</i>
VBC_MODE_DYNAMIC	Image sizes varies in [5000; 25000] bytes per frame	Video bitrate varies in [250; <i>video:max_bitrate</i>] kbps
VBC_MANUAL	Image size is fixed by the <i>video:bitrate</i> key	Video stream bitrate is fixed by the <i>video:bitrate</i> key

Note : This configuration can't be changed on default configuration.

AT command example : AT*CONFIG=605,"video:bitrate_control_mode","1"

API use example :

```
VIDEO_BITRATE_CONTROL_MODE vbcMode = VBC_MODE_DYNAMIC;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (bitrate_control_mode, &vbcMode, myCallback);
```

VIDEO:bitrate_storage _____ CAT_APPLI | Read/Write**Description :**

Only for AR.Drone 2.0 Bitrate (kbps) of the recording stream, both for USB and WiFi record.

Note : This value should not be changed by the user.

VIDEO:video1_channel _____ CAT_SESSION | Read/Write**Description :**

The video channel that will be sent to the controller.

Current implementation supports 4 different channels :

- ZAP_CHANNEL_HORI
- ZAP_CHANNEL_VERT
- ZAP_CHANNEL_LARGE_HORI_SMALL_VERT (AR.Drone 1.0 only)
- ZAP_CHANNEL_LARGE_VERT_SMALL_HORI (AR.Drone 1.0 only)

AT command example : AT*CONFIG=605,"video:video_channel","2"

API use example :

```
ZAP_VIDEO_CHANNEL nextChannel = ZAP_CHANNEL_HORI;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (video_channel, &nextChannel, myCallback);
```

VIDEO:video_on_usb _____ CAT_COMMON | Read/Write**Description :**

Only for AR.Drone 2.0 If this key is set to "TRUE" and a USB key with >100Mb of freespace is connected, the record video stream will be recorded on the USB key.

In all other cases (key set to "FALSE" or no USB key plugged), the record stream is sent to the controlling device, which will be in charge of the actual recording.

AT command example : AT***CONFIG=605,"video:video_on_usb","TRUE"**

API use example :

```
bool_t recordOnUsb = TRUE;  
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (video_on_usb, &recordOnUsb, myCallback);
```

VIDEO:video_file_index

CAT_COMMON | Read/Write

Description :

Only for AR.Drone 2.0 Number of the last recorded video (video_XXX.mp4) on USB key.

Note : Application should not write any value to this key.

8.9 Leds configuration

LEDS:leds_anim

CAT_COMMON | Read/Write

Description :

Use this setting to launch leds animations.

The parameter is a string containing the animation number, its frequency (Hz) and its duration (s), separated with commas. Animation names can be found in the *led_animation.h* file.

Note : The frequency parameter is a floating point parameter, but in configuration string it will be print as the binary equivalent integer.

AT command example : AT*CONFIG=605,"leds:leds_anim","3,1073741824,2"

API use example :

```
char param[50];
float frequency = 2.0;
snprintf (param, sizeof (param), "%d,%d,%d", ARDRONE_LED_ANIMATION_BLINK_ORANGE, *(unsigned
int *)&frequency, 5);
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (leds_anim, param, myCallback);
```

8.10 Detection configuration

DETECT:enemy_colors

CAT_COMMON | Read/Write

Description :

The color of the hulls you want to detect. Possible values are green, yellow and blue (respective integer values as of 2.1/1.10 firmware : 1, 2, 3).

Note : This config will only be used for standard tags/hulls detection. Roundel detection don't use it.

AT command example : AT*CONFIG=605,"detect:enemy_colors","2"

API use example :

```
ENEMY_COLORS_TYPE enemyColors = ARDRONE_DETECTION_COLOR_BLUE;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (enemy_colors, &enemyColors, myCallback);
```

DETECT:groundstripe_colors

CAT_SESSION | Read/Write

Description :

Only for AR.Drone 1.0 , with legacy groundstripe detection.

The color of the ground stripe you want to detect. Possible values are orange/green and, yellow/blue (respective integer values as of 1.6.4 firmware : 0x10, 0x11).

Note : This config will only be used for groundstripe detection.

AT command example : AT*CONFIG=605,"detect:groundstripe_colors","0x10"

API use example :

```
COLORS_DETECTION_TYPE color = ARDRONE_ENEMY_COLOR_ARRACE_FINISH_LINE;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (groundstripe_colors, &color, myCallback);
```

DETECT:enemy_without_shell

CAT_COMMON | Read/Write

Description :

Activate this in order to detect outdoor hulls. Deactivate to detect indoor hulls.

AT command example : AT*CONFIG=605,"detect:enemy_without_shell","1"

API use example :

```
uint32_t activated = 0;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (enemy_without_shell, &activated, myCallback);
```

DETECT:detect_type

CAT_SESSION | Read/Write

Description :

Select the active tag detection.

Possible values are (see *ardrone_api.h*) :

- CAD_TYPE_NONE : No detections.
- CAD_TYPE_MULTIPLE_DETECTION_MODE : See following note.
- CAD_TYPE_ORIENTED_COCARDE_BW : Black&White oriented roundel detected on bottom facing camera.
- CAD_TYPE_VISION_V2 : Standard tag detection (for both AR.Drone 2.0 and AR.Drone 1.0).

Any other values are either deprecated or in development.

Note : It is advised to enable the multiple detection mode, and then configure the detection needed using the following keys.

Note : The multiple detection mode allow the selection of different detections on each camera. Note that you should NEVER enable two similar detection on both cameras, as this will cause failures in the algorithms.

Note : The Black&White oriented roundel can be downloaded on [Parrot website](#)

AT command example : AT***CONFIG=605,"detect:detect_type","10"**

API use example :

```
CAD_TYPE detectType = CAD_TYPE_MULTIPLE_DETECTION_MODE;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (detect_type, &detectType, myCallback);
```

DETECT:detections_select_h CAT_SESSION | Read/Write

Description :

Bitfields to select detections that should be enabled on horizontal camera.

Possible tags values are (see *ardrone_api.h*) :

- TAG_TYPE_NONE : No tag to detect.
- TAG_TYPE_SHELL_TAG_V2 : Standard hulls (both indoor and outdoor) tags, for both AR.Drone 2.0 and AR.Drone 1.0 .
- TAG_TYPE_BLACK_ROUNDEL : Black&While oriented roundel.

All other value are either deprecated or in development.

Note : You should NEVER enable one detection on two different cameras.

AT command example : AT***CONFIG=605,"detect:detections_select_h","1"**

API use example :

```
uint32_t detectH = TAG_TYPE_MASK (TAG_TYPE_SHELL_TAG);
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (detections_select_h, &detectH, myCallback);
```

DETECT:detections_select_v_hsync CAT_SESSION | Read/Write

Description :

Bitfields to select the detections that should be enabled on vertical camera.

Detection enables in the hsync mode will run synchronously with the horizontal camera pipeline, a 30fps instead of 60. This can be useful to reduce the CPU load when you don't need a 60Hz detection.

Note : You should use either *v_hsync* or *v* detections, but not both at the same time. This can cause unexpected behaviours.

Note : Notes from DETECT:detections_select_h also applies.

AT command example : AT***CONFIG=605,"detect:detections_select_v_hsync","2"**

API use example :

```
uint32_t detectVhsync = TAG_TYPE_MASK (TAG_TYPE_ROUNDEL);
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (detections_select_v_hsync, &detectVhsync, myCallback);
```

DETECT:detections_select_v*CAT_SESSION | Read/Write***Description :**

Bitfields to select the detections that should be active on vertical camera.

These detections will be run at 60Hz. If you don't need that speed, using *detections_select_v_hsync* instead will reduce the CPU load.

Note : See the *DETECT:detections_select_h* and *DETECT:detections_select_v_hsync* for further details.

AT command example : AT***CONFIG=605,"detect:detections_select_v","2"**

API use example :

```
uint32_t detectV = TAG_TYPE_MASK (TAG_TYPE_ROUNDEL);  
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (detections_select_v, &detectV, myCallback);
```

8.11 SYSLOG section

This section is a Parrot internal debug section, and should therefore not be used.

8.12 USERBOX section

USERBOX:userbox_cmd

CAT_SESSION | Read/Write

Description :

The *USERBOX:userbox_cmd* provide a feature to save navigation data from drone during a time period and to take pictures. When the *USERBOX:userbox_cmd* is sent with parameter *USERBOX_CMD_START*, the AR.Drone create directory */data/video/boxes/tmp_flight_YYYYMMDD_hhmmss* in AR.Drone flash memory and save a binary file containing navigation data and named */data/video/boxes/tmp_flight_YYYYMMDD_hhmmss/userbox_<timestamp>*. *<timestamp>* represent the time since the AR.Drone booted.

When the *USERBOX:userbox_cmd* is sent with parameter *USERBOX_CMD_STOP*, the AR.Drone finish saving of */data/video/boxes/tmp_flight_YYYYMMDD_hhmmss/userbox_<timestamp>*. and rename directory from */data/video/boxes/tmp_flight_YYYYMMDD_hhmmss* to */data/video/boxes/flight_YYYYMMDD_hhmmss*.

When the *USERBOX:userbox_cmd* is sent with parameter *USERBOX_CMD_CANCEL*, the AR.Drone stop the userbox like sending *USERBOX_CMD_STOP* and delete file */data/video/boxes/tmp_flight_YYYYMMDD_hhmmss/userbox_<timestamp>*.

When the *USERBOX:userbox_cmd* is sent with parameter *USERBOX_CMD_SCREENSHOT*, the AR.Drone takes picture of frontal camera and saves it as JPEG image named */data/video/boxes/tmp_flight_YYYYMMDD_hhmmss/picture_YYMMDD_hhmmss.jpg*.

Note : If the userbox is started, the picture is saved in userbox directory. **Note :** After stopping userbox or taking picture, you MUST call *academy_download_resume* function to download flight directory by ftp protocol.

Typical values are :

- *USERBOX_CMD_STOP* : Command to stop userbox. This command takes no parameters.
- *USERBOX_CMD_CANCEL* : Command to cancel userbox. If the userbox is started, stop the userbox and delete its content.
- *USERBOX_CMD_START* : Command to start userbox. This command takes the current date as string parameter with format *YYYYMMDD_hhmmss*
- *USERBOX_CMD_SCREENSHOT* : Command to take a picture from AR.Drone . This command takes 3 parameters.
 - 1 - delay : This value is an unsigned integer representing the delay (in seconds) between each screenshot.
 - 1 - number of burst : This value is an unsigned integer representing the number of screenshot to take.
 - 1 - current date : This value is the current date as string parameter with format *YYYYMMDD_hhmmss*.

Note : The file */data/video/boxes/tmp_flight_YYYYMMDD_hhmmss/userbox_<timestamp>* will be used for future feature. (AR.Drone Academy).

AT command example : AT***CONFIG=605,"userbox:userbox_cmd","0"**

API use example :

```
char command[ARDRONE_DATE_MAX_SIZE + 64];
char date[ARDRONE_DATE_MAX_SIZE];
time_t t = time (NULL);
ardrone_time2date(t, ARDRONE_FILE_DATE_FORMAT, date);
snprintf (command, sizeof (command), "%d,%s", USERBOX_CMD_START, date);
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (userbox_cmd, command, myCallback);
```

8.13 GPS section

GPS:latitude

CAT_SESSION | Read/Write

Description :

GPS Latitude sent by the controlling device.

This data is used for media tagging and userbox recording.

Note : value is a double precision floating point number, sent as a the binary equivalent 64bit integer on AT command

AT command example : AT*CONFIG=605,"gps:latitude","4631107791820423168"

API use example :

```
double gpsLatitude = 42.0;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (latitude, &gpsLatitude, myCallback);
```

GPS:longitude

CAT_SESSION | Read/Write

Description :

GPS Longitude sent by the controlling device.

This data is used for media tagging and userbox recording.

Note : value is a double precision floating point number, sent as a the binary equivalent 64bit integer on AT command

AT command example : AT*CONFIG=605,"gps:longitude","4631107791820423168"

API use example :

```
double gpsLongitude = 42.0;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (longitude, &gpsLongitude, myCallback);
```

GPS:altitude

CAT_SESSION | Read/Write

Description :

GPS Altitude sent by the controlling device.

This data is used for media tagging and userbox recording.

Note : value is a double precision floating point number, sent as a the binary equivalent 64bit integer on AT command

AT command example : AT*CONFIG=605,"gps:altitude","4631107791820423168"

API use example :

```
double gpsAltitude = 42.0;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (altitude, &gpsAltitude, myCallback);
```

8.14 CUSTOM section - Multiconfig support

CUSTOM:application_id _____ *CAT_SESSION | Read/Write*

Description :

Sets the current application identifier (creates application settings if they don't exist, switch to application otherwise)

Note : To remove an application, put a "-" before its id (e.g. "-01234567"). *Note :* To remove all applications, use "-all" as id. *Note :* When removing a multiconfig setting, ensure that you won't remove any "other application" setting !

AT command example : AT***CONFIG=605,"custom:application_id","0a1b2c3d"**

CUSTOM:application_desc _____ *CAT_APPLI | Read/Write*

Description :

Sets the current application description string

AT command example : AT***CONFIG=605,"custom:application_desc","My Application Name"**

CUSTOM:profile_id _____ *CAT_SESSION | Read/Write*

Description :

Sets the current user identifier (creates user settings if they don't exist, switch to user otherwise) *Note :* To remove a profile, put a "-" before its id (e.g. "-01234567"). *Note :* To remove all profiles, use "-all" as id. *Note :* When removing a multiconfig setting, ensure that you won't remove any "other application" setting !

AT command example : AT***CONFIG=605,"custom:profile_id","0a1b2c3d"**

CUSTOM:profile_desc _____ *CAT_USER | Read/Write*

Description :

Sets the current user description string (typically user name). *Note :* "Hidden" users description starts with a dot ('.')

AT command example : AT***CONFIG=605,"custom:profile_desc","My Visible User"** **AT command example :** AT***CONFIG=605,"custom:profile_desc",".My Hidden User"**

CUSTOM:session_id _____ *CAT_SESSION | Read/Write*

Description :

Sets the current session identifier (creates session settings if they don't exist, switch to session otherwise) *Note :* To remove a session, put a "-" before its id (e.g. "-01234567"). *Note :* To remove all session, use "-all" as id. *Note :* When removing a multiconfig setting, ensure that you won't remove any "other application" setting !

AT command example : AT***CONFIG=605,"custom:session_id","0a1b2c3d"**

CUSTOM:session_desc _____ *CAT_SESSION | Read/Write*

Description :

Sets the current session description string.

AT command example : AT***CONFIG=605,"custom:session_desc","Session 0a1b2c3d"**

Part II

Tutorials



9

Building the iOS Example

The AR.Drone 2.0 SDK provides the full source code of AR.FreeFlight iPhone application.

To compile both the Control Engine and the AR.FreeFlight project, you need to use Apple XCode IDE on an Apple computer.

You also need to have an Apple developer account with associated Developer profiles.
Further informations can be found on [Apple website](#)

Note : The FreeFlight application can't run on iOS Simulator, as we are using armv7 optimized assembly code for the video decoder.



10

Building the Linux Examples

The AR.Drone 2.0 SDK provides three client application examples for Linux.

The first one, called *Linux SDK Demo*, shows the minimum software required to get AR.Drone 2.0 navigation datas, and is a base skeleton for more complex applications.

The second one, called *Linux Video Demo*, shows the minimum software required to receive and decode the video from both versions of AR.Drone 2.0 . It should be used as an example for all application that require live video.

The third one, called *ARDrone Navigation*, is the tool used internally at Parrot to test drones in flight. It shows all the information sent by the drone during a flight session.

In this section, we are going to show you how to build those example on an Ubuntu 10.04 workstation. We will suppose you are familiar with C programming and you already have a C compiler installed on your machine.

Note : These examples are not compatible with 64bits versions of Linux.

10.1 Set up your development environment

If you have not done so yet, please download the latest version of the SDK [here](#) (you must first register on the site).

Then unpack the archive *ARDrone_SDK_Version_X_X_<date>.tar.gz* in the directory of your choice. In this document we will note *<SDK>* the extracted directory name.

```
$ tar xzf ARDrone_SDK_Version_X_X_<date>.tar.gz
```

10.2 Compile linux examples

To compile all linux examples :

```
$ cd <SDK>/Examples/Linux/  
$ make
```

Executable programs will be created in the *<SDK>/Examples/Linux/Build/Release*

10.3 Run the SDK Demo program

Before running any demo, make sure that your computer is connected to your AR.Drone

You can test the connection with a ping command :

```
$ ping 192.168.1.1
```

If connection is successful, the ping command will return you the time needed for data to go back and forth to the drone.

You can then launch the demo program :

```
Setting locale to en_GB.UTF-8
Wait authentication [name="pitch_front" value="-2" type="1"/>
Wait authentication [name="pitch_back" value="2" type="1"/>
Wait authentication [name="roll_left" value="-1" type="1"/>
Wait authentication [name="roll_right" value="1" type="1"/>
=====+> 192.168.1.1
Starting thread video_stage [name="yaw_left" value="-4" type="1"/>
Set IP_TOS ok [name="yaw_right" value="4" type="1"/>
Starting thread navdata_update [name="speed_up" value="-5" type="1"/>
Starting thread ardrone_control [name="speed_down" value="5" type="1"/>
</controls>
    Video stage thread initialisation

Thread navdata_update in progress...
PA : MEMORY SPACE ALLOWED : 40 MB
Start thread thread_academy_download
Academy download stage resumed
Start thread thread_academy
Start thread thread_academy_upload
Academy download stage paused
Timeout when reading navdatas - resending a navdata request on port 5554
Academy download stage resumed
Academy download stage pausedions =====
Sending default CAT_APPLI settings=====
Sending default CAT_SESSION settings=====
=====2
Navdata for flight demonstrations ===== 10261.000
Battery level : 63 mVa] 167.000 [Phi] -1500.000 [Psi] 10261.000
Control state : 131072] 165.000 [Phi] -1501.000 [Psi] 10264.000
Battery level : 63 mV167.000 [vY] -1500.000 [vZPsi] 10261.000
Orientation : [Theta] 238.000 [Phi] -1513.000 [Psi] 10204.000
Altitude : 0
Speed : [vX] 238.000 [vY] -1513.000 [vZPsi] 10204.000
```

It will display various informations about the AR.Drone

Note that this demo does not include gamepad management code, so you won't be able to pilot the AR.Drone with it.

10.4 Run the Video Demo program

Before running any demo, make sure that your computer is connected to your AR.Drone

You can test the connection with a ping command :

```
$ ping 192.168.1.1
```

If connection is successful, the ping command will return you the time needed for data to go back and forth to the drone.

You can then launch the demo program : The program should launch a GTK2+Cairo window, displaying the drone front camera live streaming.

The demo program accepts two arguments :

- -c : Change the codec used by the AR.Drone . On AR.Drone 1.0 , it changes the codec from P264 to VLIB. On AR.Drone 2.0 , it changes the codec to h.264-360p to h.264-720p.
- -b : Use bottom camera instead of front camera.

Note that this demo does not include gamepad management code, so you won't be able to pilot the AR.Drone with it.

10.5 Run the *Navigation* program

Before running any demo, make sure that your computer is connected to your AR.Drone

You can test the connection with a ping command :

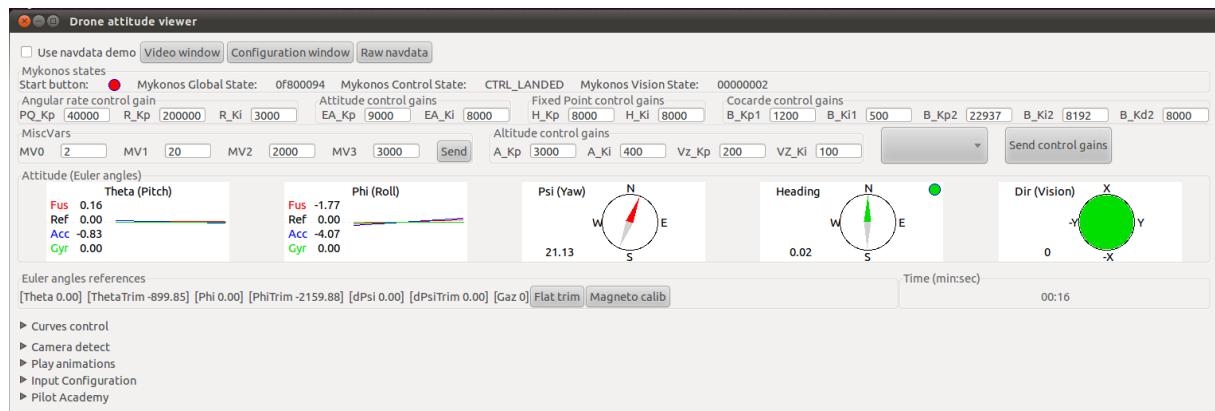
```
$ ping 192.168.1.1
```

If connection is successful, the ping command will return you the time needed for data to go back and forth to the drone.

You can then launch the navigation program :

What to expect ?

You should see a GUI showing the current drone state. If connection with the drone is successful, the central graphs entitled *theta,phi* and *psi* should change according to the drone angular position.



Configure your controller

To configure your controller, open the "Input Configuration" part, and click on the "USB Configuration" button. You should then see a window allowing you to select the controlling device, and to configure the keys.

Note : The keyboard piloting mode is not supported, even if the configuration windows seems to work. You'll have to use an USB gamepad.

Depending on the computers, it may not be possible to configure directly the controller from the software. In this case, you'll need to edit the `<SDK>/Examples/Linux/Build/Release/ardrone.xml` file.

ardrone.xml file structure

An example of the ardrone.xml file structure can be found on listing 10.1.

Listing 10.1: ardrone.xml device example

```
<?xml version="1.0"?>
<ardrone>
  <devices>
    <device id="74301981" name="Generic X-Box pad" default="yes">
      <controls>
        <control name="takeoff" value="6" type="3"/>
        <control name="emergency" value="10" type="3"/>
        <control name="pitch_front" value="-2" type="1"/>
        <control name="pitch_back" value="2" type="1"/>
        <control name="roll_left" value="-1" type="1"/>
        <control name="roll_right" value="1" type="1"/>
        <control name="yaw_left" value="-4" type="1"/>
        <control name="yaw_right" value="4" type="1"/>
        <control name="speed_up" value="-5" type="1"/>
        <control name="speed_down" value="5" type="1"/>
      </controls>
    </device>
    [...]
  </devices>
</ardrone>
```

The ID can be retrieved with lsusb tool. Its the integer value corresponding to the USB id of your device.

All other informations can be found with the jstest tool :

The type determines if the control is an analogic axis (1) or a button (3).

The value determines the axis/button which control this feature.

- Axis values are jstest's axis numbers +1 (i.e. jstest axis 2 will be "3" or "-3" in the xml file)
- Negative axis values indicates a negative value on this axis (e.g. -2 for forward, 2 for backward movement means that our axis takes negative values to the front)
- Button values are the same as jstest button numbers (i.e. jstest button 2 will be "2" in the xml file)

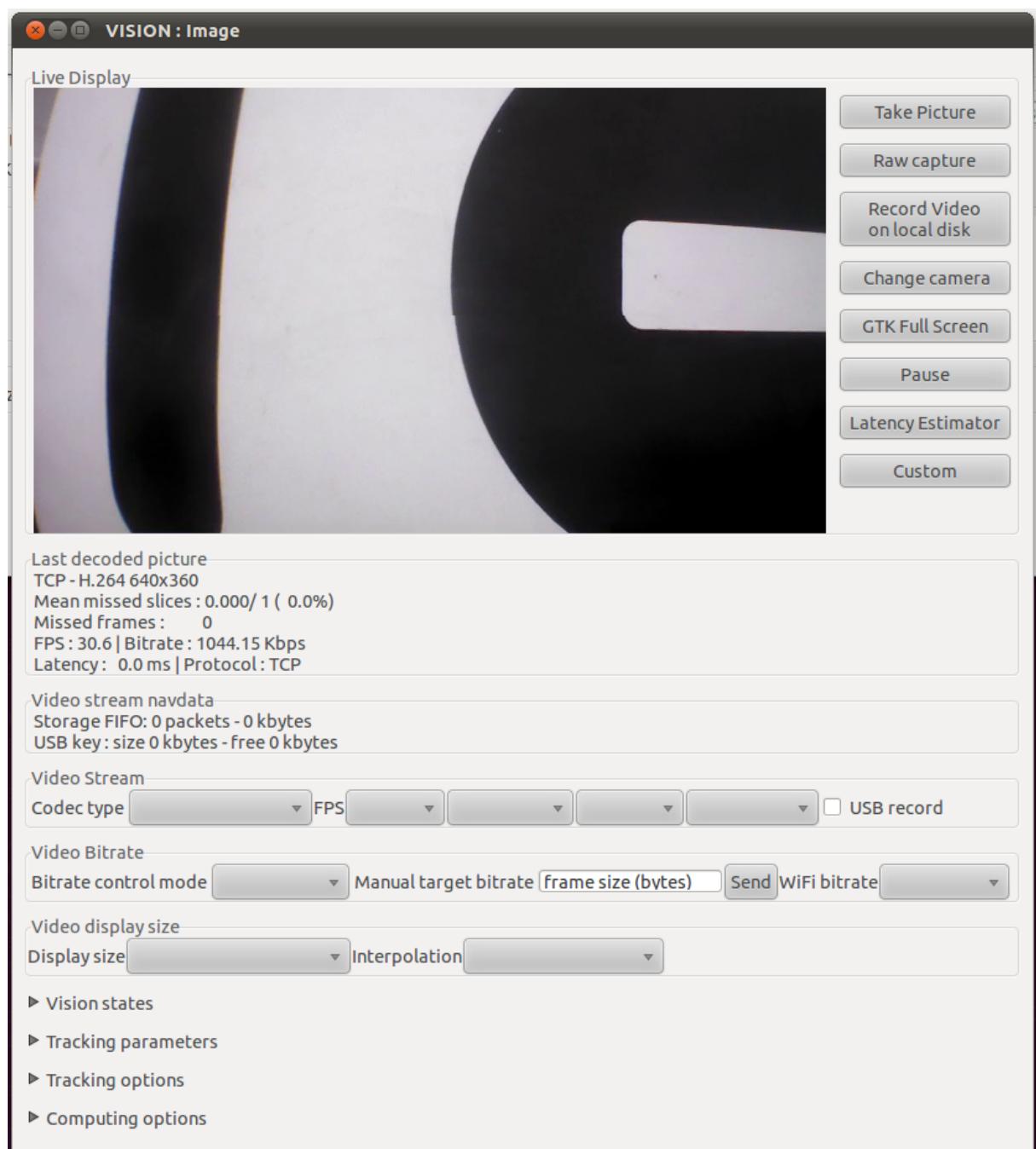
Fly

Now press the button you chose as the *select* button. Press it several times to make the motors leds switch between red (emergency mode) and green(ready to fly mode).

Clear the drone surroundings and press the button you chose as the *start* button. The drone should start flying.

Get video

Press the *Video Window* button to show the video from the AR.Drone .





11

Android example

The AR.Drone SDK now also provide the Android app example.

In this section, we are going to show you how to build this example app on an Ubuntu workstation.

11.1 Set up your development environment

Before compiling any Android example, you will need to download and install the latest Android SDK and NDK.

The example was written using SDK 4.1 (API Level 16), and NDK r8b.

The Android SDK can be found [here](#).

The Android NDK can be found [here](#).

Unzip both SDK and NDK files where you want. In this document, we will refer to this folder as `<ANDROID_SDK>`.

After this, add the following folders to your PATH. You can do this by editing the `~/.bashrc` file, and add the following line at the end:

```
PATH=$PATH:<ANDROID_SDK>/android-sdk-linux/tools:<ANDROID_SDK>/android-sdk-linux/platform-tools:<ANDROID_SDK>/android-ndk-r8b
```

Then run the following commands:

```
$ source ~/.bashrc  
$ android
```

The Android SDK Manager window should appear.

To build the AR.Drone Android example, you must select and install the following packets:

- Android SDK Tools (rev 20.0.3 or newer)
- Android SDK Platform Tools (rev 14 or newer)
- Android 4.1.x (API 16) SDK Platform
- (Optionnal) Android Support Library

You will also need to install *ant*. You can install it using the following command line:

```
$ sudo apt-get install ant
```

Last part is to edit the `<SDK>/Examples/Android/trunk/FreeFlight2/environment.properties` file. This file contains some environment variables that you will need to modify according to your android sdk/ndk installation:

- `ANDROID_SDK_PATH` : Path to the root folder of Android SDK (`<ANDROID_SDK>/android-sdk-linux`)
- `ANDROID_NDK_PATH` : Path to the root folder of Android NDK (`<ANDROID_SDK>/android-ndk-r8b`)

11.2 Building and installing the Android example

To build the android example, go to the `<SDK>/Examples/Android/trunk/FreeFlight2/` folder, and run the following script:

```
$ ./build.sh release
```

This will build the app in release mode. Other options to the script are:

- `release` > build the app in release mode (APK File : `bin/FreeFlight-release.apk`)
- `debug` > build the app in debug mode (APK File : `bin/FreeFlight-debug.apk`)
- `clean` > clean the app and the SDK

In either case, the app will be signed with a dummy release key. This means that:

- You'll need to uninstall the Google Play app before installing this one as the keys are not the same
- You can share the resulting APK with anyone as long as it is not published to an online app market (Google Play, Amazon marketplace ...)
- If you want to distribute a resulting app, you will need to generate (or reuse) your own release key

To install the app using adb, just type the following command:

```
$ adb install -r bin/FreeFlight-release.apk
```

11.3 Modifying the Android example source code

11.3.1 Modifying the ARDroneLib part

The ARDroneLib is ONLY compiled by running the `build.sh` script. If you do any modification to this part, you will need to rerun that script to rebuild the libraries.

11.3.2 Modifying the JNI part

The JNI part of the application is the link between the ARDroneLib (Native C code) and the UI (Android Java code). The JNI part source code can be found in the `jnil` folder inside the example folder.

To rebuild the final .so file for the app, you can either use the `build.sh` script, or simply run `ndk-build` from the root app folder.

Note : Using `ndk-build` will only work if `build.sh` had been run at least once

11.3.3 Modifying the UI part

The root folder of the example contains an eclipse project that you can import to an existing workspace as long as you don't copy the project into your workspace (relative path to ARDroneLib must be kept).

Note : To be able to run the application from ecplise, native parts of the project must have already been built using the *build.sh* file

The project was developped using the following eclipse configuration, but should work on newer version with minor changes:

- Eclipse version : Juno Classic 4.2.0
- Android ADT version : 20.0.3
- Additionnal plugins : CDT v8.1.0 (if you want to edit C code from eclipse)