INSA Toulouse

PROJECT RELEASE REPORT

Autonomous Drone Guide

Engineers:
Mathieu OTHACEHE
Florian VOLKMANN
Duc Ngoc HUYNH
Fadi HALLOUMI

Supervisors:
Mr.Pierre-Emmanuel HLADIK
Ms.Dinnah MCCARTHY
Mentor :
Mr.Didier LE BOTLAN

Autonomous target-guided navigation in low obstacle enviroment

Halloumi Fadi
`halloumi@etud.insa-toulouse.fr`
Othacehe Mathieu
`othacehe@etud.insa-toulouse.fr`
Volkmann Florian
`volkmann@etud.insa-toulouse.fr`
Huynh Duc Ngoc
`huynh@etud.insa-toulouse.fr`

January 11, 2013

Abstract

Being fifth year students at INSA Toulouse, we saw that many visitors had difficulties to find the Electrical and Computer Engineering Department (DGEI) and they were late for the meeting. At the end of our study in Critical and Embedded System, we have an opportunity to work with a drone so we decided to make our project more useful. Our Pink team is working on developing a UAV that can go to DGEI wherever it was by searching a signal power of an Xbee emitter (or a GPS).

Contents

1 Introduction

Our team has been in charge to develop a guide to help people find their way at INSA. Our drone is able to converge to a target thanks to GPS coordinates. It is naturally able to fly in autonomy without human interactions. // autre doc, pres plan

2 General project presentation

2.1 Goal of our project

Our project, proposed by our tutor D. Le Botlan is titled “Adptive Flight”. It is composed of two main parts.

The first one is about target detection and autonomous flight.
The other one is obstacle avoidance.

Actually, our UAV must be able to detect a target, flight to this point and above all, avoid the human obstacles.

2.2 Documentation & deliverables

In the beginning of the project we have created a website on our project. It can be accessed at :

`https://sites.google.com/site/projetsecinsa/projets-2012-2013/projet-de-la-mission-pink`

We have also written technical documentation : PMP, SRS and STD. They can be downloaded in the “Sources” section of our website.

Regarding software production, we have used the versionning manager git. Our code is hosted at this url :

`https://github.com/insadrone/InsaQuadDrone`

Finally, various documents, unsorted, like mid-term presentations, hardware benchmarks can be seen in the included folder named “google_drive_dump”

3 Hardware considerations

3.1 Parrot 2.0

The Parrot AR.Drone is a radio controlled flying quadrotor helicopter built by the French company Parrot. The drone is designed to be controlled by any electronic device having wi-fi connection and sufficient ressource to run a control software. Only android, iOS and windows have offically distributed control applications. In other hand, only Windows and Linux OS are supported as a development platform. The choice of the department for this drone as a base for the project is due to the harmless character of this quadrotor. Two versions of the Parrot AR.Drone exists. For our project only the new version(Ar.Drone 2.0) is supported.

3.1.1 Ressources

Many ressources could be found on this UAV and this what make it one of the best choices:

- `http://www.parrot.com/fr`
- `http://ardrone.parrot.com/parrot-ar-drone/usa/`
- `http://projects.ardrone.org/`
- `http://devzone.parrot.com/`

3.1.2 Specification

Here a quick overview of the general specification of the drone:

- Autonomy : Approximately 12 minutes, recharging time of 1h30.
- Maximum range : 50m average, 100m in a wide-open space with few Wi-Fi waves.
- Maximum altitude : 6m is the stability limitation, 50m the wi-fi limitation (which could be hacked with wi-fi booster to go up to 75m).
- Maximum additional supported weight : 80g is the limit of stability, 100g is the limit of motors propulsion.
- Maximum speed : 18 km/h.
- Maximum supported wind speed : 2km/h.

3.1.3 Hardware

The hardware reference is as follow:

* AR.Drone 1.0 : carte Mykonos, processeur ARM926EJ-S rev 5 (v5l) Wi-Fi: AR6000 Memory: 128MB RAM
* AR.Drone 2.0 : Mykonos2 card, Processor OMAP 3640 1GHz 32 bit ARM Cortex A8 with a video DSP 800MHz TMS320DMC64x

3.1.4 Software

The board has an embedded Linux with these reference : //

* Linux myhost 2.6.27.47-parrot-01227-g93dde09 #1 preempt Fri Jul 2 15:23:06 CEST 2010 armv5tejl GNU/Linux
* Linux 2.6.32 kernel: Linux uclibc 2.6.32.9-g0d605ac #1 preempt Fri Apr 6 12:01:59 CEST 2012 armv7l GNU/Linux

This embedded linux contains these basic packages :

- BusyBox
- Mtdutils
- zlib
- ethtool
- propcps
- udev
- dsp bridge
- lcml dsp codec
- wireless tools
- exif
- iptables
- usbmodeswitch
- lsusb
- alsa lib
- barry
- busydroid
- webkit

We added multiple modules in order to be able to communicate with usb port:

- cdc-acm
- usbserial
- ftdi_sio

you need to cross-compile these modules you can use the following step :

1. Download an ARM Cross-compiler you can find one on the website of Mentor Graphics (successor of Code sourcery) `http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/arm-gnu-linux`
2. Install the cross-compiler :
```
#: chmod +x arm-2012.03-57-arm-none-linux-gnueabi.bin
#: ./arm-2012.03-57-arm-none-linux-gnueabi.bin
```
3. Compile the module :
```
#: /opt/CodeSourcery/Sourcery_CodeBench_Lite_for_ARM_GNU_Linux/bin/arm-
   none-linux-gnueabi-gcc -march=armv7-a toto.c -o toto.elf
```

3.1.5 Development

3.2 Arduino

We have used the arduino board uno, without any shield or add ons. The electrical schematics is just below.

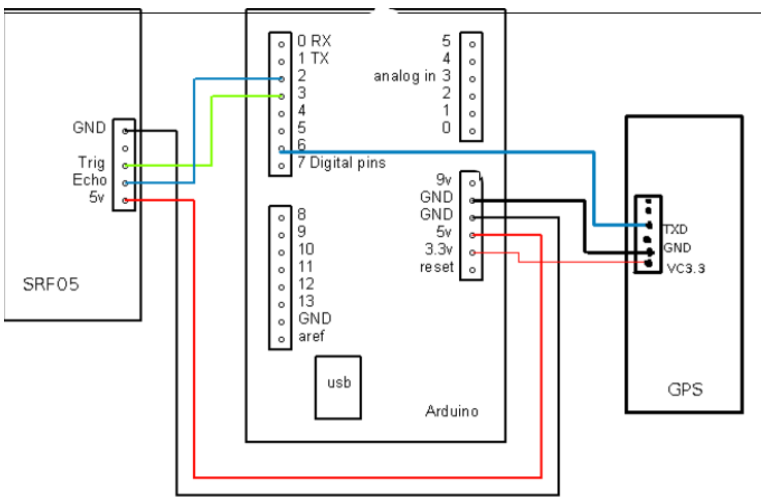


Figure 3.1: Arduino schematics

Then, to upload a specific code into the board, the arduino application is needed. We used the 1.0.1 version of the application.

The code uploaded is :

```
#include <SoftwareSerial.h>

// GPS PINS
#define SoftrxPin 2
#define SofttxPin 7
// SRF PINS
#define ECHOPIN1 3                              // Pin to receive echo pulse
#define TRIGPIN1 4                               // Pin to send trigger pulse
#define ECHOPIN2 5                              // Pin to receive echo pulse
#define TRIGPIN2 6                               // Pin to send trigger pulse

// initialisation de la liasion serie
SoftwareSerial gps = SoftwareSerial(SoftrxPin, SofttxPin);

int incomingByte = 0;           // Pour stocker les donnees entrantes
// Stocke la chaine GPS
char line[300] = "";
// Position dans la chaine
int index = 0;
// La chaine recherchee
char commandeGPR[7] = "$GPRMC";
// Chaine ok
int commande_ok = 0;
int i,j = 0;

int readExtractGpsGPRMC(){
// Envoie des donnees que quand on en recoit
  int ret = 0;

  while (gps.available () > 0)
    {
      // On lit le byte:
      incomingByte = gps.read ();
      line[index] = incomingByte;

      if (incomingByte == 10)
        {

          // Verifie si la chaine est bien de type $GPR
          for (int i = 0; i < 4; i++)
            {
              if (line[i] != commandeGPR[i])
                {
                  commande_ok = 1;
                  break;
                }
            }
          //-------------------------------------------------

          // Si on a recupere la bonne chaine, on l'affiche
          if (commande_ok == 0)
            {
              for (int pc = 0; pc <= index; pc++)
                {
                  Serial.write (line[pc]);
                }
             ret = 1;
            }
          //---------------------------------------------------
          index = 0;
          commande_ok = 0;
        }
      else
        {
          index++;
        }
    }
  return ret;
}


float calculateDistance(int pinEcho, int pinTrig){
  digitalWrite(pinTrig, LOW);            // Set the trigger pin to low for
     2uS
  delayMicroseconds(2);
  digitalWrite(pinTrig, HIGH);          // Send a 10uS high to trigger
     ranging
  delayMicroseconds(10);
  digitalWrite(pinTrig, LOW);           // Send pin low again
  int distance = pulseIn(pinEcho, HIGH); // Read in times pulse
  return distance/58.0;                  // Calculate distance from time of
      pulse
}


void setup(){
  Serial.begin(9600);// ouvre le port serie et regle le debit a 9600 bps
  gps.begin(9600);// pareil pour les ports digitaux
  pinMode(ECHOPIN1, INPUT);
  pinMode(TRIGPIN1, OUTPUT);
}

void loop(){
  unsigned long time;
  int r = readExtractGpsGPRMC();

   while(r == 0) {
     r = readExtractGpsGPRMC();
   }
   time = millis();
   gps.flush();

   while ((millis() - time) < 600) {
     Serial.print("$SRFR,");
     Serial.println(calculateDistance(ECHOPIN1,TRIGPIN1));
   }
}
```

Basically, in the main loop, we are extracting GPS strings and srf datas.

About GPS data, we just have to extract the GPRMC string of the 6 lines we receive every second. We need to test the beginning of the string. If is equal to "$GPRMC" string, we store the string and send it via serial connection to the UAV.

To catch SRF data, it is necessary to read the datasheet, and do the following tasks :

1. Set the trigger pin to low for 2uS
2. Send a 10uS high to trigger ranging
3. Read the time at high level, it corresponds to the duration of ultrasonic roundtrip.
4. Dividing this duration by 58, we obtain the distance to obstacle

We have to send this distance to the UAV, like for the GPS strings.

Using separately ultrasonic sensors and GPS works perfectly well. However it is difficult to use both components at the same time. The GPS extraction is using SoftwareSerial class and all incoming data are stored in a buffer of this class. Data's are written every 1 second, and if we are listening this buffer all the time (except for extraction), this buffer won't overflow. But, if data are written in this buffer while we are capturing ultrasonic wave, the writting will overflow the buffer and cause the loss of some information.

It is impossible to multithread this program, so we have to schedule manually the execution of GPS and sensor listening. We know GPS data are incoming every second, and the execution of gps extraction last 100 ms, so we dedicate 600 ms to ultrasonic listening, and then we wait for GPS data to arrive.

This solution is not very satisfying because we take a margin of 300ms, and we don't have SRF data during 400ms, but we didn't find a better way to combine GPS and SRF sensors.

3.3 Android

3.4 Ultrasonic Sensor

3.5 Bluetooth

This part explains how to connect and read data from an android smartphone that sends information via bluetooth. ShareGPS, an application running on the phone is sending its GPS coordinate to the computer. The aim of the document is to give the process to get this coordinate.

First step: with the smartphone

- Turn on the bluetooth and the GPS
- Launch ShareGPS and share coordinate via bluetooth
- Make the bluetooth visible by other devices

Second step: with the computer on Linux

- Turn on the bluetooth and open a terminal (you may need to be root) Scan devices with: "hcitool scan" -> Write down the MAC address of the corresponding device
- Find the good channel which gives GPS coordinate with: "sdptool records MAC_ADDR" -> This command displays all channels and their functions -> Look for the one called "ShareGPS" and write down the corresponding channel
- Create the connection with: "rfcomm bind X MAC_ADDR CH" -> X: positive integer corresponding to the rfcomm you want to bind -> MAC_ADDR: MAC address of the smartphone -> CH: channel of the ShareGPS application

You can now check that /dev/rfcommX exists. The last step is to read data. To kill it use the command "rfcomm release rfcommX" or "rfcomm release all".

Third step: read data

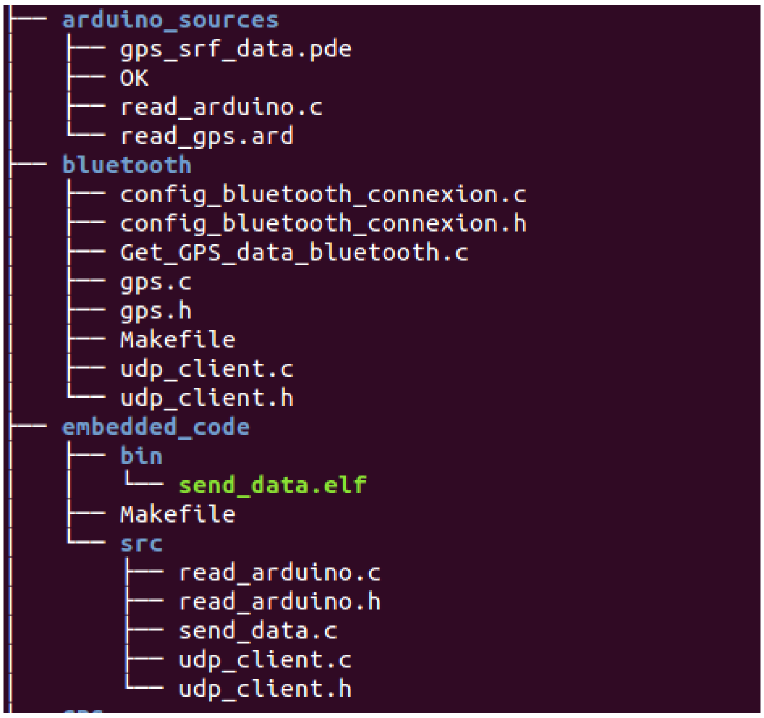
- Using PuTTY: connect via serial to /dev/rfcommX with a 9600 baudrate
- You can also use a self-made program

Source: `http://www.thinkwiki.org/wiki/How_to_setup_Bluetooth`

4 Software producted

4.1 Repository architecture

The general architecture of our code repository is explained in this tree :



```
├── arduino_sources
│   ├── gps_srf_data.pde
│   ├── OK
│   ├── read_arduino.c
│   └── read_gps.ard
├── bluetooth
│   ├── config_bluetooth_connexion.c
│   ├── config_bluetooth_connexion.h
│   ├── Get_GPS_data_bluetooth.c
│   ├── gps.c
│   ├── gps.h
│   ├── Makefile
│   ├── udp_client.c
│   └── udp_client.h
├── embedded_code
│   ├── bin
│   │   └── send_data.elf
│   ├── Makefile
│   └── src
│       ├── read_arduino.c
│       ├── read_arduino.h
│       ├── send_data.c
│       ├── udp_client.c
│       └── udp_client.h
```

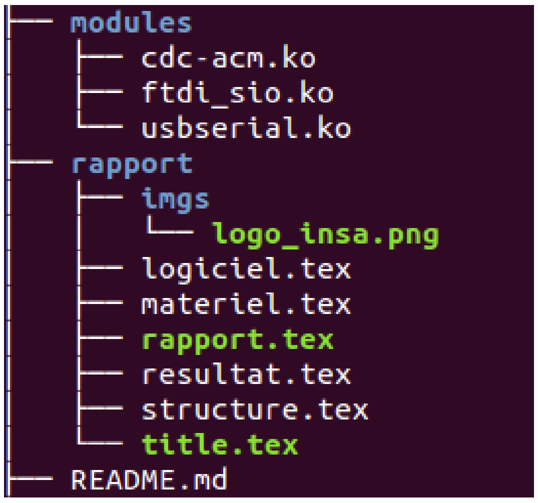
Figure 4.1: 1st part of the repository

In arduino_sources folder, all the code related to the arduino board can be found. The software used to program the arduino is expecting .pde or .ard files.

In the following parts, we will provide more details for each files.

The bluetooth folder contains C sources et headers used on the target to catch informations sent by the smartphone.

The embedded code section includes all the programs we created to run on the UAV embedded linux.



```
├── modules
│   ├── cdc-acm.ko
│   ├── ftdi_sio.ko
│   └── usbserial.ko
├── rapport
│   ├── imgs
│   │   └── logo_insa.png
│   ├── logiciel.tex
│   ├── materiel.tex
│   ├── rapport.tex
│   ├── resultat.tex
│   ├── structure.tex
│   └── title.tex
├── README.md
```

Figure 4.2: 2nd part of the repository

All the compiled modules we added on the embedded linux can be found in the module folder.

The “rapport” directory contains the sources of the documentation you are reading !

Finally, the most important folder is the one named sdk_apps. It contains one application named auto_flight. Like most of the c projects, you can find a Makefile, a bin/ and a src/ folder. The Makefile is configurated to create the application binary and to move it to the bin/ folder. You can find additionnal informations about the Makefile in the appedices.

About the src/ folder :

The most important C file is named ardrone_testing_tool.c. This file is making the link between our application and the ARdrone library. Moreover, it is launching and joining all our threads.

The Auto/ folder contains code related to the auto_control thread.

The Avoidance/ folder contains code related to the avoidance thread.

The Comm/ folder contains code related to the receive_gps thread.

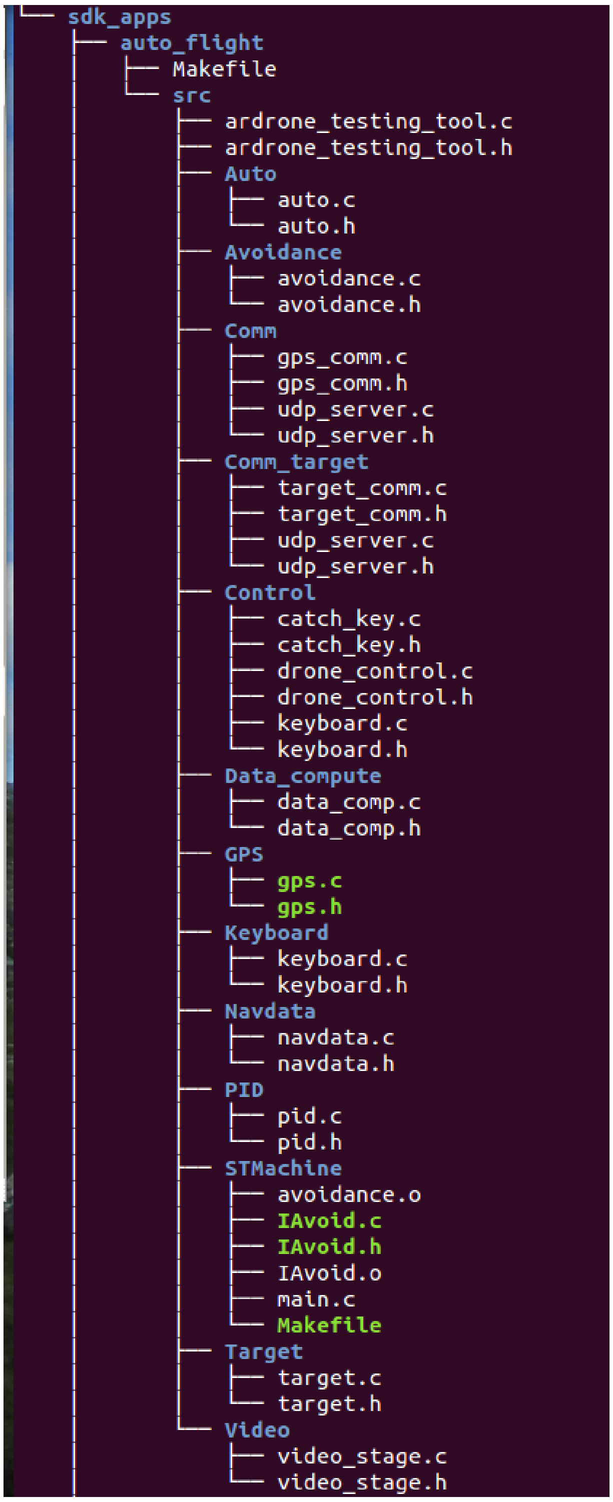
The Comm_target/ folder contains code related to the gps_target thread.

The Control/ folder contains an small library we wrote to handle UAV travelling.

The GPS/ folder contains algorithms used to manipulate GPS strings and make distance and angle calculation.
The Navdata/ folder contains the fonctions used to read and store navdata sent by the UAV.

The STMachine/ folder contains the fonctions generated by SCADE KCG compiler.

The Target/ folder contains code related to the target thread.



```
└── sdk_apps
    ├── auto_flight
    │   ├── Makefile
    │   └── src
    │       ├── ardrone_testing_tool.c
    │       ├── ardrone_testing_tool.h
    │       ├── Auto
    │       │   ├── auto.c
    │       │   └── auto.h
    │       ├── Avoidance
    │       │   ├── avoidance.c
    │       │   └── avoidance.h
    │       ├── Comm
    │       │   ├── gps_comm.c
    │       │   ├── gps_comm.h
    │       │   ├── udp_server.c
    │       │   └── udp_server.h
    │       ├── Comm_target
    │       │   ├── target_comm.c
    │       │   ├── target_comm.h
    │       │   ├── udp_server.c
    │       │   └── udp_server.h
    │       ├── Control
    │       │   ├── catch_key.c
    │       │   ├── catch_key.h
    │       │   ├── drone_control.c
    │       │   ├── drone_control.h
    │       │   ├── keyboard.c
    │       │   └── keyboard.h
    │       ├── Data_compute
    │       │   ├── data_comp.c
    │       │   └── data_comp.h
    │       ├── GPS
    │       │   ├── gps.c
    │       │   └── gps.h
    │       ├── Keyboard
    │       │   ├── keyboard.c
    │       │   └── keyboard.h
    │       ├── Navdata
    │       │   ├── navdata.c
    │       │   └── navdata.h
    │       ├── PID
    │       │   ├── pid.c
    │       │   └── pid.h
    │       ├── STMachine
    │       │   ├── avoidance.o
    │       │   ├── IAvoid.c
    │       │   ├── IAvoid.h
    │       │   ├── IAvoid.o
    │       │   ├── main.c
    │       │   └── Makefile
    │       ├── Target
    │       │   ├── target.c
    │       │   └── target.h
    │       └── Video
    │           ├── video_stage.c
    │           └── video_stage.h
```

Figure 4.3: 3rd part of the repository

Thread name	Period	Description
ardrone_control	2	Library related thread
navdata_update	20	Library related thread
auto_control	20	Thread allowing manual command of the UAV
receive_gps	50	Thread used to received gps coordinated sent by the UAV
avoidance	60	Thread used to perform autonomous obstacle avoidance
gps_target	50	Thread used to received gps coordinated sent by the target
gps_target	50	Thread used to perform autonomous convergence to the target

This table summarizes all the thread launched in our application. In the following sections, we will detail the operations of each threads.

4.2 The auto thread

The auto thread is mainly based on the movement API we developped with the Brown team.

4.2.1 Movement API

This API is an overlay of the Parrot API, it allows more intuitive UAV control. It also provide movement quatification, which means you can quantify an order by speed, distance or duration. For example, you can emit orders like “go forward on 300 cm”.

Les mouvements élémentaires

Les commandes de déplacement sont définis dans le fichier Control/drone_control.c, elles ont toutes le même format :

The travelling commands are all defined in the file Control/drone_control.c, they have the same format :

```
C_RESULT ordre (void *arg)
```

The list of available travellings :

- turn_left
- turn_right
- forward
- backward
- up
- down
- right
- left
- stop

4.2.2 Send an order

The above order can be passed to both functions depending on whether you want to make a move that will be termed “elementary”, or a longer trip by specifying the distance.

Elementary move

An elementary movement allows for a sudden displacement of around 10 cm. You must use the following function:

```
C_RESULT small_move(ORDER* order)
```

Sample code :

```
small_move(turn_right);
```

Long move

All orders to send move commands accept an argument of type void *. This argument must be cast to void * but available commands manage only the type arguments mov_t which includes different arguments :

```
typedef struct mov_t{
  int32_t power;      //engine power between 0 and 100
  int32_t distance;  //distance in cm
  int32_t time;       //time in usec
}mov;
```

Before sending orders, it's important to fill correctly this structure. Unused fields must be initialized to -1. To send an order, this function must be used :

```
C_RESULT send_order(ORDER* order, void *arg)
```

Sample code :

```
/* 30% on 70 cm*/
mov mv = {30, 70, -1};
send_order(backward, &mv);
```

4.2.3 Using of this API

In the auto thread, we need to control the UAV with the keyboard of the station. This automatic control is really useful in case of problems occuring during the automatic control.

Basically, we are just running a scanf in an infinite loop. Depeding on the key pressed, we just have to call the wright function of the API.

```
  while (1) {
    usleep(100000);
    scanf("%c", &c);
    printf("%c\n",c);
    switch(c){
    case 'f':
      small_move(forward);
      break;
    case 'b':
      small_move(backward);
      break;
    case 'u':
      small_move(up);
      break;
    case 'd':
      small_move(down);
      break;
    case 'o':
      //small_move(left);
      printf("Batt␣:%d\n",sauv_ndata.bat_level_current);
      break;
```

Very helpful functionalities are battery control, by pressing the "o" key, and recover from emergency m ode by pressing "x". All the other command are pretty basic ("land", "go up", "go down").

4.2.4 Storing navdatas

We also use this thread to store the navdata received. To achieve this goal, it is necessary to declare three functions, defined in the ARDrone library :

```
/* Initialization local variables before event loop  */
inline C_RESULT auto_navdata_client_init( void* data )

/* Receving navdata during the event loop */
inline C_RESULT auto_navdata_client_process( const navdata_unpacked_t*
    const navdata )

/* Relinquish the local resources after the event loop exit */
inline C_RESULT auto_navdata_client_release( void )
```

We just keep the informations we have to use later, like battery level, control state, altitude or psi angle.

```
sauv_ndata.psi_current = nd->psi / 1000;
sauv_ndata.bat_level_current = nd->vbat_flying_percentage;
sauv_ndata.ctrl_state_current = nd->ctrl_state;
sauv_ndata.tag_detected = nv->nb_detected;
sauv_ndata.tag_tab = nv->camera_source;
sauv_ndata.alt = nd->altitude / 1000.0;
```

We store all these informations in a global structure that will be accessed by the other threads.

4.3 The avoidance thread

This thread is mainly based on SCADE generated code. The statechart modeling the avoidance is printed below :

Then, the KCG compiler gave us some independent code, that we included in our project.

```
switch (AvoidMachine_state_sel) {
    case SSM_st_Travelling_AvoidMachine :
      outC->AvoidMachine_reset_act = inC->obstacle_detected;
      break;
    case SSM_st_Hovering_AvoidMachine :
      if (br_1_guard_AvoidMachine_Hovering) {
        outC->AvoidMachine_reset_act = 1;
      }
      else {
        outC->AvoidMachine_reset_act = br_2_guard_AvoidMachine_Hovering;
      }
      outC->init = 0;
      break;
    case SSM_st_Avoidance_up_AvoidMachine :
      if (inC->obstacle_detected) {
        outC->AvoidMachine_reset_act = 1;
      }
      else {
        outC->AvoidMachine_reset_act = br_2_guard_AvoidMachine_Avoidance_up
            ;
      }
      outC->init2 = 0;
      break;
```

The top-level function is defined in the file IAvoid.h.

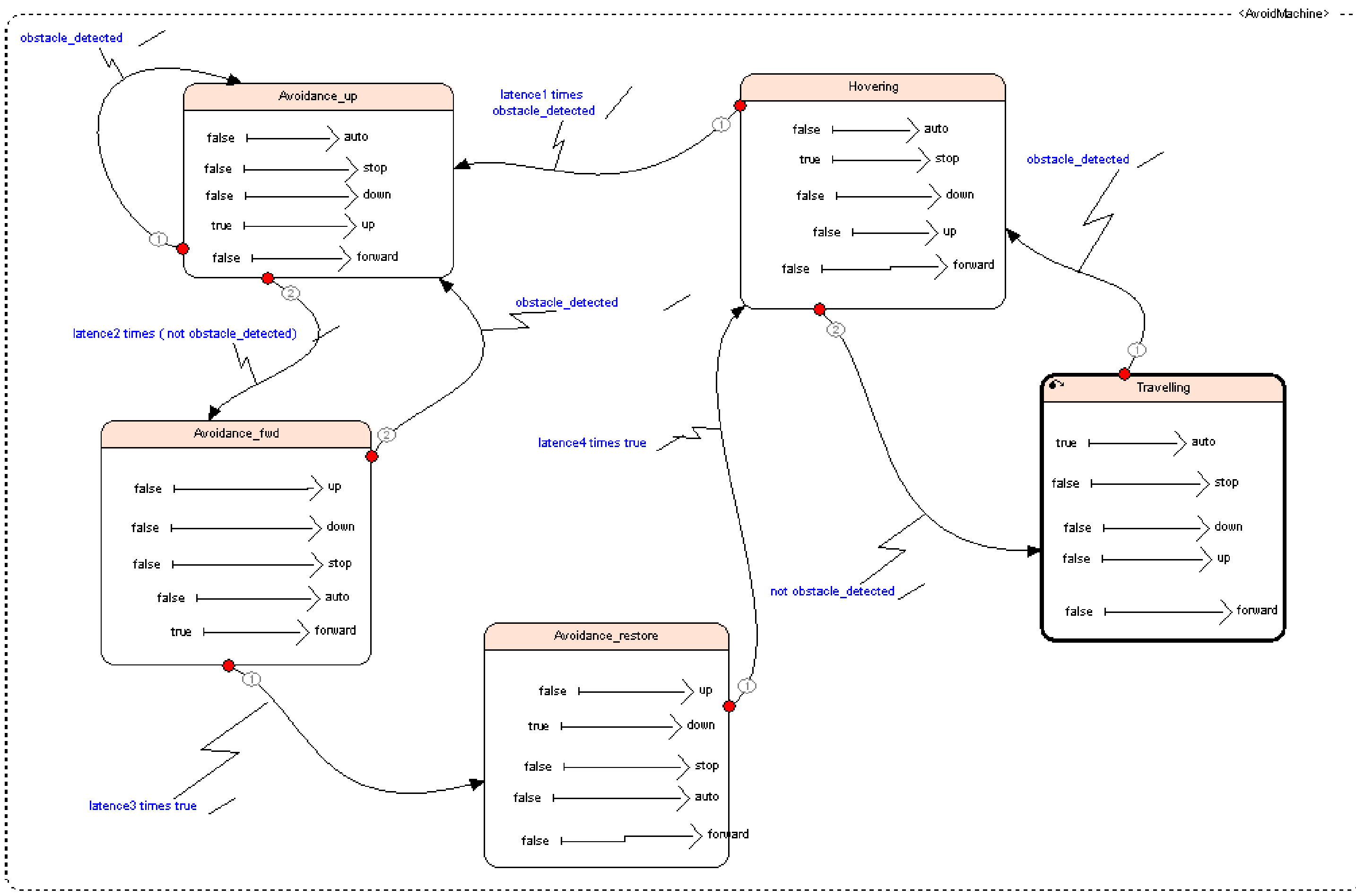


Figure 4.4: SCADE statechart

```
extern void IAvoid(inC_IAvoid *inC, outC_IAvoid *outC);
```

The inputs of this statechart are a boolean named “obstacle_detected” (its value is 1 if there is an obstacle and 0 otherwise) and 4 integers describing latencies between transitions.

The outputs are 5 booleans : 4 orders (up, down, stop, forward) and a neutral order (if there is nothing to do).

Each call to the function represent a cycle. We have planned to execute a cycle each 10ms. The following code is illustrating the use of the auto-generated function :

```
DEFINE_THREAD_ROUTINE(avoidance, data) {

  comm_datas datas;
  double dangerThreshold=100;
  int detection;
  double average_left;
  int ret;

  init_array_obstacle_pos2();

  IAvoid_reset(&output);
  input.obstacle_detected = 0;
  input.latence1 = 100;
  input.latence2 = 100;
  input.latence3 = 300;
  input.latence4 = 100;

 while (1) {
   usleep(10);
   if (auto_ready) {
     //get srf datas
     datas = get_comm_datas();

     ret = average_obstacle_pos2(&datas.srfr, &average_left);
     printf("Moyenne␣capteur(%d)␣:␣%f\n", ret, average_left);

     //check threshold
     if (average_left < dangerThreshold) {
       detection = 1;
     } else {
       detection = 0;
     }

     input.obstacle_detected = detection;
     IAvoid(&input,&output);
     command(output);
     usleep(10000);
   }
 }

  return (THREAD_RET) 0;
}
```

And the command function is :

```
void command(outC_IAvoid comm) {
  mov speed;

  if (comm.up) {
    speed.power = 8;
    send_fast_order(up,(void *)&speed);
  } else if (comm.down) {
    speed.power = 3;
    send_fast_order(down,(void *)&speed);
  } else if (comm.stop) {
    send_fast_order(stop,NULL);
  } else if (comm.auto1) {
    speed.power = 1;
    printf("avance\n");
    send_fast_order(forward,(void *)&speed);
  } else if (comm.forward) {
    speed.power = 1;
    printf("avance\n");
    send_fast_order(forward,(void *)&speed);
  }

}
```

4.4 The receive_gps & gps_target threads

Those threads are designed to receive and store the GPS strings sent by UAV and Target. The receive_gps thread also handle SRF datas sent by the ultrasonic sensor.

We have created a small library with functions to receive udp datas. This library is described in the udp_server.h file.

This function is reading udp datas on port 6444 (sent by the UAV).

```c
/*
   Start listening the buffers sent from uav
*/
int start_comm(void)
{
    udp_struct udp_uav;
    int msglen_uav;

    if(udpserver_init(&udp_uav,UDP_UAV,1)) diep("udp_UAV␣init");


    while (start_listen) {

        do {
            msglen_uav = udpserver_receive(&udp_uav, buf_uav, 512);
        } while(msglen_uav<=0);

       record_data(buf_uav);

    }

    udpserver_close(&udp_uav);
    return 0;
}
```

Then, depending on whether it is a sensor or GPS data, we fill a structure.

```c
int record_data(char *buf) {
  char *gprmc_begin = "$GPRMC";
  char *gpgga_begin = "$GPGGA";
  char *srfl_begin = "$SRFL";
  char *srfr_begin = "$SRFR";

  int ret_val = 0;

  if (!strncmp(gprmc_begin,buf,6)) {
    strncpy(ret_datas.gprmc_string,buf_uav,sizeof(ret_datas.gprmc_string));
    ret_val = 1;
  } else if (!strncmp(gpgga_begin,buf,6)) {
    strncpy(ret_datas.gpgga_string,buf_uav,sizeof(ret_datas.gpgga_string));
  } else if (!strncmp(srfl_begin,buf,5)) {
    ret_datas.srfl = atof(buf+6*sizeof(char));
  } else if (!strncmp(srfr_begin,buf,5)) {
    ret_datas.srfr = atof(buf+6*sizeof(char));
  } else {
    printf("ERROR␣NO␣STRING␣DETECTED\n");
  }
  return ret_val;
}
```

The other thread (receiving from target), uses similar functions.

4.5 The target thread

This thread contains functions related to auto-convergence. We have impleted different algorithms, described thereafter.

4.5.1 Turn angle algorithm

When it converges to the target, the UAV need to turn to an absolute angle between 0 et 360 from north. Depending on the UAV angle, it can be better to turn right, or to turn left, to reduce the turning duration.

The UAV will turn to reach the angle given as an argument. We have defined a tolerance. Indeed, because of transmission delays, the UAV can keep turning even if the angle is ok. A tolerance of 5.0 degrees is acceptable and always working.

```
/*
  INPUT: angle    (float)
  tolerance (float)
  This function will make uav turn to the direction of target with an error
      of +- tolerance
*/
void turn_angle2(float target_angle, float tol) {

  fdata sauv_ndata = get_ndata();
  float angle_360 = sauv_ndata.psi_current;
  float angle_inf, angle_sup;

  angle_inf = target_angle - tol;
  angle_sup = target_angle + tol;

  if (sauv_ndata.psi_current < 0) angle_360 = 360 + sauv_ndata.psi_current;
  if (target_angle < 0) target_angle += 360;

  if (target_angle > angle_360) {
     (target_angle - angle_360) < 180 ? send_order(turn_left,NULL) :
        send_order(turn_right,NULL);
  } else {
    (angle_360 - target_angle) < 180 ? send_order(turn_right,NULL) :
       send_order(turn_left,NULL);
  }

  while (!(sauv_ndata.psi_current > angle_inf && sauv_ndata.psi_current <
     angle_sup)) {
    sauv_ndata = get_ndata();
    usleep(100);
  }

  send_order(stop,NULL);
  sleep(1);
}
```

4.5.2 Convergence algorithm 1

Let's describe the first algorithm :

1. Magneto calibration (the UAV has to be flying)
2. Receive GPS datas
3. Turn to wright angle
4. Go forward 2 seconds
5. If target reached, land otherwise, jump to step 3.

```
if ( (landed == 1) && (calibration == 0) ){
  printf("start␣calibration\n");
  calibrate_magneto(NULL);
  sleep(4);
  calibration = 1;
  printf("Calibration␣done\n");
}

if ( (landed == 1) && (calibration == 1) && (mission == 0) ) {
  datas = get_comm_datas();
  extract_coord(datas.gprmc_string,&depart);

  datas_target = get_comm_datas_target();

  if ((check_gps_coord_struc(&depart) > 0) && (check_gps_coord_struc(&
      datas_target.dest) > 0)) {
    navigation(&depart, &datas_target.dest, &distance, &angle, NULL); //&
        relatif_error

    turn_angle2(angle ,5.0);

    if (distance > 5.0){
      fprintf(redir_sortie,"La␣distance␣restante␣est␣%f\n",distance);
      fflush(redir_sortie);
      speed.power = 3;
      send_order(forward,(void *)&speed);
      sleep(3);
    } else {
      send_order(land,NULL);
      printf("LANDING␣\n");
      mission = 1;
    }
  }
}
```

4.5.3 Convergence algorithm 2

Let's describe the second algorithm :

1. Magneto calibration (the UAV has to be flying)
2. Receive GPS datas
3. Turn to wright angle
4. While distance is decreasing, go forward
5. If target reached land, otherwise go to step 3.

```c
if ( (landed == 1) && (calibration == 0) ){
  printf("start␣calibration\n");
  calibrate_magneto(NULL);
  sleep(4);
  calibration = 1;
  printf("Calibration␣done\n");
}

if ( (landed == 1) && (calibration == 1) && (mission == 0) ) {
  datas = get_comm_datas();
  extract_coord(datas.gprmc_string,&depart);

  datas_target = get_comm_datas_target();

  if ((check_gps_coord_struc(&depart) > 0) && (check_gps_coord_struc(&
     datas_target.dest) > 0)) {
    navigation(&depart, &datas_target.dest, &distance, &angle, NULL); //&
       relatif_error

    turn_angle2(angle ,5.0);

    while (last_distance > distance) {
      fprintf(redir_sortie,"La␣distance␣restante␣est␣%f\n",distance);
      fflush(redir_sortie);
      speed.power = 3;
      send_order(forward,(void *)&speed);
      last_distance = distance;
      sleep(1);
    }

    if (distance < 10.0){
      send_order(land,NULL);
      printf("LANDING␣\n");
      mission = 1;
      fflush(redir_sortie);
      exit(0);
    }
  }
}
```

4.6 GPS algorithms

4.6.1 GPRMC string

After receive GPRMC strings sent from UAV, at the ground station, we can extract some necessaire informations such as coordinates GPS : latitude et longitude and data status.

The GPRMC sentence consists of twelve comma-delimited words:

```
$GPRMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,ddmmyy,x.x,a*hh
1    = UTC of position fix
2    = Data status (V=navigation receiver warning)
3    = Latitude of fix
4    = N or S
5    = Longitude of fix
6    = E or W
7    = Speed over ground in knots
8    = Track made good in degrees True
9    = UT date
10   = Magnetic variation degrees (Easterly var. subtracts from true course)
11   = E or W
12   = Checksum
```

First, we check if Data status is valide (A) or not (V). If status is not valide, coordinate GPS will be set -1. Then we only interest in Latiude (3) and Longitude(5) data. We know also that the data of (4) and (6) are always N and E so the coordinate GPS is positive. For others informations, we don't count in.

```
void extract_coord( char str_gps[], struct gps_coordinate *point )
{
 char delims[] = ",";
 char *result;
 result = malloc(sizeof(char));
 int j,i = 0;
 result = strtok( str_gps, delims );

 char *degree ;
 degree = malloc(sizeof(char));
 char *minute ;
 minute = malloc(sizeof(char));

 while( i<6 )
   {
     i++;
     result = strtok( NULL, delims );
     //printf( "result is %d \"%s\"\n", i, result );
     if (!strcmp(result, "V") )//sortie la boucle quand GPS ne fonctionne
        pas
    { i = 7;
      point->longitude = -1.0;
      point->latitude =  -1.0;
      printf( "coordinates␣undetermined␣\n" );
    }

     if (i==3) //effecter la valeur de latitude
    {
      for (j=0;j<2;j++) //extract degree
        {
          degree[j] = result[j];
        }

      while (j< strlen(result)) //extract minute
```

```
        {
          minute[j-2] = result[j];
          j++;
        }

      point->latitude = atof(degree) + atof(minute)/60.0 ;    /      if (
         point->latitude > 90.0 || point->latitude < -90.0) {
        point->latitude = -1.0;
        point->longitude = -1.0;

      }
      //printf( "latitude %f\n", point->latitude);
    }

     if (i==5) //effecter la valeur de longitude
    {
      for (j=0;j<3;j++) //extract degree
        {
          degree[j] = result[j];
        }

      while (j< strlen(result)) //extract minute
        {
          minute[j-3] = result[j];
          j++;
        }

      point->longitude = atof(degree) + atof(minute)/60.0 ; //convert into
         degree
      if (point->longitude > 180.0 || point->longitude < -180.0) {
        point->latitude = -1.0;
        point->longitude = -1.0;
      }

          }
   }
}
```

4.6.2 Navigation

Bearing

Bearing is the angle measured in a clockwise direction from the north line. Thanks to this, we can know the direction to reach the target. The formula is given :

$$\theta = atan2(sin(\Delta\lambda).cos(\phi2), cos(\phi1).sin(\phi2) - sin(\phi1).cos(\phi2).cos(\Delta\lambda))$$

where ϕ is latitude, λ is longitude.

```
//calcul direction // North

y = sin(dest->longitude - depart->longitude) * cos(dest->latitude);

x = cos(depart->latitude)*sin(dest->latitude) - sin(depart->latitude)*cos(
   dest->latitude)*cos(dest->longitude - depart->longitude);
```

```
*angle = (atan2(y,x) * 180.0/M_PI);
```

Distance

We uses the ‘haversine’ formula to calculate the distance between two points – that is the shortest distance over the earth’s surface.

Haversine formula:

$$a = sin^2(\Delta\phi/2) + cos(\phi1).cos(\phi2).sin^2(\Delta\lambda/2)$$

$$c = 2.atan(\sqrt{a}, \sqrt{(1-a)})$$

$$d = R.c$$

where ϕ is latitude,
λ is longitude

```
 // calcul distance
     x = sin((dest->latitude - depart->latitude)/2) * sin((dest->latitude
        - depart->latitude)/2) + sin((dest->longitude - depart->longitude)
        /2)*sin((dest->longitude - depart->longitude)/2) * cos(depart->
        latitude)*cos(dest->latitude);
    y = 2 * atan2(sqrt(x),sqrt(1-x));
    *distance = RAYON *y;
```

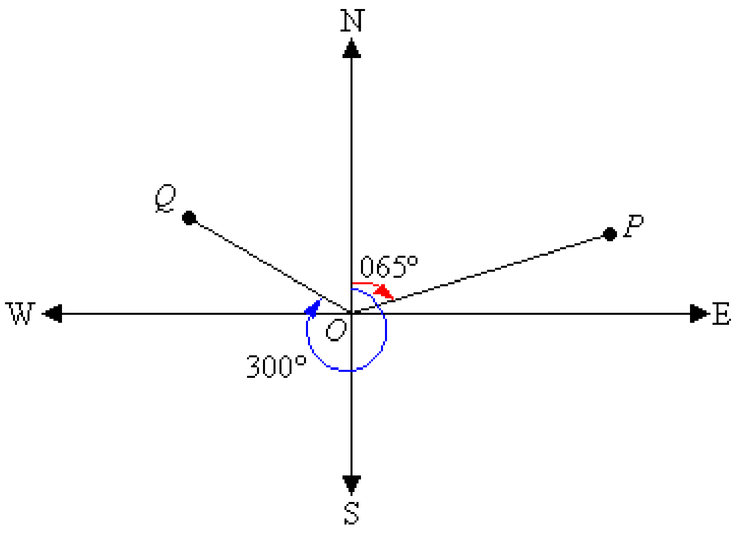


Figure 4.5: Bearing angle

The point O is position of UAV, if we want to go to the point Q, we must turn to left an angle of 60° relative to the north. According to the formula above, it give us a negative angle -60° that means the point of destination is on the left. Another case, we have an angle of 65° on the right if we want to reach to the point P. And the Haversine formula will also give us the distance of OQ or OP.

5 Final results

6 Conclusion

Conclusion