PprzLink Documentation

Release 0.1.0

Paparazzi Team

Sep 21, 2023

CONTENTS

1 Overview **1**

2 Contents **3**
- 2.1 PprzLink message formats . . . 3
- 2.2 Generating code . . . 7
- 2.3 Python Library . . . 7
- 2.4 OCaml Library . . . 11
- 2.5 Tutorials . . . 11

3 Indices and tables **21**

Python Module Index **23**

Index **25**

CHAPTER

ONE

OVERVIEW

PPRZLINK is a messages toolkit (message definition, code generators, libraries) to be used with **Paparazzi UAV System** (https://paparazziuav.org) and compatible systems. One tool that uses PPRZLINK is the Flying Robot Commander, a web based, RESTful application for controlling multiple aircraft.

CHAPTER

TWO

CONTENTS

2.1 PprzLink message formats

Pprzlink define how data are encoded, as well as how they are encapsulated.

2.1.1 Messages

Messages are defined in the messages.xml file and are grouped into the following message classes:

- telemetry
- datalink
- ground
- alert
- intermcu

The list of Paparazzi Messages can be found in the generated Doxygen documentation. Details of the messages binary format can be found in the related Wiki page.

Message Classes

Telemetry

Telemetry messages are sent from the aircraft to the ground and are defined in the telemetry class of the messages.xml file.

Datalink

Datalink messages are sent from the ground to the aircraft and are defined in the datalink class of the messages.xml file.

Ground

Ground messages are sent to the ground network agents(GCS, server, link, etc...) and are defined in the ground class of the messages.xml file.

Alert

TBD

InterMCU

InterMCU messages are used for communication between airborne MCU when supported and are defined in the intermcu class of the messages.xml file.

2.1.2 Message Format

PPRZ format

The Pprzlink v2.0 standard message is defined as such:

```
PPRZ-message: ABCxxxxxxxDE
    A PPRZ_STX (0x99)
    B LENGTH (A->E)
    C PPRZ_DATA
      0 SOURCE (~sender_ID)
      1 DESTINATION (can be a broadcast ID)
      2 CLASS/COMPONENT
        bits 0-3: 16 class ID
        bits 4-7: 16 component ID
      3 MSG_ID
      4 MSG_PAYLOAD
      . DATA (messages.xml)
    D PPRZ_CHECKSUM_A (sum[B->C])
    E PPRZ_CHECKSUM_B (sum[ck_a])
```

- **PPRZ_STX** (1 bytes) start byte, defined as `0x99`.
- **LENGTH** (1 bytes) full length of the message, from *PPRZ_STX* to *PPRZ_CHECKSUM_B*.
- **SOURCE** (1 bytes) ID of the sender.
- **DESTINATION**: (1 bytes) ID of the receiver.
- **CLASS**: (half a byte) The message class id, coded on the 4 least significant bits. See messages.xml.
- **COMPONENT**: (half a byte) The component id, coded on the 4 most significant bits. This is not yet broadly supported in paparazzi. Default value is 0.
- **MSG_ID**: (1 bytes) The message id. See messages.xml.
- **MSG_PAYLOAD**: (LENGTH-8 bytes) message payload. See messages.xml.
- **PPRZ_CHECKSUM_***: (2 bytes) message checksum.

aircraft ID

- ID `0x00` is reserved for the ground.
- ID `0xFF` is the broadcast ID.

Checksum

PPRZ_CHECKSUM_A is computed as the 1 byte wrapping sum of all bytes from *LENGTH* to *MSG_PAYLOAD*.

PPRZ_CHECKSUM_B is computed as the 1 byte wrapping sum of all values of *PPRZ_CHECKSUM_A*.

In this example code, `data` are the bytes from *LENGTH* to *MSG_PAYLOAD*.

```
def ck(data):
    cka=0
    ckb=0
    for b in data:
        cka += b
        ckb += cka
    return cka,ckb
```

Xbee API format

Pprz data are the same as in the PPRZ format, just the encapsulation differs.

```
XBee-message: ABCDxxxxxxxE
    A XBEE_START (0x7E)
    B LENGTH_MSB (D->D)
    C LENGTH_LSB
    D XBEE_PAYLOAD
      0 XBEE_TX16 (0x01) / XBEE_RX16 (0x81)
      1 FRAME_ID (0)     / SRC_ID_MSB
      2 DEST_ID_MSB      / SRC_ID_LSB
      3 DEST_ID_LSB      / XBEE_RSSI
      4 TX16_OPTIONS (0) / RX16_OPTIONS
      5 PPRZ_DATA
        0 SOURCE (~sender_ID)
        1 DESTINATION (can be a broadcast ID)
        2 CLASS/COMPONENT
            bits 0-3: 16 class ID
            bits 4-7: 16 component ID
        3 MSG_ID
        4 MSG_PAYLOAD
        . DATA (messages.xml)
    E XBEE_CHECKSUM (sum[D->D])

   ID is AC_ID for aircraft, 0x100 for ground station
```

XBee destination ID is 2 bytes long. Use the paparazzi AC_ID for the LSB and 0x00 for the MSB, with 2 exceptions:

- The ground address is `0x0100`
- The broadcast address is `0xFFFF`.

Example

Lets take this "PPRZ" encoded message:

```
99 0C 07 00 01 02 03 00 01 02 1C C4
```

This message can be decomposed as :

STX	LENGHT=12	SOURCE=7	DESTINATION=0	CLASS=1, COMPONENT=0
99	0C	07	00	01

MSG_ID=2	MSG_PAYLOAD	CKA	CKB
02	03 00 01 02	1C	C4

2.1.3 Payload format

The message payload is defined in the messages.xml.

Fields base types are :

- int8
- int16
- int32
- uint8
- uint16
- uint32
- float
- double
- char

Note: values should be encoded as little endian.

Field type can be:

- a base type: `int32`
- an variable size array : `int32[]`
- a fixed size array : `int32[3]`

A variable length array is encoded as its lenght (on 1 byte), then all its values (from low to high indices).

Example

Lets take this payload:

`03 00 01 02`

For this message definition :

```
<message name="ALIVE" id="2">
    <field name="md5sum" type="uint8[]"/>
</message>
```

This message is defined as a single field, a variable length array.

We can then decode its content as the uint8 array [0, 1, 2].

2.2 Generating code

Generating code standard documentation

2.3 Python Library

General description of python lib

2.3.1 Available interfaces

The following interfaces are available :

- *Serial*
- *Ivy*
- *UDP*

2.3.2 Supported protocols

- pprz binary format
- ascii (ivy)

The XBee binary protocol is not supported at the moment.

2.3.3 Running the test programms

The PYTHONPATH environnement variable needs to be set to add the current folder: On bask-like shells: *export PYTHONPATH=<path/to/this/directory>:$PYTHONPATH*

Then you can run test programs with *python -m pprzlink.serial* in the case of the serial interface.

2.3.4 API documentation

Python Messages

Paparazzi message representation

class pprzlink.message.**PprzMessage**(*class_name*, *msg*, *component_id=0*)

base Paparazzi message class

property class_id

Get the class id.

fieldbintypes(*t*)

Get type and length for binary format

property fieldcoefs

Get list of field coefs.

property fieldnames

Get list of field names.

property fieldtypes

Get list of field types.

property fieldvalues

Get list of field values.

get_field(*idx*)

Get field value by index.

ivy_string_to_payload(*data*)

parse Ivy data string to PPRZ values header and message name should have been removed

Basically parts/args in string are separated by space, but char array can also contain a space: `|f,o,o, ,b,a,r|` in old format or `"foo bar"` in new format

property msg_class

Get the message class.

property msg_id

Get the message id.

property name

Get the message name.

to_csv(*payload_only=False*)

return message as CSV string for use with RAW_DATALINK msg_name;field1;field2;

exception pprzlink.message.**PprzMessageError**(*message*, *inner_exception=None*)

Transport interface

Paparazzi transport encoding utilities

class pprzlink.pprz_transport.**PprzParserState**(*value*)

An enumeration.

class pprzlink.pprz_transport.**PprzTransport**(*msg_class='telemetry'*)

parser for binary Paparazzi messages

parse_byte(*c*)

parse new byte, return True when a new full message is available

unpack()

Unpack the last received message

unpack_pprz_msg(*data*)

Unpack a raw PPRZ message

Ivy Interface

The IvyMessagesInterface class allows to send paparazzi messages on the Ivy bus and receive paparazzi messages from Ivy.

See the *tutorial* to learn how to use it.

class pprzlink.ivy.**IvyMessagesInterface**(*agent_name=None*, *start_ivy=True*, *verbose=False*, *ivy_bus=''*)

This class is the interface between the paparazzi messages and the Ivy bus.

bind_raw(*callback*, *regex='(.*)'*)

Bind callback to Ivy messages matching regex (without any extra parsing)

Parameters

- **callback** – function called on new message with agent, message, from as params
- **regex** – regular expression for matching message

static parse_pprz_msg(*ivy_msg*)

Parse an Ivy message into a PprzMessage.

Parameters
ivy_msg – Ivy message string to parse into PprzMessage

Return ac_id, request_id, msg
The parameters to be passed to callback

send(*msg*, *sender_id=None*, *receiver_id=None*, *component_id=None*)

Send a message

Parameters

- **msg** – PprzMessage or simple string
- **sender_id** – Needed if sending a PprzMessage of telemetry msg_class, otherwise message class might be used instead

Returns
Number of clients the message was sent to

Raises
ValueError: if msg was invalid or *sender_id* not provided for telemetry messages

Raises
RuntimeError: if the server is not running

send_raw_datalink(*msg*)

Send a PprzMessage of datalink msg_class embedded in RAW_DATALINK message

Parameters
msg – PprzMessage

Returns
Number of clients the message was sent to

Raises
ValueError: if msg was invalid

Raises
RuntimeError: if the server is not running

send_request(*class_name*, *request_name*, *callback*, ***request_extra_data*)

Send a data request message and passes the result directly to the callback method.

Returns
Number of clients this message was sent to.

Return type
int

Parameters

- **class_name** (*str*) – Message class, the same as PprzMessage.__init__
- **request_name** (*str*) – Request name (without the _REQ suffix)
- **callback** (*Callable[[str, PprzMessage], Any]*) – Callback function that accepts two parameters: 1. aircraft id as int 2. The response message
- **request_extra_data** (*Dict[str, Any]*) – Payload that will be sent with the request if any

Raises
ValueError: if msg was invalid or *sender_id* not provided for telemetry messages

Raises
RuntimeError: if the server is not running

subscribe(*callback*, *regex_or_msg='(.*)'*)

Subscribe to Ivy message matching regex and call callback with ac_id and PprzMessage

Parameters

- **callback** – function called on new message with ac_id and PprzMessage as params
- **regex_or_msg** – regular expression for matching message or a PprzMessage object to subscribe to

subscribe_request_answerer(*callback*, *request_name*)

Subscribe to advanced request messages.

Parameters

- **callback**(*Callable[[int, PprzMessage], PprzMessage]*) – Should return the answer as a PprzMessage

- **request_name** (*str*) – Request message name to listen to (without *_REQ* suffix)

Returns
binding id

Serial Interface

The SerialMessagesInterface class allows to send and receive messages to and from a serial device.

See the *tutorial* to learn how to use it.

class pprzlink.serial.**SerialMessagesInterface**(*callback*, *verbose=False*, *device='/dev/ttyUSB0'*, *baudrate=115200*, *msg_class='telemetry'*, *interface_id=0*)

run()
Thread running function

send(*msg*, *sender_id=None*, *receiver_id=0*, *component_id=0*)
Send a message over a serial link

UDP Interface

The SerialMessagesInterface class allows to send and receive messages to and from an udp socket???

See the *tutorial* to learn how to use it.

class pprzlink.udp.**UdpMessagesInterface**(*callback*, *verbose=False*, *uplink_port=4243*, *downlink_port=4242*, *msg_class='telemetry'*, *interface_id=0*)

run()
Thread running function

send(*msg*, *sender_id*, *address*, *receiver=0*, *component=0*)
Send a message over a UDP link

2.4 OCaml Library

General description of ocaml lib

2.5 Tutorials

2.5.1 Ivy

These tutorials explain how to send and receive messages to and from an Ivy bus.

Send and receive messages from Ivy with python

This tutorial explains how to send and receive messages to and from an Ivy bus. You should be familiar with basic IVY concepts.

The majority of the work will be done through *pprzlink.ivy.IvyMessagesInterface* class. You can have a look at the documentation *here*.

In this tutorial, we will build a simple application which receives *PING* messages and sends back *PONG* messages.

Creating an ivy bus

The first step is to create a *pprzlink.ivy.IvyMessagesInterface* object, which is straightforward:

```
from ivy.std_api import *
import pprzlink.ivy

# Creation of the ivy interface
ivy = pprzlink.ivy.IvyMessagesInterface(
            agent_name="PprzlinkIvyTutorial",   # Ivy agent name
            start_ivy=False,                    # Do not start the ivy bus now
            ivy_bus="127.255.255.255:2010")     # address of the ivy bus
```

Subscribe to messages

You can then issue subscriptions on the ivy bus with the *pprzlink.ivy.IvyMessagesInterface.subscribe()* function. You can subscribe both with a regexp in a string or a `PprzMessage` specifying the type of message you want. Here we are subscribing to all PING messages.

```
    # Subscribe to PING messages and sets recv_callback as the callback function.
    ivy.subscribe(recv_callback,message.PprzMessage("datalink", "PING"))
```

The parameter `message.PprzMessage("datalink", "PING")` creates an empty *PING* message which serves as a prototype.

The function passed as first argument to *pprzlink.ivy.IvyMessagesInterface.subscribe()* (here `recv_callback()`) will be called when a matching message is received. It will be passed two parameters, the id of the sender of the message (*ac_id*) and the message itself (*pprzMsg*).

For now, we only print the message and the id of the sender. Note that since *PING* is a datalink message, the sender *ac_id* will be 0.

```
# Function called when a PING message is received
def recv_callback(ac_id, pprzMsg):
    # Print the message and the sender id
    print ("Received message %s from %s" % (pprzMsg,ac_id))
```

Sending messages

To send a message we use the *pprzlink.ivy.IvyMessagesInterface.send()*. We will use this to send a *PONG* message back when we receive a *PING* message. We identify ourselves as *ac_id* 2.

```
# Function called when a PING message is received
def recv_callback(ac_id, pprzMsg):
    # Print the message and the sender id
    print ("Received message %s from %s" % (pprzMsg,ac_id))
    # Send back a PONG message
    ivy.send(message.PprzMessage("telemetry", "PONG"), sender_id= 2, receiver_id= ac_id)
```

Complete file

The complete file for this tutorial including waiting for keyboard interuption is here.

```
#!/usr/bin/env python

from ivy.std_api import *
import pprzlink.ivy

import pprzlink.messages_xml_map as messages_xml_map
import pprzlink.message as message
import time

# Function called when a PING message is received
def recv_callback(ac_id, pprzMsg):
    # Print the message and the sender id
    print ("Received message %s from %s" % (pprzMsg,ac_id))
    # Send back a PONG message
    ivy.send(message.PprzMessage("telemetry", "PONG"), sender_id= 2, receiver_id= ac_id)

# Creation of the ivy interface
ivy = pprzlink.ivy.IvyMessagesInterface(
            agent_name="PprzlinkIvyTutorial",   # Ivy agent name
            start_ivy=False,                    # Do not start the ivy bus now
            ivy_bus="127.255.255.255:2010")     # address of the ivy bus

try:
    # starts the ivy interface
    ivy.start()

    # Subscribe to PING messages and sets recv_callback as the callback function.
    ivy.subscribe(recv_callback,message.PprzMessage("datalink", "PING"))

    # Wait untill ^C is pressed
    while True:
        time.sleep(5)
except KeyboardInterrupt:
    ivy.shutdown()
```

2.5.2 Serial

These tutorials explain how to send and receive messages to and from a serial device.

Send and receive messages from a serial device with python

This tutorial explains how to send and receive messages to and from a serial device.

We will make a program that periodically sends *PING* messages to a serial device, while monitoring the device to print every paparazzi message received on it. When it receives a *PING* it will send back a *PONG*. This is ment to be tested with a loopback serial device (a FTDI cable with RX and TX linked).

The tutorial code will be written in the `SerialTutorial` class so as to keep state information.

Creating the `pprzlink.serial.SerialMessagesInterface`

First of all we need to create a *pprzlink.serial.SerialMessagesInterface* object to handle sending and receiving data.

```
        self.serial_interface = pprzlink.serial.SerialMessagesInterface(
                                callback = self.process_incoming_message,   # callback␣
↪function
                                device = args.dev,                          # serial device
                                baudrate = args.baud,                       # baudrate
                                interface_id = args.ac_id,                     # id of the␣
↪aircraft
                                )
```

Here we create an interface that will bind to a serial device (`args.dev`) at the specified baud rate (`args.baud`). The id of the local system is set to `args.id` and messages arriving on the serial device for this id will be passed along with the source id to the `self.process_incoming_message()` callback function (see *Receiving messages*).

Note that the construction of *pprzlink.serial.SerialMessagesInterface* can take additional parameters that we will ignore in this tutorial.

Receiving messages

When a message arrives on the serial device specified in the creation of the interface, the id of the destination of the message is checked. If the `interface_id` specified in the interface creation is the same as the destination id of the message, then the callback function will be called.

```
    # Callback function that process incoming messages
    def process_incoming_message (self, source, pprz_message):
```

Here the callback function just prints the message and the id of the source.

Sending messages

To send a message, we just need to call the *pprzlink.serial.SerialMessagesInterface.send()* function. We send the message from our id to ourselves.

```
                # create a ping message
                ping = message.PprzMessage('datalink', 'PING')

                # send a ping message to ourselves
                print("Sending ping")
                self.serial_interface.send(ping, self.ac_id,self.ac_id)
```

Filtering messages on type

In order to filter the messages according to their type, we will use the *pprzlink.message* API. It can be as simple as testing the `name` attribute of the message.

Here we use this so as to answer with a *PONG* message to any *PING* message sent to us. We send it from our id to the id of the source of the *PING* message.

```
        if pprz_message.name == "PING":
            print ("Sending back PONG")
            pong = message.PprzMessage('telemetry', 'PONG')
            self.serial_interface.send(pong, self.ac_id,source)
```

Complete file

The complete file for this tutorial is `here`.

```
#!/usr/bin/env python
import pprzlink.serial
import pprzlink.messages_xml_map as messages_xml_map
import pprzlink.message as message
import time

import argparse

class SerialTutorial:
    """
    Class SerialTutorial that uses pprzlink.serial.SerialMessagesInterface to send
    PING messages on a serial device and monitors incoming messages.
    It respond to PING messages with PONG messages.
    """

    # Construction of the SerialTutorial object
    def __init__(self,args):
        self.serial_interface = pprzlink.serial.SerialMessagesInterface(
                            callback = self.process_incoming_message,    # callback␣
↪function
                            device = args.dev,                           # serial device
                            baudrate = args.baud,                        # baudrate
```

(continues on next page)

(continued from previous page)

```
                            interface_id = args.ac_id,                    # id of the␣
↪aircraft
                            )
        self.ac_id = args.ac_id
        self.baudrate = args.baud

    # Main loop of the tutorial
    def run(self):
        print("Starting serial interface on %s at %i baud" % (args.dev, self.baudrate))

        try:
            self.serial_interface.start()

            # give the thread some time to properly start
            time.sleep(0.1)


            while self.serial_interface.isAlive():
                self.serial_interface.join(1)

                # create a ping message
                ping = message.PprzMessage('datalink', 'PING')

                # send a ping message to ourselves
                print("Sending ping")
                self.serial_interface.send(ping, self.ac_id,self.ac_id)

        except (KeyboardInterrupt, SystemExit):
            print('Shutting down...')
            self.serial_interface.stop()
            exit()

    # Callback function that process incoming messages
    def process_incoming_message (self, source, pprz_message):
        print("Received message from %i: %s" % (source, pprz_message))
        if pprz_message.name == "PING":
            print ("Sending back PONG")
            pong = message.PprzMessage('telemetry', 'PONG')
            self.serial_interface.send(pong, self.ac_id,source)


if __name__ == '__main__':

    parser = argparse.ArgumentParser()
    parser.add_argument("-d", "--device", help="device name", dest='dev', default='/dev/
↪ttyUSB0')
    parser.add_argument("-b", "--baudrate", help="baudrate", dest='baud', default=115200,
↪ type=int)
    parser.add_argument("-id", "--ac_id", help="aircraft id (receiver)", dest='ac_id',␣
↪default=42, type=int)
    args = parser.parse_args()
```

(continues on next page)

(continued from previous page)

```
    serialTutorial = SerialTutorial(args)

    serialTutorial.run()
```

2.5.3 UDP

These tutorials explain how to send and receive messages to and from an udp socket???

Send and receive messages from an udp socket with python

This tutorial explains how to send and receive messages to and from an udp socket.

We will make a program that wait for messages and print them. If the message received is a *PING* message it answers with a *PONG* message. Furthermore it will have the option to send *PING* messages every two seconds.

The tutorial code will be written in the `UDPTutorial` class so as to keep state information.

Creating the `pprzlink.udp.UdpMessagesInterface`

First of all we need to create a *pprzlink.udp.UdpMessagesInterface* object to handle sending and receiving data.

```
import pprzlink.messages_xml_map as messages_xml_map
            self.udp = pprzlink.udp.UdpMessagesInterface(
                            self.process_msg,            # Callback function
                            uplink_port = pong_port,     # Port we send messages to
                            downlink_port = ping_port,   # Port used to receive messages
                            interface_id = my_receiver_id # Numerical id of the␣
↪interface (ac_id)
```

Here, we create an interface that will receive message on port `ping_port` and send its messages toward port `pong_port`. It will call `self.proccess_msg()` when a message is received (see *Receiving messages*) and filter messages that are sent to id `my_receiver_id`.

Receiving messages

When a message arrives on the udp port specified as `downlink_port` in the creation of the interface, the id of the destination of the message is checked. If the `interface_id` specified in the interface creation is the same as the destination id of the message or if it was specified as `None`, then the callback function will be called.

```
    # Function used as callback when messages are received
    def proccess_msg(self,sender,address,msg,length,receiver_id=None, component_id=None):
            print("Received message from %i %s [%d Bytes]: %s" % (sender, address,␣
↪length, msg))
```

Here the callback function just prints the message.

Sending messages

To send a message, we just need to call the *pprzlink.udp.UdpMessagesInterface.send()* function.

```
                    # Create a PING message
                    ping = message.PprzMessage('datalink', 'PING')

                    # Send PING message
                    self.udp.send(
                        ping,           # The message to send
                        my_sender_id,   # Our numerical id
                        remote_address, # The IP address of the destination
                        my_receiver_id  # The id of the destination
                        )
```

Filtering messages on type

In order to filter the messages according to their type, we will use the *pprzlink.message* API. It can be as simple as testing the `name` attribute of the message.

```
        if msg.name=="PING":
            print("Received PING from %i %s [%d Bytes]" % (sender, address, length))
```

Complete file

The complete file for this tutorial including waiting for keyboard interuption and selecting who sends the *PING* is `here`.

It can be tested by running it twice, one time without the *-s* switch and one with it.

```
#!/usr/bin/env python

import threading
import time

import pprzlink.udp
import pprzlink.messages_xml_map as messages_xml_map
import pprzlink.message as message

# Some constants of the program
ping_port = 2010 # The port to which the PING are sent
pong_port = 2011 # The port to which the PONG are sent
remote_address = "127.0.0.1"
my_sender_id = 1
my_receiver_id = 2


class UDPTutorial:
    """
    Class UDPTutorial that uses udp.UdpMessagesInterface to listen to incoming messages.
    If a PING message arrives, it will answer with a PONG message back to the sender.
```

(continues on next page)

(continued from previous page)

```
    It can also send PING every 2 seconds if constructed with the parameter ping_sender
    to True.
    """

    # Construction of the UDPTutorial object
    def __init__(self,ping_sender = False):
        self.ping_sender = ping_sender
        if ping_sender:
            # If we should send the pings, use ping_port as uplink_port (the port we␣
↪send to)
            # and pong_port as the downlink (the port we listen to)
            self.udp = pprzlink.udp.UdpMessagesInterface(
                            self.process_msg,          # Callback function
                            uplink_port = ping_port,    # Port we send messages to
                            downlink_port = pong_port,  # Port used to receive messages
                            interface_id = my_sender_id # Numerical id of the interface␣
↪(ac_id)
                            )
        else:
            # If we should not send the pings, use pong_port as uplink_port (the port we␣
↪send to)
            # and ping_port as the downlink (the port we listen to)
            self.udp = pprzlink.udp.UdpMessagesInterface(
                            self.process_msg,            # Callback function
                            uplink_port = pong_port,      # Port we send messages to
                            downlink_port = ping_port,    # Port used to receive messages
                            interface_id = my_receiver_id # Numerical id of the␣
↪interface (ac_id)
                            )

    # Function used as callback when messages are received
    def process_msg(self,sender,address,msg,length,receiver_id=None, component_id=None):
        # If it is a PING send a PONG, else print message information
        if msg.name=="PING":
            print("Received PING from %i %s [%d Bytes]" % (sender, address, length))
            pong = message.PprzMessage('telemetry', 'PONG')
            print ("Sending back %s to %s:%d (%d)" % (pong,address[0],address[1],sender))
            self.udp.send(pong, receiver_id, address[0], receiver = sender)
        else:
            print("Received message from %i %s [%d Bytes]: %s" % (sender, address,␣
↪length, msg))

    # Activity function of this object
    def run(self):
        try:
            # Start the UDP interface
            self.udp.start()

            if self.ping_sender:
                    # Create a PING message
                    ping = message.PprzMessage('datalink', 'PING')
```

(continues on next page)

(continued from previous page)

```
            # Wait for a ^C
            while True:
                if self.ping_sender:
                    # Send PING message
                    print ("Sending %s to %s:%d (%d)" % (ping,remote_address,ping_port,
↪my_receiver_id))
                    self.udp.send(
                        ping,           # The message to send
                        my_sender_id,   # Our numerical id
                        remote_address, # The IP address of the destination
                        my_receiver_id  # The id of the destination
                        )
                time.sleep(2)

        except KeyboardInterrupt:
            self.udp.stop()

if __name__ == '__main__':
    from argparse import ArgumentParser

    # Parse arguments looking for the -s switch telling us we should send the PING
    parser = ArgumentParser(description="UDP Tutorial for pprzlink")
    parser.add_argument("-s","--send_ping",dest="send",default=False, action='store_true
↪', help="Send the PING messages")
    args = parser.parse_args()

    # Run the UDPTutorial
    UDPTutorial(args.send).run()
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

p

pprzlink.message, 8
pprzlink.pprz_transport, 9

INDEX

B

bind_raw() (*pprzlink.ivy.IvyMessagesInterface method*), 9

C

class_id (*pprzlink.message.PprzMessage property*), 8

F

fieldbintypes() (*pprzlink.message.PprzMessage method*), 8
fieldcoefs (*pprzlink.message.PprzMessage property*), 8
fieldnames (*pprzlink.message.PprzMessage property*), 8
fieldtypes (*pprzlink.message.PprzMessage property*), 8
fieldvalues (*pprzlink.message.PprzMessage property*), 8

G

get_field() (*pprzlink.message.PprzMessage method*), 8

I

ivy_string_to_payload() (*pprzlink.message.PprzMessage method*), 8
IvyMessagesInterface (*class in pprzlink.ivy*), 9

M

module
    pprzlink.message, 8
    pprzlink.pprz_transport, 9
msg_class (*pprzlink.message.PprzMessage property*), 8
msg_id (*pprzlink.message.PprzMessage property*), 8

N

name (*pprzlink.message.PprzMessage property*), 8

P

parse_byte() (*pprzlink.pprz_transport.PprzTransport method*), 9
parse_pprz_msg() (*pprzlink.ivy.IvyMessagesInterface static method*), 9
pprzlink.message
    module, 8
pprzlink.pprz_transport
    module, 9
PprzMessage (*class in pprzlink.message*), 8
PprzMessageError, 8
PprzParserState (*class in pprzlink.pprz_transport*), 9
PprzTransport (*class in pprzlink.pprz_transport*), 9

R

run() (*pprzlink.serial.SerialMessagesInterface method*), 11
run() (*pprzlink.udp.UdpMessagesInterface method*), 11

S

send() (*pprzlink.ivy.IvyMessagesInterface method*), 9
send() (*pprzlink.serial.SerialMessagesInterface method*), 11
send() (*pprzlink.udp.UdpMessagesInterface method*), 11
send_raw_datalink() (*pprzlink.ivy.IvyMessagesInterface method*), 10
send_request() (*pprzlink.ivy.IvyMessagesInterface method*), 10
SerialMessagesInterface (*class in pprzlink.serial*), 11
subscribe() (*pprzlink.ivy.IvyMessagesInterface method*), 10
subscribe_request_answerer() (*pprzlink.ivy.IvyMessagesInterface method*), 10

T

to_csv() (*pprzlink.message.PprzMessage method*), 8

U

UdpMessagesInterface (*class in pprzlink.udp*), 11
unpack() (*pprzlink.pprz_transport.PprzTransport method*), 9

unpack_pprz_msg() (*pprzlink.pprz_transport.PprzTransport method*), 9