Parrot

AR.Drone

THE FLYING VIDEO GAME ....

Developer Guide

SDK 1.6

Prepared	Title		
Stephane Piskorski Nicolas Brulez	AR.Drone Developer Guide		
Approved	Date February 24, 2011	Revision SDK 1.6	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`**.

Contents

Part I

SDK documentation

1 Introduction

Welcome to the AR.Drone Software Development Kit !

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

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

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

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

This SDK includes :

- this document explaining how to use the SDK, and describes the drone communications protocols;
- the AR.Drone Library (**`ARDroneLIB`**), which provides the APIs needed to easily communicate and configure an AR.Drone product;
- the AR.Drone Tool (**`ARDroneTool`**) library, which provides a fully functionnal drone client where developers only have to insert their custom application specific code;
- the AR.Drone Control Engine library which provides an intuitive control interface developed by Parrot for remotely controlling the AR.Drone product from an iPhone;

- an open-source iPhone game example, several code examples that show how to control the drone from a Linux or Windows personal computer, and a simple example for Android phones.

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 :

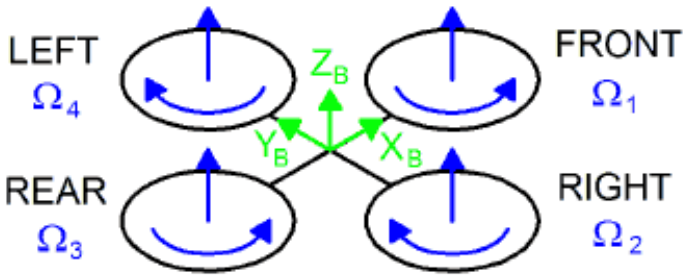
- using the provided library 5 and modifying the provided examples (10, 11, 12) to suit your needs
- trying to write your own software from scratch by following the specifications given in 6 and 7.

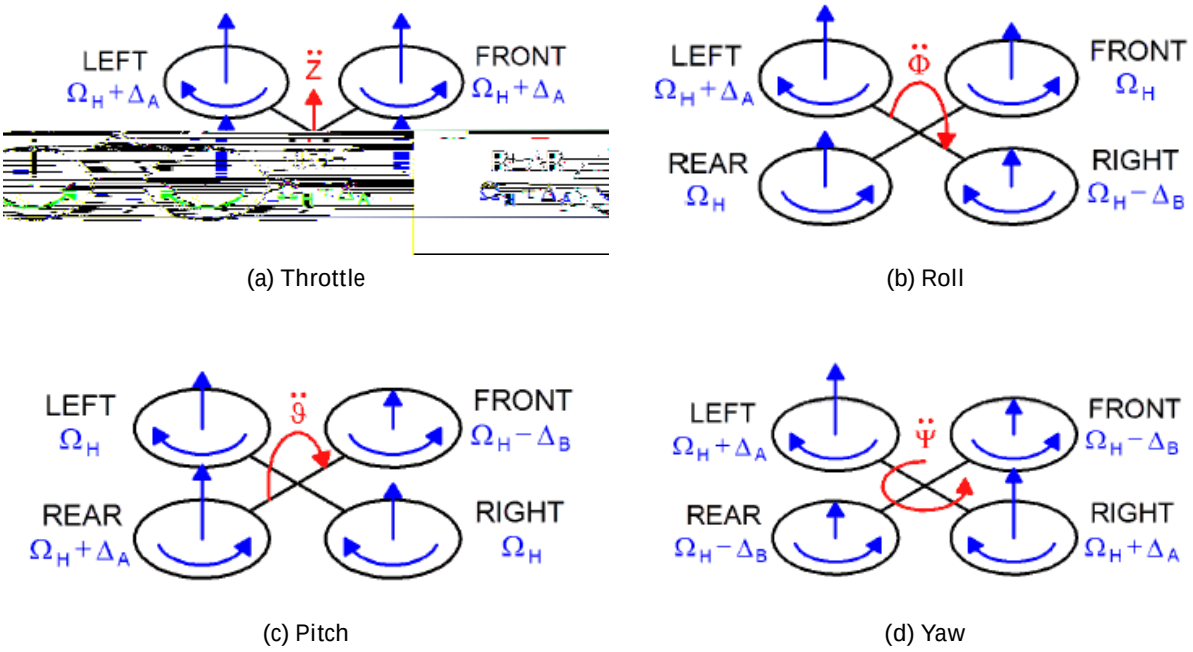
2 AR.Drone Overview

2.1 Introduction to quadrotor UAV

AR.Drone 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.



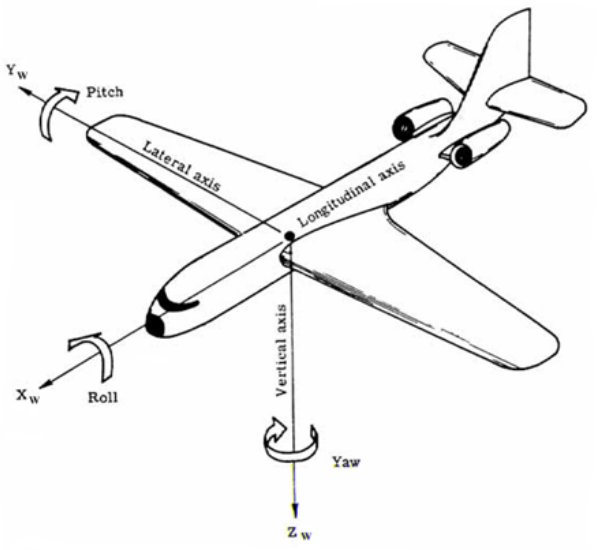


(a) Throttle (b) Roll

(c) Pitch (d) Yaw

Figure 2.1: Drone movements

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



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.

(a) Indoor (b) Outdoor

Figure 2.2: Drone hulls

2.2 Indoor and outdoor design configurations

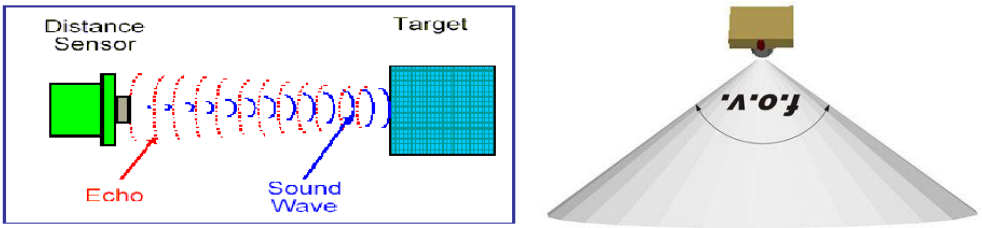
When flying outdoor the AR.Drone can be set in a light and low wind drag configuration (2.2b). Flying indoor requires the drone to be protected by external bumpers (2.2a).

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 is powered with brushless engines with three phases current controlled by a micro-controller

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



(a) Ultrasound sensor (b) Camera

Figure 2.3: Drone Sensors

2.4 LiPo batteries

The AR.Drone 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 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.

2.5 Motion sensors

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

The AR.Drone 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 with 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.

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 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 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 as the AR.Drone tilting commands.

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

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

2.8 Video streaming and tags 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. QCIF and QVGA image resolutions are supported. The video stream frame rate is set to 15 Hz.

Tags painted on drones can be detected by the drone front camera. These tags can be used to detect other drones during multiplayer games, or to help a drone find its way in the environment. Both tags on the external and internal hull can be detected.

(a) 2D tags on outdoor shell (b) 2D tags on indoor shell

Figure 2.4: Drone shell tags

2.9 Wifi network and connection

The AR.Drone can be controlled from any client device supporting the Wifi ad-hoc mode. The following process is followed :

1. the AR.Drone creates a WIFI network with an ESSID usually called *adrone_xxx* and self allocates a free, odd IP address.
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 drones prior to version 1.1.3)
   - the drone own IP address plus a number between 1 and 4 (starting from version 1.1.3)
5. the client device can start sending requests the AR.Drone IP address and its services ports.

The client can also initiate the Wifi ad-hoc network. If the drone detects an already-existing network with the SSID it intented to use, it joins the already-existing Wifi channel.

2.10 Communication services between the AR.Drone 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 30 times per second.

A video stream is sent by the AR.Drone to the client device on port 5555. 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.

3 AR.Drone SDK Overview

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

- from any personal computer with Wifi connectivity (Linux or Windows);
- from an Apple iPhone;
- (*soon*) from an Android mobile phone.

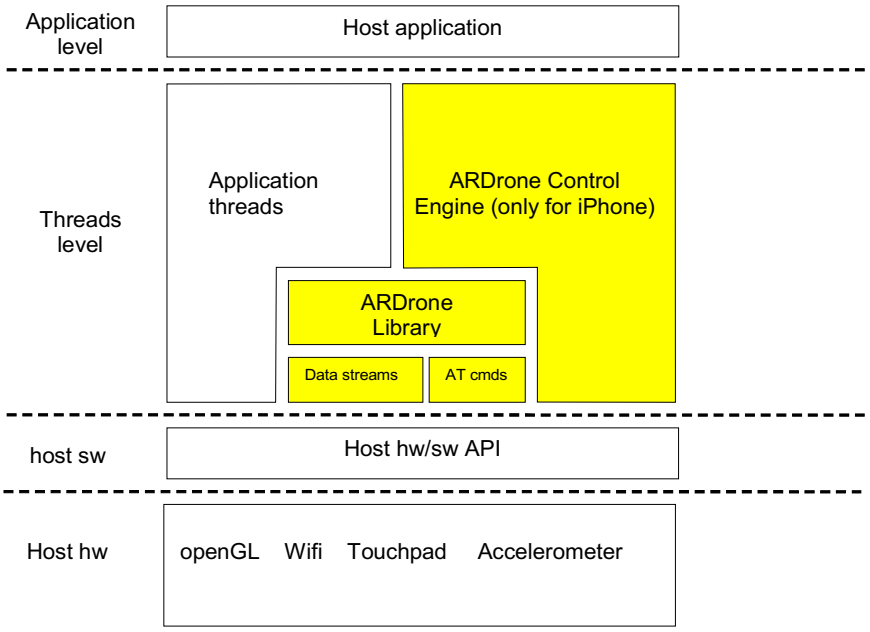
It also allows you, with a bit more effort, to remotely control the drone from any programmable device with a Wifi network card and a TCP / UDP / IP stack - for devices which are not supported by Parrot, a complete description of the communication protocol used by the drone is given in this document;

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 SDK.



3.2 The AR.Drone Library

The AR.Drone 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
- **VLIB** : the video processing library. It contains the functions to receive and decode the video stream
- **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 *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

3.3 The AR.Drone Tool

Part of the AR.Drone 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
- a *control* thread which handles requests from other threads for sending reliable commands from the drone, and automatically checks for the drone acknowledgements.

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 *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 retrived as mentionned in 5.6.

3.4 The AR.Drone Control Engine - *only for Apple iPhone*

The AR.Drone control engine (aka. ARDrone engine) provides all the AR.Drone applications for iPhonewith common methods for managing the drone, displaying its video stream and managing touch / tilt controls and special events on the iPhone.

It is meant to be a common base for all iPhone 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 iPhone application. It is the Control Engine task to acces the **`ARDroneLIB`** .

The AR.Drone Control Engine automatically opens, receives, decodes and displays video stream coming from toy using OpenGL routines. Only one AR Drone Control Engine function need be called inside application for displaying automatically the incoming video stream. Another function allows getting a status of this process.

The following flight parameters are superimposed on video:

- AR Drone battery life will be displayed on top right

The following controls are superimposed on video:

- At the bottom, a take-off button when landed or a landing button when flying
- On the left, a settings button and a zap (change video channel) button
- On the top, an emergency button, which will stop the AR.Drone motors

Special events can occur when in game, and trigger warning messages :

- battery too low
- wifi connection loss
- video connection loss
- engine problem

User can be requested to acknowledge special event message on touch pad.

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 bitfiels 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).

4.2 Drone configuration functions

Before recieving the navdata, your application must set the GENERAL:navdata_demo configuration on the AR.Drone . This can be done using the following functions, or directly by calling the **ARDRONE_TOOL_CONFIGURATION_ADDEVENT** macro.

ardrone_at_navdata_demo

Summary : Makes the drone send a limited amount of navigation data

Corresponding AT command : **AT*CONFIG**

[This function does not take any parameter.

Description :
Some navigation data are used for debugging purpose and are not useful for every day flights. You can choose to receive only the most useful ones by calling this function. This saves some network bandwidth. Most demonstration programs in the SDK (including the iPhone FreeFlight application) use this restricted set of data. Ardrone Navigation uses the whole set of data.

Note : You must call this function or ***ardrone_at_set_navdata_all*** to make the drone start sending any *navdata*.

ardrone_at_set_navdata_all

Summary : Makes the drone send all the navigation data

Corresponding AT command : **AT*CONFIG**

[This function does not take any parameter.

Description :
Some navigation data are used for debugging purpose and are not useful for every day flights. You can choose to receive all the available *navdata* by calling this function. This used some more network bandwidth.

Note : You must call this function or ***ardrone_at_navdata_demo*** to make the drone start sending any *navdata*.

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 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 :

- customize the *demo_navdata_client_process* function to react to navigation information reception (more details in 5.3)
- customize the *output_gtk_stage_transform* function to react to video frames reception (more details in 5.6)
- customize the *update_gamepad* function to react to inputs on a game pad (more details in 5.7)
- 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)

Customizing mainly means sending the appropriate commands from the **ARDroneTool** API. Those commands are listed in chapter 4.

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

5.2 Customizing the client initialization

As is true for every C-based application, the initial entry point for every AR.Drone application is a function called main. The good news is that, when you create a new application using the **ARDroneTool** library, you do not have to write this function yourself.

The **ARDroneTool** library includes a version of this 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 to get a fully functional drone client application.

Listing 5.1 shows the 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 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_add* function
  - the starting off all threads except the *navdata_update* and *ardrone_control* that are started by the 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. To send an event you must use ardrone_control_send_event function declared in *ardrone_control.h*.
- 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.

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 5.1: Application initialization with ARDroneLIB

```
int 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;
}
```

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

```
/* Receving 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 :

- *adrone_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
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***>**.

Definition of the structure of a stage:

Listing 5.7 shows how to build a pipeline with stages. In this sample a socket is opened and the video stream is retrieved.

Listing 5.8 shows how to run the pipeline. This code must be implemented by the user; a sample is provided in the SDK Demo program. For example, you can use a thread dedicated to this.

Functions are defined in **<***VP_Api/vp_api.h***>** file : *vp_api_open* : Initialization of all the stages embedded in the pipeline. *vp_api_run* : Running of all the stages, in loop. *vp_api_close* : Close of all the stages.

Listing 5.6: The video pipeline

```
typedef struct _vp_api_io_stage_
{
//Enum corresponding to the type of stage available, include in file <VP_Api/
    vp_api.h>.
//Only used for Debug.
VP_API_IO_TYPE type;
//Specific data to the current stage. Definitions are given in include files <
    VP_Stages/*> .
void *cfg;
//Structure {vp_api_stage_funcs} is included in <VP_Api/vp_api_stage.h> with the
    definition of stages.
Stages are included also in the directory Soft/Lib/ardrone_tool/Video.
vp_api_stage_funcs_t funcs;
/ /This structure is included in the file <VP_Api/vp_api.h> and is shared between
     all stages. It contains video
buffers and information on decoding the video.
vp_api_io_data_t data;
} vp_api_io_stage_t;
Definition of the structure of a pipeline:
The structure is included in file <VP_Api/vp_api.h> :
typedef struct _vp_api_io_pipeline_
{
//Number of stage to added in the pipeline.

uint32_t nb_stages;
//Address to the first stage.
vp_api_io_stage_t *stages;
//Must equal to NULL
vp_api_handle_msg_t handle_msg;
//Must equal to 0.
uint32_t nb_still_running;
//Must equal to NULL.
vp_api_fifo_t fifo;
} vp_api_io_pipeline_t;
```

5.7 Adding control devices

The **ARDroneTool** and demonstration programs come with an example of how to use a gamepad to pilot the drone.

The *gamepad.c[pp]* files in the example contain the code necessary to detect the presence of a gamepad, poll its status and send corresponding commands.

To add a control device and make **ARDroneTool** consider it, you must write :

- an initialization function, that **ARDroneTool** will call once when initializing the application. The provided example scans the system and searches a known gamepad by looking for its USB Vendor ID and Product ID.
- an *update* function, that **ARDroneTool** will systatically call every 20ms during the application lifetime, unless the initialization fails
- a clean up function, that **ARDroneTool** calls once at the end of the application

Listing 5.7: Building a video pipeline

```
#include <VP_Api/vp_api.h>
#include <VP_Api/vp_api_error.h>
#include <VP_Api/vp_api_stage.h>
#include <ardrone_tool/Video/video_com_stage.h>
#include <ardrone_tool/Com/config_com.h>
vp_api_io_pipeline_t pipeline;
vp_api_io_data_t out;
vp_api_io_stage_t stages[NB_STAGES];
video_com_config_t icc;
icc.com = COM_VIDEO();
icc.buffer_size = 100000;
icc.protocol = VP_COM_UDP;
COM_CONFIG_SOCKET_VIDEO(&icc.socket, VP_COM_CLIENT, VIDEO_PORT, wifi_ardrone_ip);
pipeline.nb_stages = 0;
stages[pipeline.nb_stages].type = VP_API_INPUT_SOCKET;
stages[pipeline.nb_stages].cfg = (void *)&icc;
stages[pipeline.nb_stages].funcs = video_com_funcs;
pipeline.nb_stages++;
```

Listing 5.8: Running a video pipeline

```
res = vp_api_open(&pipeline, &pipeline_handle);
if( SUCCEED(res) )
{
int loop = SUCCESS;
out.status = VP_API_STATUS_PROCESSING;
while(loop == SUCCESS)
{
if( SUCCEED(vp_api_run(&pipeline, &out)) )
{
if( (out.status == VP_API_STATUS_PROCESSING || out.status ==
VP_API_STATUS_STILL_RUNNING) )
{
loop = SUCCESS;
}
}
else loop = -1; // Finish this thread
}
vp_api_close(&pipeline, &pipeline_handle);
}
```

Once these functions are written, you must register your to **ARDroneTool** by using *ardrone_tool_input_add* function which takes a structure parameter *input_device_t* defined in <*ardrone_tool/UI/ardrone_input.h*>. This structure holds the pointers to the three above-mentioned functions.

Structure *input_device_t* with fields :

Listing 5.9: Declaring an input device to ARDroneTool

```
struct _input_device_t {
  char name[MAX_NAME_LENGTH];

  C_RESULT (*init)(void);
  C_RESULT (*update)(void);
  C_RESULT (*shutdown)(void);
} input_device_t;
```

The *update* function will typically call the following functions depending on the buttons pressed by the user :

- *ardrone_tool_set_ui_pad_start* : Takeoff / Landing button
- *ardrone_tool_set_ui_pad_select* : emergency reset all button
- *ardrone_at_set_progress_cmd* : directional buttons or sticks

Note : In SDKs newer than 1.0.4, the following functions are **deprecated**, and are all replaced by the *ardrone_at_set_progress_cmd* command :

- *ardrone_tool_set_ui_pad_ad* : turn right button
- *ardrone_tool_set_ui_pad_ag* : turn left button
- *ardrone_tool_set_ui_pad_ab* : go down button
- *ardrone_tool_set_ui_pad_ah* : go up button
- *ardrone_tool_set_ui_pad_l1* : go left button
- *ardrone_tool_set_ui_pad_r1* : go right button
- *ardrone_tool_set_ui_pad_xy* : go forward / backward buttons (2 arguments)

6 AT Commands

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 A.R.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 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 *Line Feed* character (byte value $10_{(10)}$), noted **<LF>**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=21625,1,0,0,0,0**<LF>AT***REF=21626,290717696**<LF>**

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.

Note : Drones with SDK version 0.3.1 (prior to May 2010) had a different and incompatible sequencing system which used the **AT*SEQ** command. These must be considered deprecated.

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=xx,xx,-1085485875,xx,xx.

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

```
assert(sizeof(int)==sizeof(float));
sprintf(my_buffer,"AT*PCMD,%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)  );
```

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 Deprecated commands

The following commands might have existed in old version of SDKs, and are not supported any more. They should thus NOT be used.

Deprecated commands: **AT*SEQ**, **AT*RADGP**, **AT*MTRIM**

6.5 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*FTRIM	-	Sets the reference for the horizontal plane
AT*CONFIG	key, value	Configuration of the AR.Drone
AT*LED	animation, frequency, duration	Set a led animation on the AR.Drone
AT*ANIM	animation, duration	Set a flight animation on the AR.Drone
AT*COMWDG	-	Reset the communication watchdog

[1] apart from the sequence number

6.6 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**<LF>**

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**<LF>AT***REF=2,290717952**<LF>AT***REF=3,290717696**<LF>**

AT*PCMD

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

Corresponding API function : *ardrone_at_set_progress_cmd*

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

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 $[-1..1]$

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 .. 2	1	0
Usage	Do not use	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 frontward. 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.

AT*FTRIM

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

Corresponding API function : *ardrone_at_set_flat_trim*

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

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.

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

AT*CONFIG

Summary : Sets an configurable option on the drone

Corresponding API function : *ardrone_at_set_toy_configuration*

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

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*COMWDG

Summary : reset communication watchdog

AT*LED

Summary : Sets the drone control loop PID coefficients

Corresponding API function : *ardrone_at_set_led_animation*

Syntax : **AT***LED=%d,%d,%d,%d**<LF>**

Argument 1 : the sequence number
Argument 2 : integer - animation to play
Argument 3 : float - frequence in Hz of the animation
Argument 4 : integer - total duration in seconds of the animation (animation is played (duration$\times$frequence times)

Description :
This command makes the four motors leds blink with a predetermined sequence. The leds cannot be freely controlled by the user.

See the API function description to get the list of the available animations.

Note : The frequency parameter is a floating point number, as previously explained.

AT*ANIM

Summary : Makes the drone execute a predefined movement (called *animation*).

Corresponding API function : *ardrone_at_set_anim*

Syntax : **AT***ANIM=%d,%d,%d**<LF>**

Argument 1 : the sequence number
Argument 2 : integer - animation to play
Argument 3 : integer - total duration in seconds of the animation

Description :
Plays an animation, ie. a predetermined sequence of movements. Most of these movements are small movements (shaking for example) superposed to the user commands.

See the API function description to get the list of the available animations.

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 and **??**).

7.1 Navigation data

The navigation data (or *navdata*) is are 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 is 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	**Drone**	**Sequence**	**Vision**	**Option 1**			...	**Checksum block**		
0x55667788	**state**	**number**	**flag**	id	size	data	...	cks id	size	cks data
32-bit int.	32-bit int.	32-bit int.	32-bit int.	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 that 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.

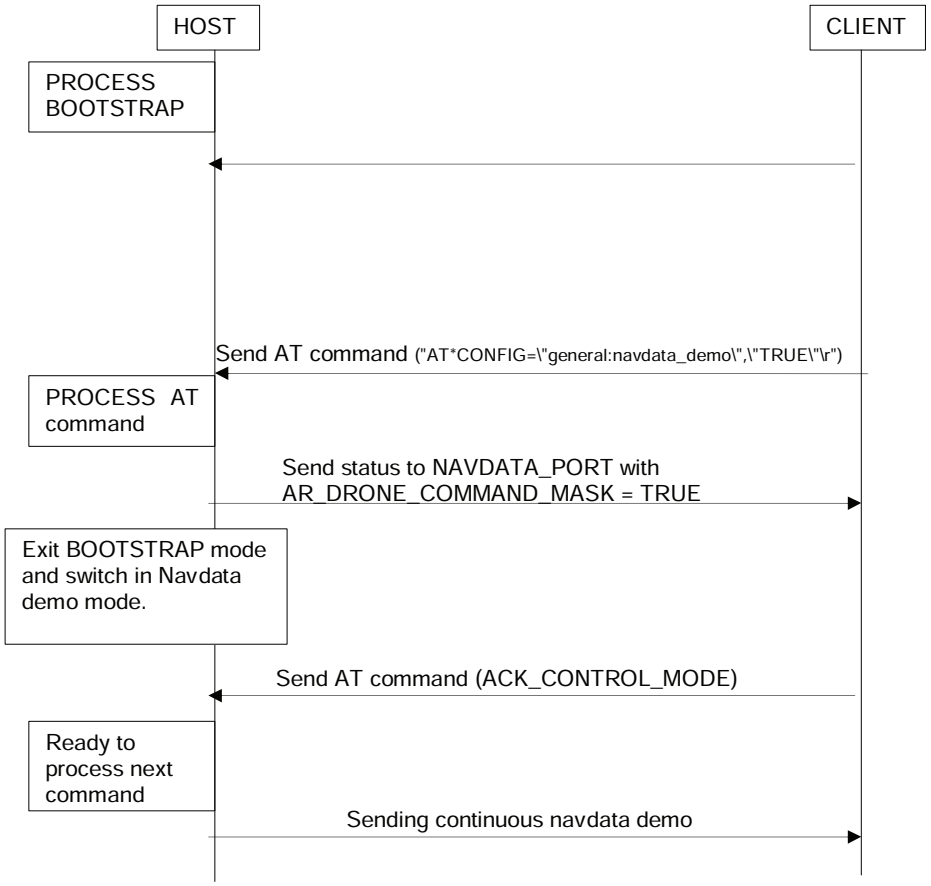


Figure 7.1: Navdata stream initiation

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[4];
uint32_t xc[4];
uint32_t yc[4];
uint32_t width[4];
uint32_t height[4];
uint32_t dist[4];
} __attribute__ ((packed)) navdata_vision_detect_t;
```

The drone can detect up to four 2D-tags. 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 2D-tags.
- type[i]: Type of the detected tag #i ; see the **CAD_TYPE** enumeration.
- xc[i], yc[i]: X and Y coordinates of detected 2D-tag #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 (2D-tag #i), when applicable.
- dist[i]: Distance from camera to detected 2D-tag #i in centimeters, when applicable.

7.2 The video stream

The embedded system uses a proprietary video stream format, based on a simplified version of the H.263 UVLC (Universal Variable Length Code) format (http://en.wikipedia.org/wiki/H263). The images are encoded in the YCBCR color space format, 4:2:0 type (http://en.wikipedia.org/wiki/YCbCr), with 8 bits values. The proprietary format used by the drone is described in 4.2.2, but here is firstly shown the way to produce it.

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 :

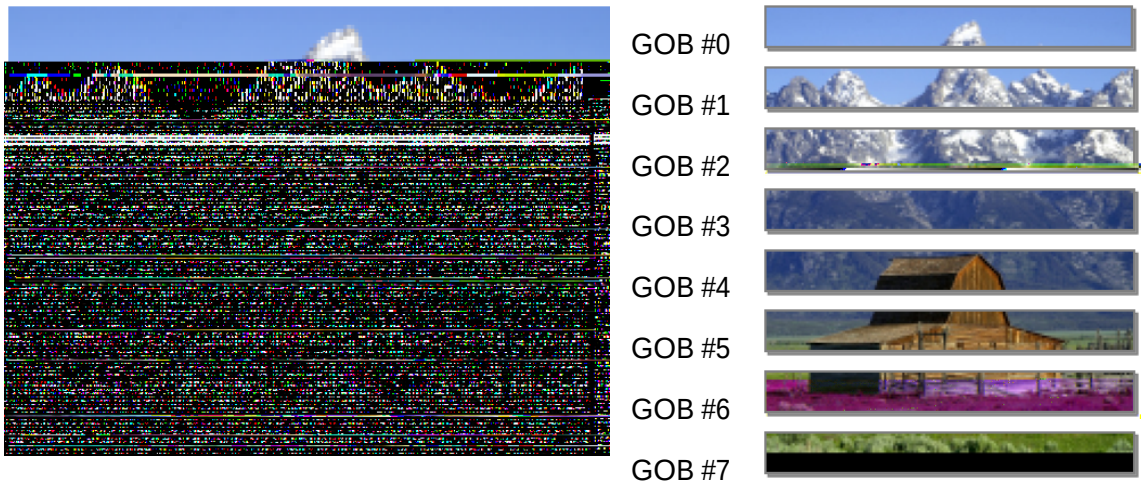
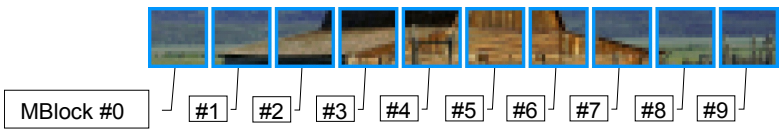


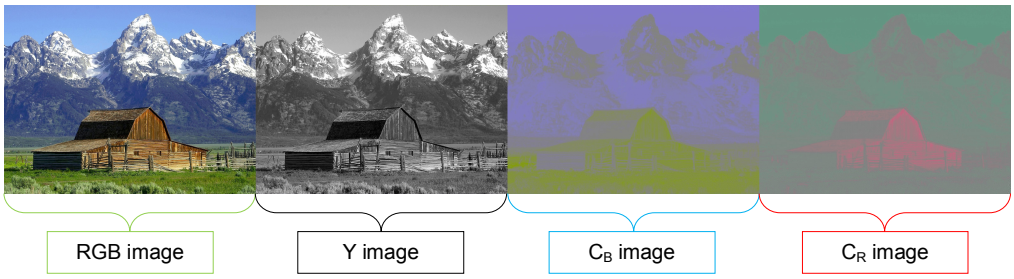
Figure 7.2: Example image (source : http://en.wikipedia.org/wiki/YCbCr)

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



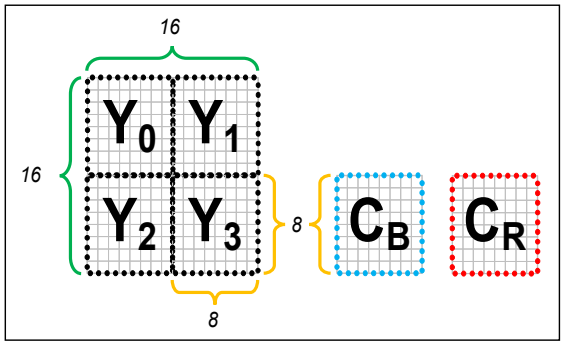
Each macroblock contains informations of a 16x16 image, in YC_BC_R format, type 4:2:0.

(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:

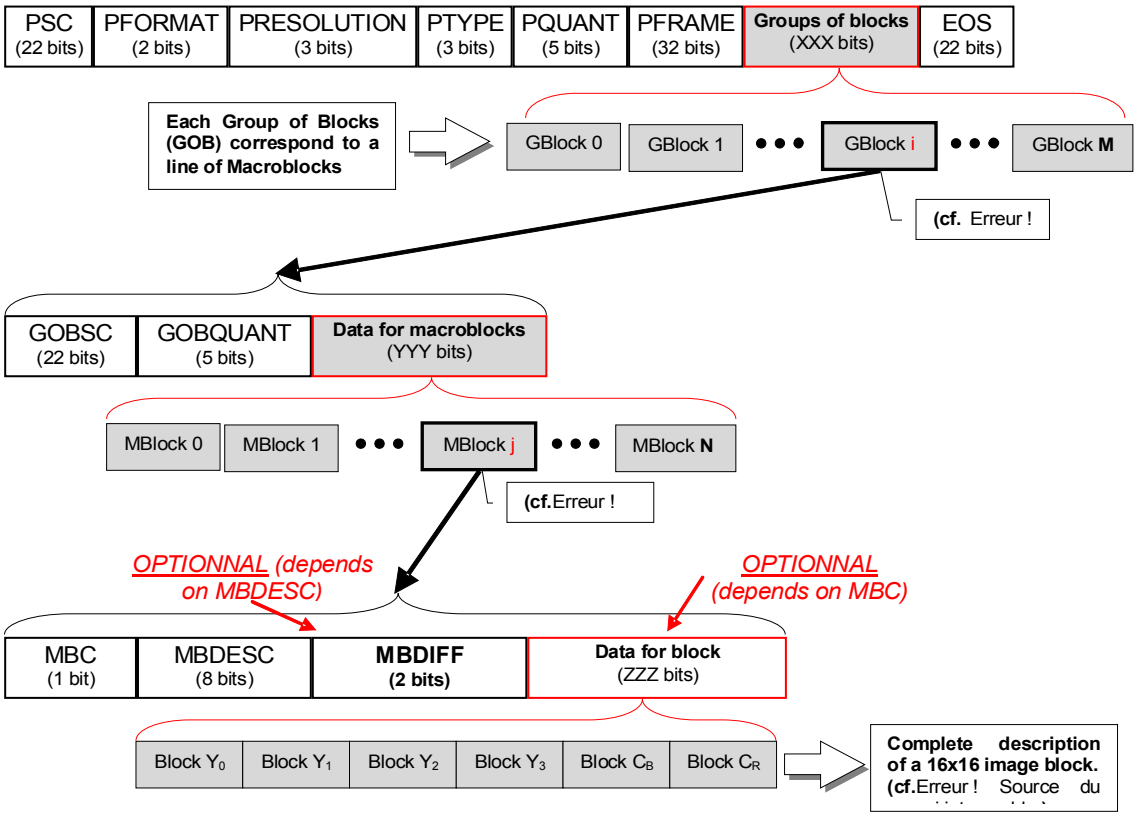
- 4 blocks (Y0, Y1, Y2 and Y3) to form the 16x16 pixels Y image of the luma component (corresponding to a greyscale version of the original 16x16 RGB image).
- 2 blocks of down-sampled chroma components (computed from the original 16x16 RGB image):
  - Cb: blue-difference component (8x8 values)
  - Cr: red-difference component (8x8 values)



7.2.1.1 Layer of images

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:



7.2.1.2 Picture start code (PSC) (22 bits)

Like H.263 UVLC start with a PSC (Picture start code) which is 22 bits long:

0000 0000 0000 0000 1 00000

A PSC is always byte aligned.

7.2.1.3 Picture format (PFORMAT) (2 bits)

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

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

7.2.1.4 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

7.2.1.5 Picture type (PTYPE) (3 bits)

Picture type:

- 000 : INTRA picture
- 001 : INTER picture

7.2.1.6 Picture quantizer (PQUANT) (5 bits)

The PQUANT code is a 5-bits-long word. The quantizer reference for the picture that range from 1 to 31.

7.2.1.7 Picture frame (PFRAME) (32 bits)

The frame number (32 bits).

7.2.1.8 Layer groups of blocks (GOBs)

Data for each group of blocks (GOB) consists of a header followed by data group corresponding to the macroblock, according to the structure shown in Figure 10 and Figure 11. Each GOB corresponds to one or more blockline (cf. Figure 6).

Note : For the first GOB of each picture, the GOB header is always omitted.

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

Each GOB starts with a GOBSC (Group of block start code) which is 22 bits long: 0000 0000 0000 0000 1xxx xx

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

7.2.1.10 Group of block quantizer (GOBQUANT) (5 bits)

The quantizer reference for the GOB that ranges from 1 to 31 (5 bits).

DEPRECATED

7.2.1.11 Layer of macroblocks

Data for each macroblock corresponding to an header of macroblock followed by data of macroblock. The layer structure shown in Figure 10 and Figure 12:

MBC (1 bit)	MBDESC (8 bits)	MBDIFF (2 bits)	Data for block (cf. xxx)

Figure 12 - Layer structure of macroblocks

7.2.1.12 Coded macroblock bit (MBC) (1 bit)

Coded macroblock bit: Bit 0 : 1 means there is a macroblock / 0 means macroblock is all zero.

7.2.1.13 Macroblock description code (MBDESC) (8 bits)

Macroblock description code :

- Bit 0 : 1 means there is non dc coefficients for block y0.
- Bit 1 : 1 means there is non dc coefficients for block y1.
- Bit 2 : 1 means there is non dc coefficients for block y2.
- Bit 3 : 1 means there is non dc coefficients for block y3.

- Bit 4 : 1 means there is non dc coefficients for block cb.
- Bit 5 : 1 means there is non dc coefficients for block cr.
- Bit 6 : 1 means there is a quantization value following this code.
- Bit 7 : Always 1 to avoid a zero byte.

7.2.1.14 Macroblock differential (MBDIFF) (2 bits)

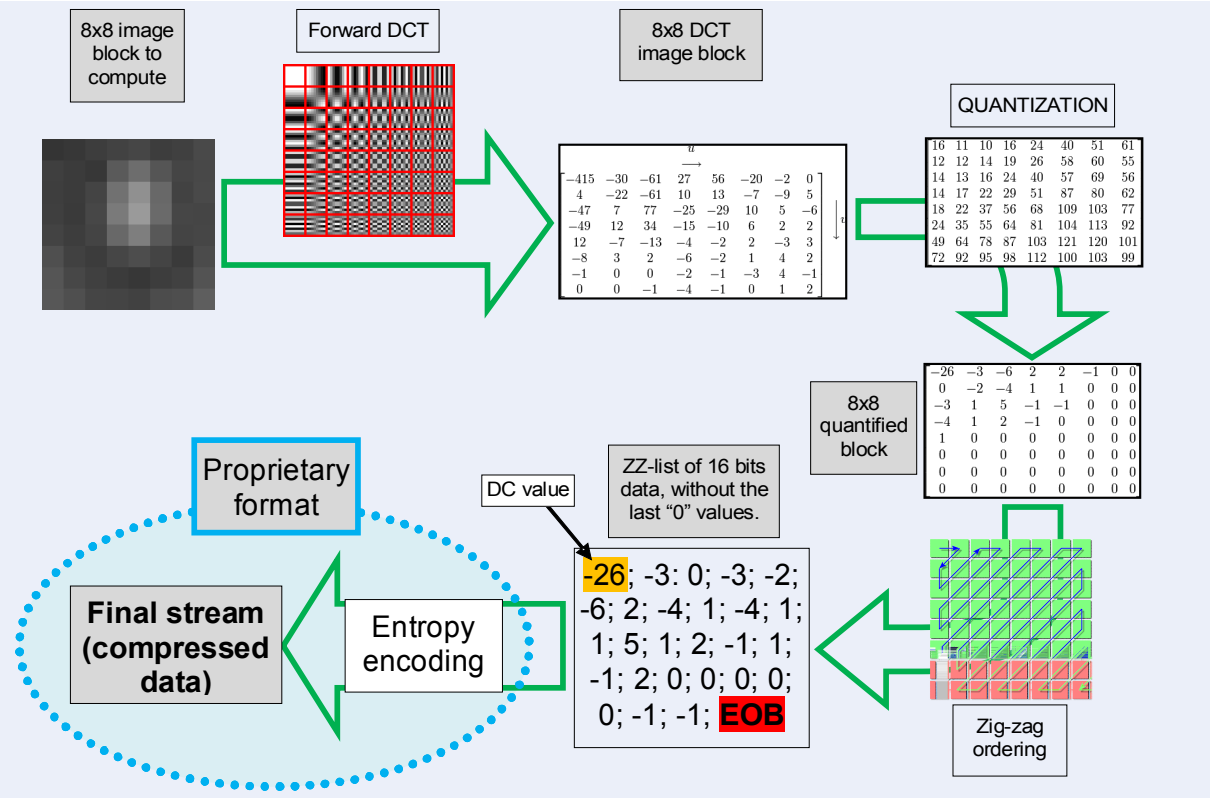
Macroblocks differential value for the quantization (2 bits):

- 00 → -1
- 01 → -2
- 10 → 1
- 11 → 2

7.2.1.15 Layer of blocks

As shown in Figure 9, a macroblock corresponds to four luminance ("luma") blocks and two color ("chroma") blocks. Each of those 8x8 pixels blocks is computed by hardware with the first steps of JPEG encoding (http://en.wikipedia.org/wiki/Jpeg). It concerns steps "DCT", "Quantization", and "zig-zag ordering", but exclude data compression. This last step is based on a proprietary format, detailed further below.

Note: The encoding of blocks made by hardware needs 16 bits values, so there is a "8?16 bits values" conversion before the "pseudo" JPEG encoding.



7.2.1.16 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: The first value of the list (the "DC value", cf. Figure 13) is not compressed, but 16 to 10 bits encoded.

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	x x	6	(x x) +4	4 : 7	3
0 0 0 0 1	x x x	8	(x x x) +8	8 : 15	4
0 0 0 0 0 1	x x x x	10	(x x x x) +16	16 : 31	5
0 0 0 0 0 0 1	x x x x x	12	(x x x x x) +32	32 : 63	6

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

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

7.2.2 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")

- 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")

7.2.3 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.
- While there remains compressed data (till the "EOB" code):
  - Reading of the zero-counter value as explain below:
    - Read the *coarse*S 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 explain 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: $0001_{(2)} \rightarrow$ xx**s** $\rightarrow$ 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 x (-1)
    - Add the resulting non-zero value to the output list.
- (End of "while")

7.2.4 Example

Encoding :

- Initial data list :
  -26; -3; 0; 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):
  1111111111100110; 1; 001 11; 0001 00; 0001 110; 1; 0001 011; 1; 01
- Final stream :
  1111100110100111000100000111010001011101

Decoding :

- Initial bit-data stream :
  {1111000111011100011000101001010000101000110 1}
- 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"}; {01110001100010100101000110011010 1}

- Step 4, first couple of values:
  {"-57"}; [{01.}; {1.1}]; {0001100010100101000110011010 1}
  {"-57"}; ["0"; "-1"]; {0001100010100101000110011010 1}

- 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"]; {001100110101}

- 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"; "0"; "+3"; "EOB"}

7.2.5 End of sequence (EOS) (22 bits)

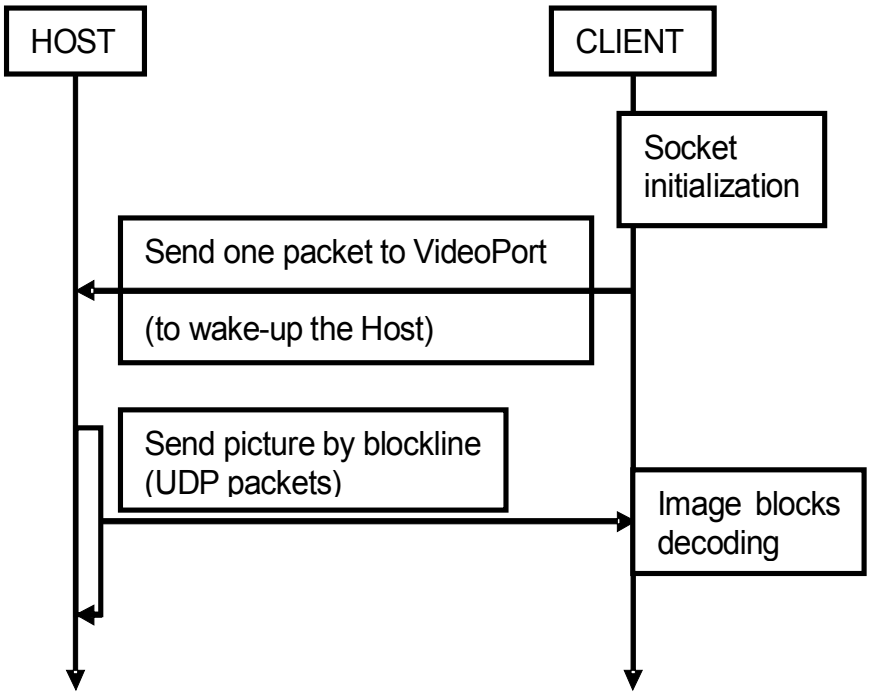
The end of sequence (EOS) which is 22 bits long :

```
0000 0000 0000 0000 1 11111
```

7.2.6 Intiating 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.



8 Drone Configuration

The drone behaviour depends on many parameters which can be modified by using the AT*CONFIG AT command, or by using the appropriate **ARDroneLIB** function (see chapter 4.1).

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 = 17
GENERAL:num_version_soft = 1.1.3
GENERAL:soft_build_date = 2010-08-06 09:48
GENERAL:motor1_soft = 1.8
GENERAL:motor1_hard = 3.0
GENERAL:motor1_supplier = 1.1
GENERAL:motor2_soft = 1.8
GENERAL:motor2_hard = 3.0
GENERAL:motor2_supplier = 1.1
GENERAL:motor3_soft = 1.8
GENERAL:motor3_hard = 3.0
GENERAL:motor3_supplier = 1.1
GENERAL:motor4_soft = 1.8
GENERAL:motor4_hard = 3.0
GENERAL:motor4_supplier = 1.1
GENERAL:ardrone_name = My ARDrone
GENERAL:flying_time = 1810
GENERAL:navdata_demo = TRUE
GENERAL:com_watchdog = 2
GENERAL:video_enable = TRUE
GENERAL:vision_enable = TRUE
GENERAL:vbat_min = 9000
CONTROL:accs_offset = { -2.2696499e+03 1.9345000e+03 1.9331300e+03 }
CONTROL:accs_gains = {  9.6773100e-01 2.3794901e-02 7.7836603e-02  -1.2318300e-02
     -9.8853302e-01 3.2103900e-02  7.2116204e-02 -5.6212399e-02 -9.8713100e-01 }
CONTROL:gyros_offset = { 1.6633199e+03 1.6686300e+03 1.7021300e+03 }
CONTROL:gyros_gains = { 6.9026551e-03 -6.9553638e-03 -3.8592720e-03 }
CONTROL:gyros110_offset = { 1.6560601e+03 1.6829399e+03 }
CONTROL:gyros110_gains = { 1.5283586e-03 -1.5365391e-03 }
CONTROL:gyro_offset_thr_x = 4.0000000e+00
CONTROL:gyro_offset_thr_y = 4.0000000e+00
CONTROL:gyro_offset_thr_z = 5.0000000e-01
CONTROL:pwm_ref_gyros = 471
CONTROL:control_level = 1
CONTROL:shield_enable = 1
CONTROL:euler_angle_max = 3.8424200e-01
CONTROL:altitude_max = 3000
CONTROL:altitude_min = 50
CONTROL:control_trim_z = 0.0000000e+00
CONTROL:control_iphone_tilt = 3.4906584e-01
CONTROL:control_vz_max = 1.2744493e+03
CONTROL:control_yaw = 3.1412079e+00
CONTROL:outdoor = TRUE
CONTROL:flight_without_shell = TRUE
CONTROL:brushless = TRUE
CONTROL:autonomous_flight = FALSE
CONTROL:manual_trim = FALSE
NETWORK:ssid_single_player = AR_hw11_sw113_steph
NETWORK:ssid_multi_player = AR_hw11_sw113_steph
NETWORK:infrastructure = TRUE
NETWORK:secure = FALSE
NETWORK:passkey =
NETWORK:navdata_port = 5554
NETWORK:video_port = 5555
NETWORK:at_port = 5556
NETWORK:cmd_port = 0
NETWORK:owner_mac = 04:1E:64:66:21:3F
NETWORK:owner_ip_address = 0
NETWORK:local_ip_address = 0
NETWORK:broadcast_address = 0
```

```
PIC:ultrasound_freq = 8
PIC:ultrasound_watchdog = 3
PIC:pic_version = 100925495
VIDEO:camif_fps = 15
VIDEO:camif_buffers = 2
VIDEO:num_trackers = 12
DETECT:enemy_colors = 1
SYSLOG:output = 7
SYSLOG:max_size = 102400
SYSLOG:nb_files = 5
.
```

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 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. paramter

```
int enemy_color;
enemy_color = ORANGE_GREEN;
ARDRONE_TOOL_CONFIGURATION_ADDEVENT(enemy_colors, 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)commandId withParameter:(void *)parameter fromSender:(id)sender*.

The first parameter is the ID of the config you want to change. The ID list can be found in the ***ARDroneTypes.h***.

The second parameter is the parameter of the command.
As 1.6 SDK, parameter types are the following :
- ARDRONE_COMMAND_ISCLIENT : raw integer casted to (void *).
- ARDRONE_COMMAND_DRONE_ANIM : ARDRONE_ANIMATION_PARAM pointer.
- ARDRONE_COMMAND_DRONE_LED_ANIM : ARDRONE_LED_ANIMATION_PARAM pointer.
- ARDRONE_COMMAND_VIDEO_CHANNEL : raw integer casted to (void *). Values can be found in ***ARDroneGeneratedTypes.h*** file
- ARDRONE_COMMAND_CAMERA_DETECTION : raw integer casted to (void *). Values can be found in ***ARDroneGeneratedTypes.h*** file
- ARDRONE_COMMAND_ENEMY_SET_PARAM : ARDRONE_ENEMY_PARAM pointer.
- ARDRONE_COMMAND_ENABLE_COMBINED_YAW : boolean casted to (void *).
- ARDRONE_COMMAND_SET_FLY_MODE : raw integer casted to (void *). Values can be found in ***ardrone_api.h*** file

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

Listing 8.3: Example of setting detection parameters with Control Engine

```
ARDRONE_ENEMY_PARAM enemyParam;
enemyParam.color = ARDRONE_ENEMY_COLOR_GREEN;
enemyParam.outdoor_shell = 0;
[ardrone executeCommandIn:ARDRONE_COMMAND_ENEMY_SET_PARAM withParam:(void *)&
    enemyParam fromSender:nil];
```

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 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 _____ *Read only*

Description :
Version of the configuration subsystem (Currently 1).

GENERAL:num_version_mb _____

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

Read only

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

GENERAL:navdata_demo

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 cae in the Linux example *ardrone_navigation*).

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

API use example :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (navdata_demo, TRUE, myCallback);
```

GENERAL:navdata_options

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","65537"**

API use example :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (navdata_options, (NAVDATA_OPTION_-
MASK (NAVDATA_DEMO_TAG) | NAVDATA_OPTION_MASK (NAVDATA_VISION_DETECT_-
TAG)), myCallback);
```

GENERAL:com_watchdog

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 — *Read/Write*

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

GENERAL:vision_enable — *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 algoritms

GENERAL:vbat_min — *Read only*

Description :
Reserved for future use. Minimum battery level before shutting down automatically the AR.Drone.

8.4 Control configuration

CONTROL:accs_offset — *Read only*

Description :
Parrot internal debug informations. AR.Drone accelerometers offsets.

CONTROL:accs_gains — *Read only*

Description :
Parrot internal debug informations. AR.Drone accelerometers gains.

CONTROL:gyros_offset — *Read only*

Description :
Parrot internal debug informations. AR.Drone gyrometers offsets.

CONTROL:gyros_gains — *Read only*

Description :
Parrot internal debug informations. AR.Drone gyrometers gains.

CONTROL:gyros110_offset — *Read only*

Description :
Parrot internal debug informations.

CONTROL:gyros110_gains — *Read only*

Description :
Parrot internal debug informations.

CONTROL:gyro_offset_thr_x — *Read only*

Description :
Parrot internal debug informations.

Note : Also exists for the y and z axis.

CONTROL:pwm_ref_gyros — *Read only*

Description :
Parrot internal debug informations.

CONTROL:control_level — *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 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.

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

API use example :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (control_level, (ardrone_control_-
config.control_level | (1 « CONTROL_LEVEL_COMBINED_YAW)), myCallback);
```

CONTROL:shield_enable — *Read/Write*

Description :
Reserved for future use.

CONTROL:euler_angle_max — *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.

This parameter is a positive floating-point value between 0 and 0.52 (ie. $30\,\mathrm{deg}$). 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 :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (euler_angle_max, 0.25, myCallback);
```

CONTROL:altitude_max — *Read/Write*

Description :

Maximum drone altitude in millimeters.

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.

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

API use example :
```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (altitude_max, 3000, myCallback);
```

CONTROL:altitude_min — *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 :
```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (altitude_min, 50, myCallback);
```

CONTROL:control_trim_z — *Read/Write*

Description :
Manual trim function : should not be used.

CONTROL:control_iphone_tilt — *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 :
```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (control_iphone_tilt, 0.25, myCallback);
```

CONTROL:control_vz_max — *Read/Write*

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

Recommanded 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 :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (control_vz_max, 1000, myCallback);
```

CONTROL:control_yaw

Read/Write

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

Recommanded 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 :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (control_yaw, 3.0, myCallback);
```

CONTROL:outdoor

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 corresponds to the *Outdoor flight* setting of AR.FreeFlight

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

API use example :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (outdoor, TRUE, myCallback);
```

CONTROL:flight_without_shell

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 :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (flight_without_shell, TRUE, myCallback);
```

CONTROL:brushless — *Read/Write*

Description :
Deprecated.

CONTROL:autonomous_flight — *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.Drones.

CONTROL:manual_trim — *Read only*

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

CONTROL:indoor_euler_angle_max — *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 — *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 — *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 — *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 — *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 — *Read/Write*

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

CONTROL:flying_mode — *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, and won't respond to the progressive commands. This new flying mode was developped for 2011 CES autonomous demonstration.

Note : *This mode is experimental, and should be used with care. The roundel (cocarde) detection and this flying mode are subject to changes without notice in further firmwares.*

Note : Roundel detection must be activated with the *DETECT:detect_type* setting if you want to use the "HOVER_ON_TOP_OF_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 :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (flying_mode, FLYING_MODE_FREE_-
FLIGHT, myCallback);
```

CONTROL:flight_anim — *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.

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_YAW_SHAKE,
MAYDAY_TIMEOUT[ARDRONE_ANIMATION_YAW_SHAKE]);
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (flight_anim, param, myCallback);
```

8.5 Network configuration

NETWORK:ssid_single_player — *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 — *Read/Write*

Description :
Currently unused.

NETWORK:infrastructure — *Read/Write*

Description :
Reserved for future use.

NETWORK:secure — *Read/Write*

Description :
Reserved for future use.

NETWORK:passkey — *Read/Write*

Description :
Reserved for future use.

NETWORK:navdata_port — *Read only*

Description :
The UDP socket port where the navdata are sent. Defaults to 5554.

NETWORK:video_port — *Read only*

Description :
The UDP socket port where the vido is sent. Defaults to 5555.

NETWORK:at_port — *Read only*

Description :

The UDP socket port where the AT*commands are sent. Defaults to 5556

NETWORK:cmd_port — *Read only*

Description :
Unused.

NETWORK:owner_mac — *Read/Write*

Description :
Mac addres 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);
```

NETWORK:owner_ip_address — *Read/Write*

Description :
Unused.

NETWORK:local_ip_address — *Read/Write*

Description :
Unused.

NETWORK:broadcast_ip_address — *Read/Write*

Description :
Unused.

8.6 Nav-board configuration

PIC:ultrasound_freq *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 ar availaible : 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 :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (ultrasound_freq, ADC_CMD_SELECT_-
ULTRASOUND_22Hz, myCallback);
```

PIC:ultrasound_watchdog *Read/Write*

Description :
Debug configuration that should not be modified.

PIC:pic_version *Read only*

Description :
The software version of the Nav-board.

8.7 Video configuration

VIDEO:camif_fps — *Read only*

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

VIDEO:camif_buffers — *Read only*

Description :
Buffer depth for the video interface.

VIDEO:num_trackers — *Read only*

Description :
Number of tracking point for the speed estimation.

VIDEO:bitrate — *Read/Write*

Description :
Reserved for future use.

VIDEO:bitrate_control_mode — *Read/Write*

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

Note : Before enabling this config, make sure that your video decoder is able to handle the variable bitrate mode !

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

API use example :

```
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (bitrate_control_mode, 0, myCallback);
```

VIDEO:codec — *Read/Write*

Description :
Reserved for future use.

VIDEO:video_channel — *Read/Write*

Description :
The video channel that will be sent to the controller.

Current implementation supports 4 different channels :
- ARDRONE_VIDEO_CHANNEL_HORI
- ARDRONE_VIDEO_CHANNEL_VERT
- ARDRONE_VIDEO_CHANNEL_LARGE_HORI_SMALL_VERT
- ARDRONE_VIDEO_CHANNEL_LARGE_VERT_SMALL_HORI

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

API use example :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (video_channel, ARDRONE_VIDEO_-
CHANNEL_HORI, myCallback);
```

8.8 Leds configuration

LEDS:leds_anim *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[20];
float frequency = 2.0;
snprintf (param, sizeof (param), "%d,%d,%d", ARDRONE_LED_ANIMATION_BLINK_-
ORANGE, *(unsigned int *)&frequency, 5);
ARDRONE_TOOL_CONFIGURATION_ADDEVENT (flight_anim, param, myCallback);
```

8.9 Detection configuration

DETECT:enemy_colors — *Read/Write*

Description :
The color of the hulls you want to detect. Possible values are green, yellow and blue (respective integer values as of 1.5.1 firmware : 1, 2, 3).

Note : This config will only be used for standard tags/hulls detection. New detection algorithms (Cocarde, Stripe) have predefined colors.

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

API use example :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (enemy_colors, ARDRONE_ENEMY_-
COLOR_BLUE, myCallback);
```

DETECT:enemy_without_shell — *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 :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (enemy_without_shell, 0, myCallback);
```

DETECT:detect_type — *Read/Write*

Description :
Select the active tag detection.

Some details about detections can be found in the ***ardrone_api.h*** file.

Note : Stripe detection is experimental and should not be used as of 1.5.1 firmware.

Note : Roundel (Cocarde) detection is experimental and may be changed in further firmwares.

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.

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

API use example :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (detect_type, ARDRONE_CAMERA_-
DETECTION_COCARDE, myCallback);
```

DETECT:detections_select_h

Read/Write

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

Detections offsets are defined in the ***ardrone_api.h*** file.

Note : Stripe detection is experimental and should not be used as of 1.5.1 firmware.

Note : Roundel (Cocarde) detection is experimental and may be changed in further firmwares.

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 :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (detections_select_h, TAG_TYPE_-
MASK (TAG_TYPE_SHELL_TAG), myCallback);
```

DETECT:detections_select_v_hsync

Read/Write

Description :
Bitfileds 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 20fps 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 :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (detections_select_v_hsync, TAG_-
TYPE_MASK (TAG_TYPE_ROUNDEL), myCallback);
```

DETECT:detections_select_v

Read/Write

Description :
Bitfileds 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 :

```
    ARDRONE_TOOL_CONFIGURATION_ADDEVENT (detections_select_v, TAG_TYPE_-
MASK (TAG_TYPE_ROUNDEL), myCallback);
```

8.10 SYSLOG section

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

9 F.A.Q.

Here you will find questions frequently asked on the different forums.

Why is the Android example incomplete ?

The AR.Drone product was designed to work with an ad-hoc Wifi connection, in order to be controllable anywhere in the wild from a mobile device like the iPhone. Unfortunately Google does not currently officially allow the use of ad-hoc connections with the Android OS. Though such connection is technically possible, the Android example application is not a priority until Google changes its mind. It was successfully run on a Nexus One phone though, and the code is provided for people who are not afraid of jail breaking their phone and getting their hands dirty.

Can I add some additional hardware to the drone ?

This is currently not supported. The drone has a USB master port that could be used later to customize the drone hardware, but it is disbaled on current drones.

Can I replace the embedded drone software with my own ? Can I get the sensors and engines specifications ?

No. This SDK is meant to allow remote controlling of the drone only.

Part II

Tutorials

10 Building the iOS Example

The AR.Drone SDK provides the full source code of AR.FreeFligt 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 developper account with associated Developper profiles.
Further informations can be found on Apple website

11 Building the Linux Examples

The AR.Drone SDK provides two client application examples for Linux.

The first one, called *Linux SDK Demo*, shows the minimum software required to control an AR.Drone with a gamepad. It should be used as a skeleton for all your Linux projects.

The second 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.

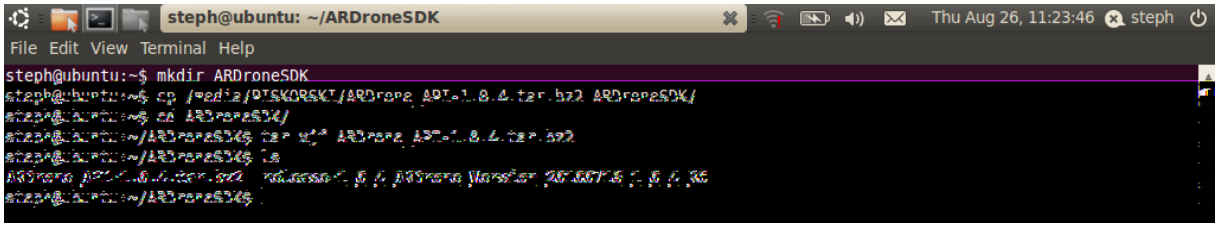
In this section, we are going to show you how to build those example on an Ubuntu 10.04 workstation. We will suppose you are familiar with C programming and you already have a C compiler installed on your machine.

11.1 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_API-x.x.x.tar.bz2* in the directory of your choice. In this document we will note ⟨**SDK**⟩the extracted directory name.

```
$  tar xjf ARDrone_API-<<ARDversion>>.tar.bz2
```



To build the examples, you will need libraries like IW (for wireless management), gtk (to display an graphic interface), and SDL (to easily display the video stream) :

```
$  sudo apt-get update
$  sudo apt-get install libsdl-dev libgtk2.0-dev libiw-dev
```

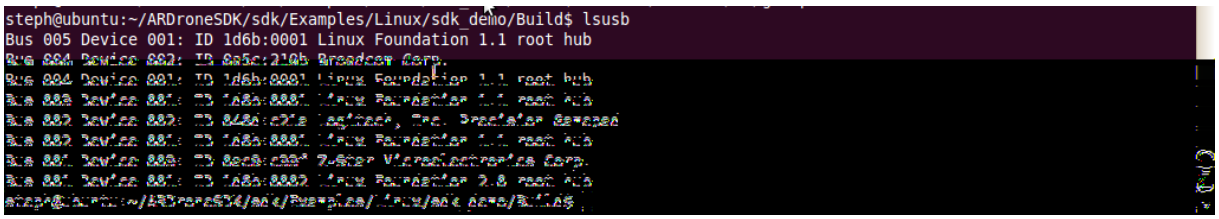
11.2 Prepare the source code

The demonstration programs allow you to pilot the drone with a gamepad. To keep the code simple enough, it only works with two models of gamepads : a Logitech Precision gamepad and a Sony Playstation 3 gamepad. If you own one of these, you can plug it now and skip this section. Otherwise you must first change a few settings in the code, or the demonstration programs won't do much.

Identify your gamepad

Plug the gamepad you want to use, and retrieve its USB identifier with the *lsusb* command. Here is an example for the logitech gamepad :

We are interested in the `ID 046d:c21a` information. It must be copied at the beginning of the *sdk_demo/Sources/UI/gamepad.h* file.



Listing 11.1: Changing gamepad.h

```
...
//  replace this value with the ID of your own gamepad
#define GAMEPAD_LOGITECH_ID 0x046dc21a
...
```

This value will be used by the SDK demo program to find which gamepad to query when flying.

Next step is identifying the buttons on your gamepad, with the *jstest* utility :

```
$  sudo apt-get install joystick
$  jstest /dev/input/js0
```

You will see a list of axes and buttons; play with your gamepads buttons and make a list of which buttons you want to use to go up / down,left / right,etc.

Once you have done so, edit the *sdk_demo/Sources/UI/gamepad.c* file, and search the *gamepad_update* function. This function is called 20 times per second and reads the gamepad state to send according commands to the drone. The code should be self-explanatory; have a look at the *update_ps3pad* for an example with a analogic pad.

Please do the same to the and *Navigation/Sources/UI/gamepad.c* file, for the two examples programs are independant and do not share their gamepad management code, though they are the same. You can even copy / paste the *gamepad* files between the two examples.

11.3 Compile the SDK Demo example

First we must build the **ARDroneLIB** library, by using the makefile provided :

```
$  cd ⟨SDK⟩/ARDroneLib/Soft/Build
$  make
```

Compilation is successful if the last executed action is "ar rcs *libpc_ardrone.a* ..." and succeeds.

The second step is to compile the example itself :

```
$  cd ⟨SDK⟩/Examples/Linux/sdk_demo/Build
$  make
```

An executable program called *linux_sdk_demo* is created in the current working directory.

11.4 Run the SDK Demo program

First unpair your drone if you flew with an iPhone before !

Although the demonstration program can configure the WIFI network itself, it is safer for a first test to manually connect your workstation to the drone WIFI network, and potentially manually configure its IP address to 192.168.1.xxx and subnet mask to 255.255.255.0, where xxx>1.

Once this is done, you can test the connection with a ping command :

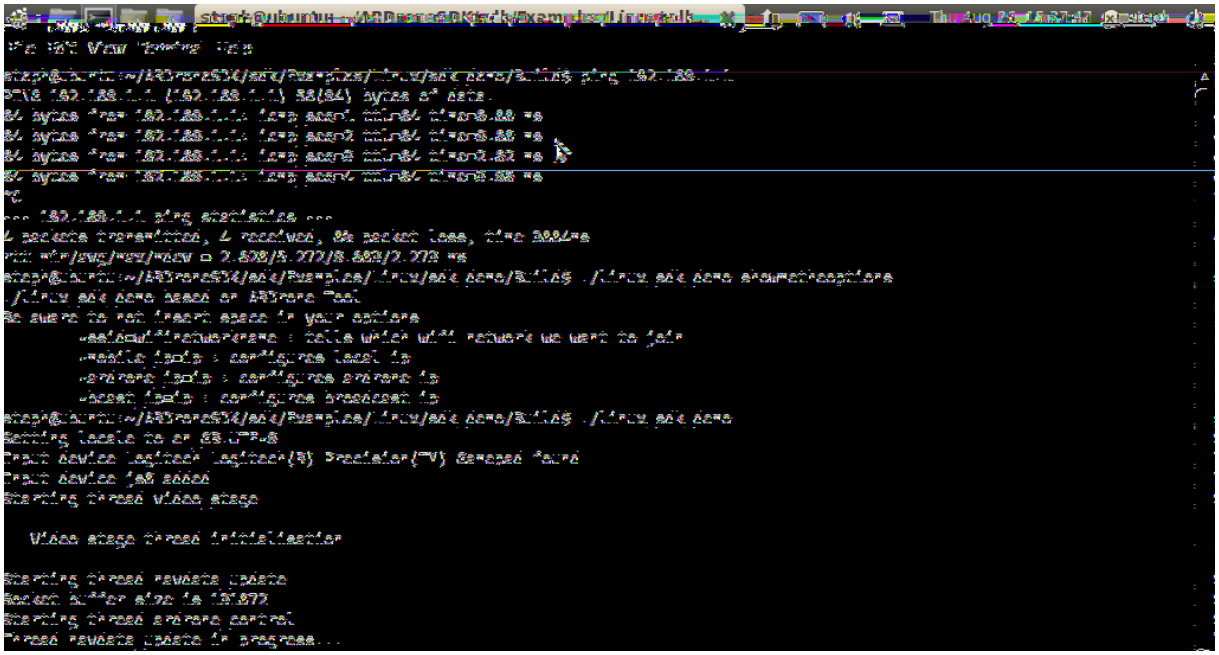
```
$  ping 192.168.1.1
```

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 :

What to expect ?

Check the printed messages; it should say a gamepad was found and then initialize a bunch of things :



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.

11.5 Compile the *Navigation* example

First we must build the **ARDroneLIB** library (unless you already have for the SDK Demo example), by using the makefile provided :

```
$  cd ⟨SDK⟩/ARDroneLib/Soft/Build
$  make
```

Compilation is successful if the last executed action is "ar rcs ***libpc_ardrone.a*** ..." and succeeds.

The second step is to compile the example itself :

```
$  cd ⟨SDK⟩/Examples/Linux/Navigation/Build
$  make
```

An executable program called ***ardrone_navigation*** is created in the ⟨**SDK**⟩ / *ARDroneLib/Version/Release* directory.

11.6 Run the *Navigation* program

First unpair your drone if you flew with an iPhone before !

Although the demonstration program can configure the WIFI network itself, it is safer for a first test to manually connect your workstation to the drone WIFI network, and potentially manually configure its IP address to 192.168.1.xxx and subnet mask to 255.255.255.0, where xxx>1.

Once this is done, you can test the connection with a ping command :

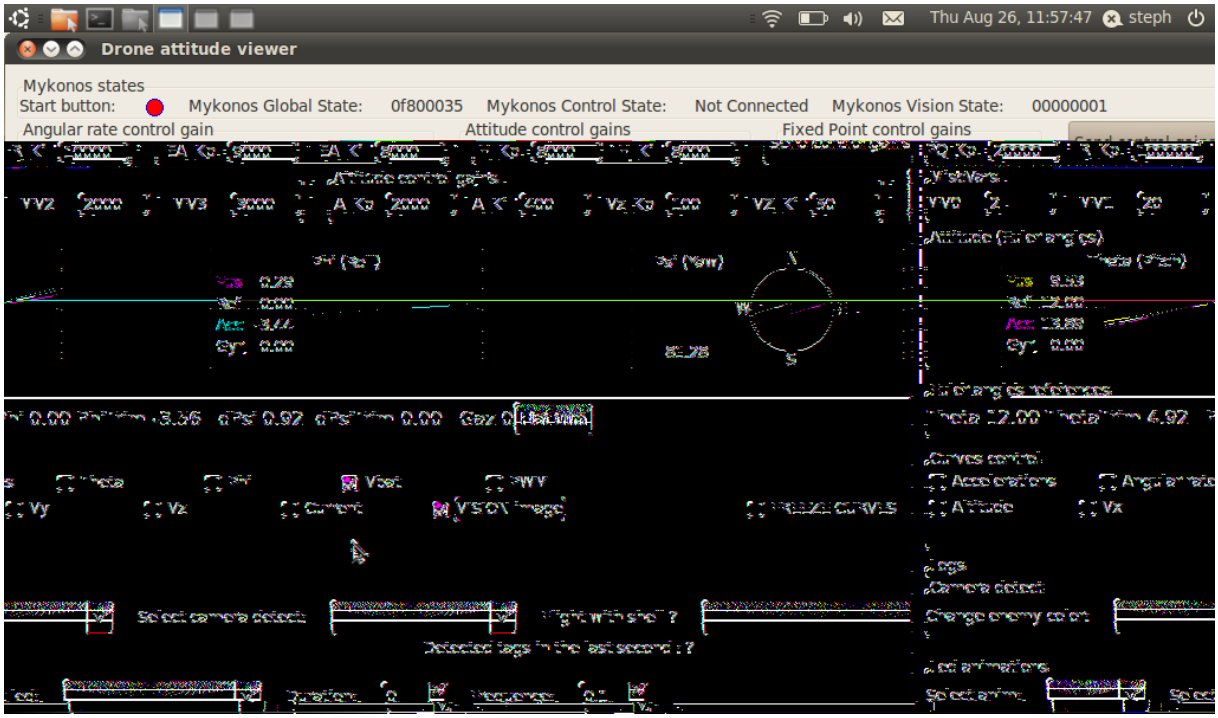
```
$  ping 192.168.1.1
```

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. In the following screenshot the drone is bending backwards.



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.

Check the *VISION image* checkbox to make a second window appear, which will display the video stream :

Feel free to click on all the buttons you want to discover all the commands and all the information sent by the drone (just be careful with the PID gains) !

12 Building the Windows Example

The AR.Drone SDK provides a client application example for Windows.

This example called *SDK Demo*, shows the minimum software required to control an AR.Drone with a gamepad or a keyboard. It should be used as a skeleton for all your Windows projects.

In this section, we are going to show you how to build this example on a Windows Seven workstation. We will suppose you are familiar with C programming and you already have Visual Studio installed on your machine.

12.1 Set up your development environment

Get some development tools

This demo was tested with Microsoft Visual C++ 2008 Express edition, on a Windows 7 64Bits Platform. It should work with any version of Windows XP, Vista, and Seven with minor changes if any.

The required libraries are :

- The Microsoft Windows SDK (available here), *Windows Headers and Library*.
- The Microsoft DirectX SDK (available here), to use a Gamepad for drone control.
- The SDL Library (available here) which is used in the example to display the video, but which can be easily replaced;
- *Only for Windows XP and earlier* - The Pthread for Win32 library (available here)

Get the development files

You should have downloaded and unpacked the following elements :

- the *ARDroneLib* directory (with the *Soft*, *VLIB* and *VP_SDK* subdirectories)
- the Examples directory and specifically its Win32 subdirectory, which contains :
  - the demonstration source code in *sdk_demo*
  - the Visual C++ 2008 Express solution in *VCProjects*

Make the project aware of the code location on your computer

- Open the Visual Studio solution (file *\Examples \Win 32\VCProjects \ARDrone \ARDrone .sln*).
- Open the Property Manager tab (next to the Solution Explorer and the Class View tabs).
- Double click on any of the *ArDrone_properties* entry to edit it.
- Go to **`Common Properties -> User Macros`**
- Edit the **`ARDroneLibDir`** macro so its contains the path to the *ARDroneLib* directory on your computer.
- Edit the **`Win32ClientDir`** macro so its contains the path to the demonstration source code directory on your computer.

Note : you can also directly modify those paths by editing the *ArDrone_properties.vsprops* file with any text editor.

Make the project aware of the libraries location on your computer

- In Visual Studio up to version 2008, the menu

  `Tools->Options->Environment->Projects and Solutions->VC++ Directories`

  must contains the paths to the Include and Libraries files for the above mentioned pre-requisite libraries.
- In Visual Studio 2010, these directories must be set in the Project settings.

12.2 Required settings in the source code before compiling

Operating System related settings

In the Solution Explorer, search for the *vp_os_signal_dep.h* in the ARDroneLib project.

In this header file, one of the two following macros must be defined :

- Use #**define USE_WINDOWS_CONDITION_VARIABLES** to use Microsoft Windows SDK for thread synchronisation (you must have Windows Vista or later)
- Use #**define USE_PTHREAD_FOR_WIN32** to use pthreads for synchronisation (mandatory if you have Windows XP or earlier, possible with later Windows versions)

Drone related settings

The *win32_custom.h* file contains the IP address to connect to. It's 192.168.1.1 by default, but the drone actual IP address might be different if several drones use the same Wifi network (drones then pick random addresses to avoid IP address conflicts). Check your drone IP address with pings or telnet commands before running the example.

Gamepad related settings

The *gamepad.cpp* file contains code to pick the first gamepad that can be found on the computer and which is supported by the DirectX subsystem. In the example, buttons are statically linked to drone actions for three specific gamepads. Other gamepads have no effect. Please modify this code to enable the drone to move with your own input device (code should be self-explanatory).

To identify the buttons available on your gamepad, go to the Windows configuration pannel and go to your gamepad properties pages (there you can calibrate your pad and test the buttons and sticks). You can also use the Microsoft DirectX SDK *\Samples \C ++\DirectInput \Joystick* tool.

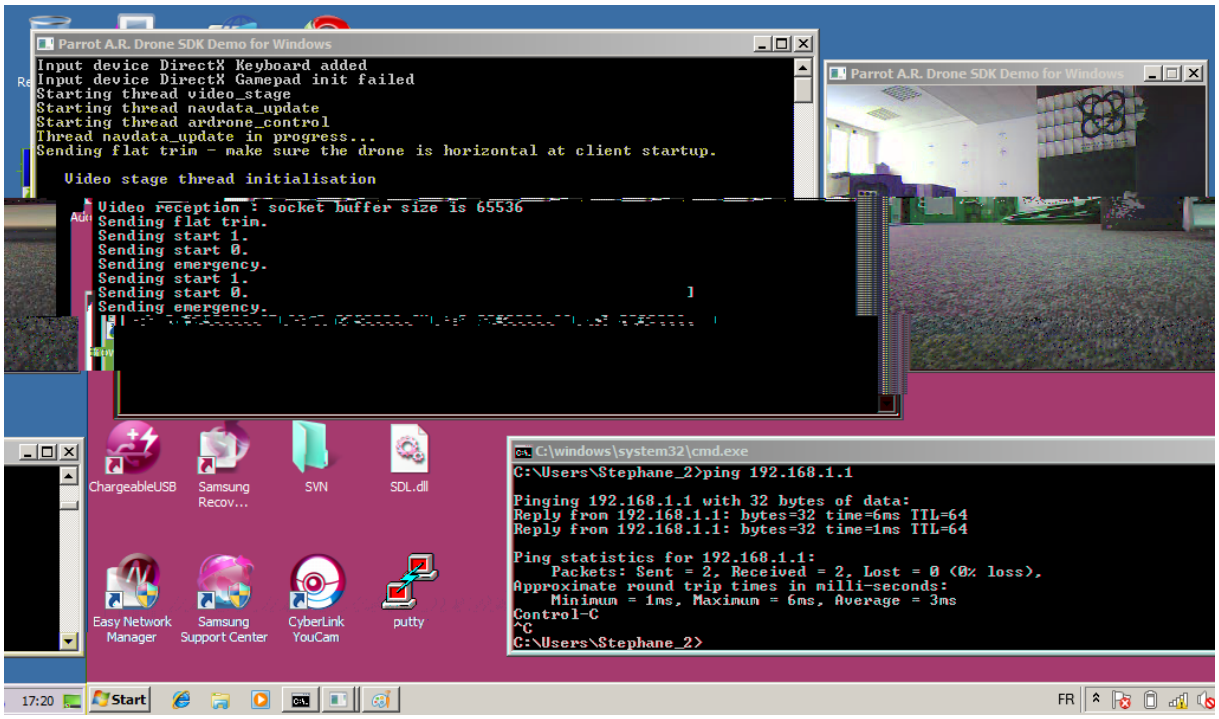
12.3 Compiling the example

Simply generate the solution once the above mentioned settings were done.

12.4 What to expect when running the example

Before running the example, one of the computer network connections must be connected with the drone. Pinging the drone or connecting to its telnet port MUST work properly. The drone MUST NOT be paired with an iPhone (using the drone with an iPhone prevents the drone from accepting connections from any other device, including a PC, unless the unpair button (below the drone) is pressed (which is acknowledged by the drone by making the motor leds blink).

If the network is correctly set, running the example will display a console window showing which commands are being sent to the drone, and a graphic window showing the drone broadcasted video.



You can then control the drone by using your gamepad, or use the keyboard with the following jeys :

Key	Action
8 or **I**	Fly forward
2 or **K**	Fly backward
4 or **J**	Fly leftward
6 or **L**	Fly rightward
A or **+**	Fly up
Q or **-**	Fly down
7 or **U**	Rotate anticlockwise
9 or **O**	Rotate clockwise
Space	Takeoff / Land
Escape or **Tab**	Send emergency[1] / recover from emergency signal
F	Send flat trim command
G	Hover

The example is basic one which simply hangs by saying "Connection timeout" if no connection is possible. Check the network connectivity and restart the application if such thing happens. The video window does not respond if no video is received from the drone. Close the console window to close the application, or deport the SDL window management code from the video processing thread to an independent thread.

12.5 Quick summary of problems solving

P. The application starts, but no video is displayed, or "Connection timeout" appears.
S. The client was to slow connecting to the drone, or could not connect. Restart the application, or check your network configuration if the problem remains.

P. I can't open any source file from the solution explorer
S. Make sure the **`ARDroneLibDir`** and **`Win32ClientDir`** macros are set in the Properties Manager

P. I get the "Windows.h : no such file or directory" message when compiling.
S. Make sure the Windows SDK was correctly installed. A light version should be installed along with VC Express.

P. I get the "Cannot open input file 'dxguid.lib' message when compiling.
S. Make sure the DirectX SDK was correctly installed

P. I get the "Error spawning mt.exe" message when compiling.
S. There is a problem with the Windows SDK installation. Please reinstall.

13 Other platforms

13.1 Android example

Please refer to the *<SDK>/Examples/Android/ardrone* directory, and its two files *INSTALL* and *README* to compile the Android example. It was successfully tested (controls and video display) on a rooted Google / HTC Nexus One phone, using NDK r3.