Spiri Application Programming Interface Reference Guide

Pleiades Robotics

The Spiri Application Programming Interface is a set of commands that allow access to the flight controller. These commands allow reading of sensor data as well as a set of defined flight controls. This set of commands can be accessed in the following languages: C/C++, Python, node.js and Java. The languages currently with the most support are C/C++ and Python. This document will describe the Python and C/C++ parts of the API. It will also describe the structure of packets that are sent to and received from the command processing server.

Python

All the Python functions to control Spiri pass their requests to the command processing server on Spiri. For each function called, the Python client generates a packet with the appropriate data and sends it to the processing server. The server then responds to the request with another packet that the client parses and provides results to the user.

All packets (sending and response) have a header as shown below (start of the packet is to the left):

Start of packet character	Start of packet character	Packet size	Function code

Start character: 8 byte value, normally '$'. All packets have two start characters ie "$$"
Packet size: 4 byte integer value that is the overall size of the packet, including the header
Function code: 4 byte integer value that is a special code assigned to the function that has sent the packet

All packets end with an end of packet character, which is '^'.

Before the functions can be called to access Spiri, a Spiri object has to be created from the Spiri class in Python. The IP address of the Spiri we want to connect to is passed to the Spiri class, at the point of creation of the Spiri object. For example to connect to a Spiri on the network with ip address

192.168.1.102, we would create the object as follows:

```
from spiri import *
spiri1 = Spiri('192.168.1.102') #Using Spiri's address on the network
```

If our code is running right on Spiri, we create the Spiri object as follows:

```
from spiri import *
spiri1 = Spiri('127.0.0.1') # Spiri is localhost
```

This scheme gives the advantage of being able to control multiple Spiri's from a centralized location (which might even be one Spiri controlling others!).

Functions and their packets

1. Mode selection

   This function is used to set the mode in which Spiri is.

   Function call: errval = spiri1.select_mode(mode)

   Arguments : Single 4 bytes integer ie mode

   Returns: Single 4 byte integer. Zero if success or negative if an error occured.

   Two modes are suppoorted:

   0 : idle, motors are turned off

   1: stabilized flight

   Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 1	mode	^

   Packet received from server:

$	$	packet_size = 15	FUNCTION_CODE = 1	errval	^

2. Get accelerometer data

   Returns the data from the accelerometer in G, with range max +/-2G

   Function call: errval, accel_data = spiri1.get_accel_data()

   Arguments : None

   Returns: A 4 byte integer error value (0 for success, negative otherwise). An object of type ThreeAxis that holds the x, y and z values of the accelerometer as floats, ie accel_data.x, accel_data.y, accel_data.z

Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 2	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 27	FUNCTION_CODE = 2	errval	x	y	z	^

3. Get Gyroscope data

   Returns the data from the gyro in degrees/second, with range max +/- 200 degrees/second.

   Function call: errval, gyro_data= spiri1.get_gyro_data()

   Arguments : None

   Returns: A 4 byte integer error value (0 for success, negative otherwise). An object of type ThreeAxis that holds the x, y and z values of the gyro as 4 byte floats, ie gyro_data.x, gyro_data.y, gyro_data.z

   Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 3	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 27	FUNCTION_CODE = 3	errval	x	y	z	^

4. Get temperature data

   Returns the temperature in celsius (C).

   Function call: errval, temperature_data = spiri1.get_temperature_data()

   Arguments : None

   Returns: A 4 byte integer error value (0 for success, negative otherwise). A 4 byte float that is the temperature in degrees celsius ie temperature_data.

   Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 4	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 19	FUNCTION_CODE = 4	errval	temperature	^

5. Get Pressure data

   Returns the calibrated data from the pressure sensor in HPA with range 300 to 1100 HPA.

   Function call: errval, pressure_data = spiri1.get_pressure_data()

Arguments : None

Returns: A 4 byte integer error value (0 for success, negative otherwise). A 4 byte float that is the pressure in HPA ie pressure_data.

Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 5	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 19	FUNCTION_CODE = 5	errval	pressure	^

6. Get Range data - returns the range in meters from the acoustic range finder, with data range of 0-6.5 meters.

   Function call: errval, acoustic_range = spiri1.get_acoustic_range_data()

   Arguments : None

   Returns: A 4 byte integer error value (0 for success, negative otherwise). A 4 byte float that is the range in meters ie acoustic_range.

   Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 6	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 19	FUNCTION_CODE = 6	errval	range	^

7. Get attitude roll readings

   Returns the roll angle as calculated from the accelerometer in degrees, the roll angle error from the desired roll angle in degrees, the roll rate in degrees/second and the correction applied to the motors to correct the roll in RPM.

   Function call: errval, roll_angle_accel, roll_angle_error, roll_rate, roll_motor_correction = spiri1.get_attitude_readings()

   Arguments : None

   Returns: A 4 byte integer error value (0 for success, negative otherwise). Four 4 byte floats ie roll_angle_accel, roll_angle_error, roll_rate, roll_motor_correction.

   Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 7	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 31	FUNCTION_CODE = 7	errval	roll_angle_accel	roll_angle _error	roll_rate	roll_motor_ correction	^

8. Get the speed of a single motor

   Returns motor speed of selected motor in RPMs.

   Function call: errval, selected_motor_speed = spiri1.get_speed_of_single_motor(1)

   Arguments : A 4 byte integer for the motor number (motor 1 is front right, then increasing clockwise).

   Returns: A 4 byte integer error value (0 for success, negative otherwise). A 4 byte float that is the motor speed in RPM ie selected_motor_speed.

   Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 8	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 19	FUNCTION_CODE = 8	errval	motor_speed	^

9. Get speed of all motors

   Returns motor speed of all motors in RPMs.

   Function call: (errval, m1, m2, m3, m4)= spiri1.get_speed_of_all_motors()

   Arguments : None

   Returns: A 4 byte integer error value (0 for success, negative otherwise). Four 4 byte floats that are the motor speeds in RPM ie m1, m2, m3, m4.

   Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 9	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 27	FUNCTION_CODE = 9	errval	m1	m2	m3	m4	^

10. Configure the attitude stabilization PID

    Sets the PID coefficients in the x and y axis of the attitude stabilization PID.

Function call: errval = spiri1.configure_stabilization((1,2,3),(4,5,6))

Arguments : Two 3-value tuples of 4 byte floats. The format is (XkP, XkD, XkI), (YkP, YkD, YkI) ie the function is called with the order PDI instead of PID

Returns: A 4 byte integer error value (0 for success, negative otherwise).

Packet sent to server:

$	$	packet_size = 35	FUNCTION_CODE = 10	XkP	XkD	XkI	YkP	YkD	YkI	^

Packet received from server:

$	$	packet_size = 15	FUNCTION_CODE = 10	errval	^

11. Get attitude stabilization PID values

Returns the PID coefficients in the x and y axis of the attitude stabilization PID. Note it is assumed that x and y PID coefficients will always be the same.

Function call: errval, stab_data=spiri1.get_stabilization_data()

Arguments : None

Returns: A 4 byte integer error value (0 for success, negative otherwise). Object of type PID containing PID coefficients ie stab_data.p, stab_data.d, stab_data.i.

Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 11	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 27	FUNCTION_CODE = 11	errval	P	I	D	^

12. Configure the altitude stabilization PID

Sets the PID coefficients of the altitude stabilization PID.

Function call: pid_alt = spiri1.PID(p_value,i_value,d_value)

errval = spiri1.configure_altitude_PID(pid_alt)

Arguments : Object of type PID, initialized with 3 floats for P, I, and D coefficients respectively

Returns: A 4 byte integer error value (0 for success, negative otherwise).

Packet sent to server:

$	$	packet_size = 23	FUNCTION_CODE = 12	P	D	I	^

Packet received from server:

$	$	packet_size = 15	FUNCTION_CODE = 12	errval	^

13. Get altitude stabilization PID values

    Returns the PID coefficients of the altitude stabilization PID.

    Function call: errval, alt_stab_data = spiri1.get_altitude_PID_configuration()

    Arguments : None

    Returns: A 4 byte integer error value (0 for success, negative otherwise). Object of type PID containing altitude PID coefficients ie alt_stab_data.p, alt_stab_data.d, alt_stab_data.i

    Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 13	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 27	FUNCTION_CODE = 13	errval	P	I	D	^

14. Set target altitude

    This function is used to set the altitude to which Spiri should fly.

    Function call: errval = spiri1.set_target_altitude(target_altitude)

    Arguments : Single 4 byte float value for the altitude

    Returns: Single 4 byte integer. Zero if success or negative if an error occured.

    Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 14	target_altitude	^

Packet received from server:

$	$	packet_size = 15	FUNCTION_CODE = 14	errval	^

15. Set attitude

    This function is used to set the attitude at which Spiri should fly.

    Function call: errval = spiri1.set_attitude(attitude_x, attitude_y)

    Arguments : Two 4 byte float values for the x and y components of the attitude in degrees

    Returns: Single 4 byte integer. Zero if success or negative if an error occured.

    Packet sent to server:

$	$	packet_size = 19	FUNCTION_CODE = 15	x	y	^

Packet received from server:

$	$	packet_size = 15	FUNCTION_CODE = 15	errval	^

16. Calibrate accelerometer

This function is used to set the offsets to the accelerometer when it is level.

Function call: errval = spiri1.calibrate_accelerometer(accel_x_offset, accel_y_offset)

Arguments : Two 4 byte float values for the x and y components of the offsets to the accelerometer in Gs.

Returns: Single 4 byte integer. Zero if success or negative if an error occured.

Packet sent to server:

$	$	packet_size = 19	FUNCTION_CODE = 16	x_offset	y_offset	^

Packet received from server:

$	$	packet_size = 15	FUNCTION_CODE = 16	errval	^

17. Get all sensor data

Returns the data from all the available sensors.

Function call: errval, sensor_data = spiri1.get_all_sensor_data()

Arguments : None

Returns: A 4 byte integer. Zero if success or negative if an error occured. A dictionary holding all sensor data of the form:

```
{"accel_data": ThreeAxis(accx, accy, accz),
 "gyro_data": ThreeAxis(gyrx, gyry, gyrz),
 "magnetometer_data": ThreeAxis(magx, magy, magz),
 "range": acoustic_range,
 "pressure": pressure}
```

Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 17	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 59	FUNCTION_CODE = 17	errval	accel_x, accel_y, accel_z, gyro_x,	^

					gyro_y, gyro_z, mag_x, mag_y, mag_z, range, pressure (Note that each of these values are 4 byte floats)	

18. Get magnetometer data

    Returns magnetometer readings in Gauss with a range of -/+2 Gauss.

    Function call: errval, magnetometer_data = spiri1.get_magnetometer_data()

    Arguments : None

    Returns: A 4 byte integer. Zero if success or negative if an error occured. An object of type ThreeAxis that holds the x, y and z values of the magnetometer as floats, ie magnetometer_data.x, magnetometer_data.y, magnetometer_data.z

    Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 18	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 27	FUNCTION_CODE = 18	errval	x	y	z	^

19. Get GPS data

    Returns readings from the GPS

    Function call: errval, x, y, z, UTC_time, HDOP = spiri1.get_GPS()

    Arguments : None

    Returns: A 4 byte integer. Zero if success or negative if an error occured. Five 4 byte floats: ECEF position x,y,z, UTC time, horizontal dilution of principle.

    Packet sent to server:

$	$	packet_size = 15	FUNCTION_CODE = 20	Packet padding = 0 (4 byte int)	^

Packet received from server:

$	$	packet_size = 35	FUNCTION_CODE = 20	errval	x	y	z	UTC_time	HDOP	^

20. Set mode one flight parameters

    Controls the attitude and average motor speed while Spiri is in mode 1.

    Function call: errval = set_Mode_one_flight_parameters(pitch_angle, roll_angle, yaw_angle, rpm)

Arguments : Four 4-byte floats; pitch angle, roll angle and yaw angle in degrees and average motor speed in RPMs

Returns: A 4 byte integer. Zero if success or negative if an error occured.

Packet sent to server:

$	$	packet_size = 27	FUNCTION_CODE = 21	pitch_angle	roll_angle	yaw_angle	rpm	^

Packet received from server:

$	$	packet_size = 15	FUNCTION_CODE = 21	errval	^

C/C++

C/C++ access is currently implemented through the robot hardware access library (RHAL). The RHAL only works for programs running on Spiri. To access Spiri over wifi, one must write their own client code in C/C++ that can connect to the command processing server. Further information on writing your own client will be provided later.

Using the RHAL

To use the RHAL, one must include the header file RHAL.h in their C/C++ project and link their project to libRHAL at compile time.

Functions used to access the robot are listed in RHAL.h. It should be noted that before before any of the functions and control functions is used, the function:

```
int init_access(int reg_flight_ctlr);
```

must be called. This function initializes connectivity to the Spiri hadware. It takes one integer parameter, which can be either 0 or 1, to determine whether the calling program would like the ability to control the flight behaviour of the robot. When the value is 1, then flight control access is granted, but when 0, no flight control is granted. With value zero, the calling program can only read sensors but cannot perform calibration or flight maneuvers. With value 1, all RHAL functions are available.

Below is a sample program that uses the RHAL to perform mode selection

```
/*
```

```
 * Test function: Created by Nicholas Othieno
 * Copyright 2013 Pleiades Consulting Inc
 */
#include <stdio.h>
#include <RHAL.h>
#include <assert.h>

int main()
{
  int retval;
  int mode = 1;

  if(init_access(1) < 0)
  {
    printf("Initialization by test func failed\n");
    return -1;
  }

  //Test writing to the library
  retval = cmdModeSelection(mode);
  shut_access();
  printf("Return value is %d\n", retval);
  assert(retval == CMD_SUCCESS);
  return 0;
}
```

Note that init_access is called with parameter value 1 to give our program flight control status. The funtion cmdModeSelection is then called to select the appropriate mode of the robot.

The above program is compiled with the following command when logged onto Spiri:

```
gcc  ModeSelection.c -lRHAL -o ModeSelection.bin
```

Note that it is linked against the libRHAL library.

Functions

Below are the functions available for communicating with the flight controller and reading sensors. It should be noted that in when calling a function that returns values to pointers, the variable to which pointer points should be created first, then the address to the variable passed as an argument of the function.

1. Mode selection

This function is used to set the mode in which Spiri is.

Function call: int cmdModeSelection(int mode);

Arguments : Single 4 bytes integer ie mode

Returns: Single 4 byte integer. Zero if success or negative if an error occured.

Two modes are suppoorted:

0 : idle, motors are turned off

1: stabilized flight

2. Get accelerometer data

   Returns the data from the accelerometer in G, with range max +/-2G

   Function call: int cmdGetAccelData(float *ACCXsigned, float *ACCYsigned, float *ACCZsigned);

   Arguments : 3 pointers to floats in which the x, y and z accelerometer readings will be stored

   Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual accelerometer results will be stored in the locations pointed to by the argument pointers.

3. Get Gyroscope data

   Returns the data from the gyro in degrees/second, with range max +/- 200 degrees/second.

   Function call: int cmdGetGyroData(float *GyroDataX, float *GyroDataY, float *GyroDataZ);

   Arguments : None

   Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual gyro results will be stored in the locations pointed to by the argument pointers.

4. Get temperature data

   Returns the temperature in celsius (C).

   Function call: int cmdGetTemperatureData(float *TempSensesigned);

   Arguments : Pointer to float

   Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual temperature result will be stored in the location pointed to by the argument pointer.

5. Get Pressure data

   Returns the calibrated data from the pressure sensor in HPA with range 300 to 1100 HPA.

   Function call: int cmdGetPressureData(float *PressureValueRaw);

Arguments : Pointer to float

Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual pressure result will be stored in the location pointed to by the argument pointer.

6. Get Range data - returns the range in meters from the acoustic range finder, with data range of 0-6.5 meters.

   Function call: int cmdGetAcousticRangeData(float *RangeData);

   Arguments : Pointer to float

   Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual range result will be stored in the location pointed to by the argument pointer.

7. Get attitude roll readings

   Returns the roll angle as calculated from the accelerometer in degrees, the roll angle error from the desired roll angle in degrees, the roll rate in degrees/second and the correction applied to the motors to correct the roll in RPM.

   Function call: int cmdGetAttitudeReadings(float *currentYangleACC, float *YangleError, float *Yrate, float *YmotorCorrection);

   Arguments : 4 pointers to float

   Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual attitude results will be stored in the locations pointed to by the argument pointers.

8. Get the speed of a single motor

   Returns motor speed of selected motor in RPMs.

   Function call: int cmdGetSpeedofSingleMotor(int selected_motor, float *PWM_Rate);

   Arguments : A 4 byte integer for the motor number (motor 1 is front right, then increasing clockwise) and a pointers to float.

   Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual motor speed result will be stored in the location pointed to by the argument pointer.

9. Get speed of all motors

   Returns motor speed of all motors in RPMs.

   Function call: int cmdGetSpeedofAllMotors(float *PWM1_Rate, float *PWM2_Rate, float *PWM3_Rate, float *PWM4_Rate);

Arguments : None

Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual motor speed results will be stored in the locations pointed to by the argument pointers.

10. Configure the attitude stabilization PID

    Sets the PID coefficients in the x and y axis of the attitude stabilization PID.

    Function call: int cmdSetPIDStabilizationConfigurationData(float XkP, float XkD, float XkI, float YkP, float YkD, float YkI);

    Arguments : 6 floats. The function is called with the order PDI instead of PID

    Returns: A 4 byte integer error value (0 for success, negative otherwise).

11. Get attitude stabilization PID values

    Returns the PID coefficients in the x and y axis of the attitude stabilization PID. Note it is assumed that x and y PID coefficients will always be the same.

    Function call: int cmdGetPIDStabilizationData(float *XkP, float *XkD, float *XkI);

    Arguments : 3 pointers to float.

    Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual stabilization PID results will be stored in the locations pointed to by the argument pointers.

12. Configure the altitude stabilization PID

    Sets the PID coefficients of the altitude stabilization PID.

    Function call: int cmdSetPIDAltitudeConfigurationData(float AltkP, float AltkD, float AltkI);

    Arguments : 3 floats for P, D and I coefficients respectively

    Returns: A 4 byte integer error value (0 for success, negative otherwise).

13. Get altitude stabilization PID values

    Returns the PID coefficients of the altitude stabilization PID.

    Function call: int cmdGetPIDAltitudeConfigurationData(float *AltkP, float *AltkD, float *AltkI);

    Arguments : 3 pointers to float

    Returns: A 4 byte integer error value (0 for success, negative otherwise). The actual altitude PID results will be stored in the locations pointed to by the argument pointers.

14. Set target altitude

    This function is used to set the altitude to which Spiri should fly in meters.

    Function call: int cmdSetTargetAltitude(float AltitudeSetADC);

    Arguments : Single 4 byte float value for the altitude

    Returns: Single 4 byte integer. Zero if success or negative if an error occured.

15. Set attitude

    This function is used to set the attitude at which Spiri should fly.

    Function call: int cmdSetAttitude(float XangleTarget, float YangleTarget);

    Arguments : Two 4 byte float values for the x and y components of the attitude in degrees

    Returns: Single 4 byte integer. Zero if success or negative if an error occured.

16. Calibrate accelerometer

    This function is used to set the offsets to the accelerometer when it is level.

    Function call: int cmdCalibrateAccelerometer(float ACCoffsetX, float ACCoffsetY);

    Arguments : Two 4 byte float values for the x and y components of the offsets to the accelerometer in Gs.

    Returns: Single 4 byte integer. Zero if success or negative if an error occured.

17. Get all sensor data

    Returns the data from all the available sensors.

    Function call: int cmdGetAllSensorData(float *ACCXsigned, float *ACCYsigned, float *ACCZsigned, float *GyroDataX, float *GyroDataY, float *GyroDataZ, float *MagnetometerDataX, float *MagnetometerDataY, float *MagnetometerDataZ, float *RangeData, float *PressureValueRaw);

    Arguments : 11 pointers to float

    Returns: A 4 byte integer. Zero if success or negative if an error occured. The actual sensor results will be stored in the locations pointed to by the argument pointers.

18. Get magnetometer data

    Returns magnetometer readings in Gauss with a range of -/+2 Gauss.

    Function call: int cmdGetMagnetometerData(float *MagDataX, float *MagDataY, float *MagDataZ);

Arguments : 3 pointers to float

Returns: A 4 byte integer. Zero if success or negative if an error occured. The actual magnetometer results will be stored in the locations pointed to by the argument pointers.

19. Get GPS data

    Returns readings from the GPS

    Function call: int cmdGetGPS(float *x, float *y, float *z, float *UTC_time, float *HDOP);

    Arguments : 5 pointers to float

    Returns: A 4 byte integer. Zero if success or negative if an error occured. The actual GPS results will be stored in the locations pointed to by the argument pointers.

20. Set mode one flight parameters

    Controls the attitude and average motor speed while Spiri is in mode 1.

    Function call: int cmdMode_one_flight_parameters(float pitch_angle, float roll_angle , float yaw_angle, float rpm);

    Arguments : Four 4-byte floats; pitch angle, roll angle and yaw angle in degrees and average motor speed in RPMs

    Returns: A 4 byte integer. Zero if success or negative if an error occured.

Writing your own C/C++ client to connect to the command processing server over WiFi

A number of things should be noted when writing your own C/C++ client to connect to the Spiri command processing server over WiFi.

1. All the packets sent must be sent in the exact format as presented in the Python section above. Expect responses to the packets as shown in the Python section.
2. All variables in the packet of size larger than 1 byte must be converted from the host endian to network endian, when sending the packets to the command processing server. Use functions like htonl to convert integers to network endian and htonf (in Windows) to convert floating point variables to network endian.
3. All variables/fields greater than 1 byte in the responses sent back by the server must be converted from network endian to host endian.
4. Never send another packet (command) to the server if you have not received the response from the previous sent packet.

5. DO NOT ignore response packets sent by the server and DO NOT ignore the error value sent back in the packet. If the error value is non-zero and negative, it means the command failed. See RHAL.h for the possible error codes that can be sent back.

Conclusion

The set of functions to communicate with Spiri will be continously expanded. We hope you enjoy programmming Spiri.