INAF
ISTITUTO NAZIONALE DI ASTROFISICA
NATIONAL INSTITUTE FOR ASTROPHYSICS

ISTITUTO DI ASTROFISICA SPAZIALE E FISICA COSMICA

PacketLib 1.3.2
Programmer's Guide

DOCUMENT: IASF-Bologna Report 410/05 **PAGE:** i of ii, 36

ISSUE No.: Issue 02
DATE: February 2005
PREPARED BY: A. BULGARELLI, F. GIANOTTI, M. TRIFOGLIO

iASF Sezione di Bologna

Area Ricerca di Bologna - via Piero Gobetti, 101 - 40129 Bologna - ITALY
Tel. (+39) 051.6398688 - Fax (+39) 051.6398724
e-mail: iasf@bo.iasf.cnr.it - www.bo.iasf.cnr.it
Cod. Fisc. 97220210583 - Part. Iva 06895721006

CHANGE RECORD

Issue	Date	Page	Description of Changes	Release
1	18/09/02		First issue related to PacketLib 1.2.2	1
2	24/02/05		Added Layout 4 and updated for PacketLib 1.3.2	1

PacketLib 1.3.2 Programmer's Guide

TABLE OF CONTENTS

1INTRODUCTION....3
1.1Scope and Purpose of the Document....3
Definitions, Acronyms, Abbreviations....3
1.1.1Acronyms....3
2REFERENCE DOCUMENTS....4
3ARCHITECTURAL OVERVIEW....5
3.1The main layers....5
3.2Package....5
3.2.1Basic type....5
4DESIGN OVERVIEW....8
4.1I/O Abstraction layer....8
4.1.1ByteStream class....8
4.1.2Device hierarchy....8
4.1.3Input hierachy....11
4.1.4Output Hierarchy....12
4.2Main Layer....12
4.2.1PacketStream....12
4.2.2Packet....14
5OVERVIEW: HOW THE LIBRARY WORKS....20
6DEVICE EXAMPLE....21
6.1File....21
6.2SHM....22
6.2.1SHM Server....22
6.2.2SHM Client....23
6.3MSGQ....24
6.3.1MSGQ Server....24
6.4MSGQ Client....25
7OUTPUT EXAMPLES....27
7.1General description....27
7.2Example 1....27
8INPUT EXAMPLES....29
8.1General description....29
8.2Example 1....29
8.3Reading from a socket....30
8.4Reading a ByteStream....30
8.5Reading a Packet....31
8.6InputPacketStreamFile example....31
9EXAMPLE ABOUT LAYOUT 4....32
10ANNEX A – CONSIDERATION ABOUT BYTESTREAM (ITALIAN)....35

List OF Figures

FIGURA 3-3.1: PACKETLIB LAYERS....6
FIGURE 5.1: LAYOUT 1 PACKET STRUCTURE....17

FIGURE 5.2: LAYOUT 2 PACKET STRUCTURE....18
FIGURE 5.3: LAYOUT 3 PACKET STRUCTURE....19
FIGURE 5.4: VARIABLE BLOCK STRUCTURE....19
FIGURE 5.5: PACKET WITH RBLOCKS....21
FIGURE 5.6: RBLOCK....21
FIGURE 5.7: TELEMETRY INSTANCE POINT OF VIEW....22
FIGURE 5.8....23
FIGURE 5.9: RBLOCK OF TYPE E....23

1 INTRODUCTION

1.1 Scope and Purpose of the Document

This document describes the PacketLib library for programmers who write applications able to handle satellite telemetry having the Source Packets structure compliant to the ESA Telemetry and Telecommand standard as defined by [3A] and [3B].

PacketLib is a C++ software library, running on Unix platform, which allows the applications to easily decode the Source packets, down to the level of the single logical structure contained in the data field.

PacketLib is aimed at providing a reusable software library for satellite telemetry production and processing and a rapid development for Test Equipment (TE), EGSE and Ground Segment applications.

The diagrams and the terms presented in this document are conformed with the UML-OMG 1.4 standard [2]. The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components. The UML represents the culmination of best practices in practical object-oriented modeling.

The architecture is described by means of the *implementation* diagrams. This diagrams shows aspects of physical implementation, including the structure of components and the run-time deployment system. These diagrams come in two forms:

- *component diagrams* show the structure of components, including the classifiers that specify them and the artifacts that implement them;
- *deployment diagrams* show the structure of the nodes on which the components are deployed.

The *package diagram* gives a logical overview of the software architecture (a package is a grouping of element as component, code, etc.).

Further details on the PacketLib are provided in [1].

Definitions, Acronyms, Abbreviations

The following is a list of definitions used throughout this document.

1.1.1 ACRONYMS

EGSE	Electrical Ground Support Equipment
GSE	Ground Support Equipment
Instrument SC	Instrument Science Console
IP	Integrated Payload
P/L	Payload
PDHU	Payload Data handling Unit
TBC	To Be Confirmed
TBD	To Be Defined
TC	Telecommand
TE	Test Equipment
TM	Telemetry

2 REFERENCE DOCUMENTS

[1] A. Bulgarelli, F. Gianotti, M. Trifoglio, "PacketLib Reference Manual*", IASF-Bologna Report 411/05, Issue 2, Febraury 2005*

[2] "OMG Unified Modeling Language Specification", Version 1.4, September 2001.

[3A] "Packet Telecommand Standard", ESA-PSS-04-107, Issue 2

[3B] "Packet Telemetry Standard", ESA-PSS-04-106, Issue 1

[4] A. Bulgarelli, F. Gianotti, M. Trifoglio, "PacketLib Interface Control Document", IASF-Bologna Report 386/04, Issue 2, Febraury 2005

3 ARCHITECTURAL OVERVIEW

3.1 The main layers

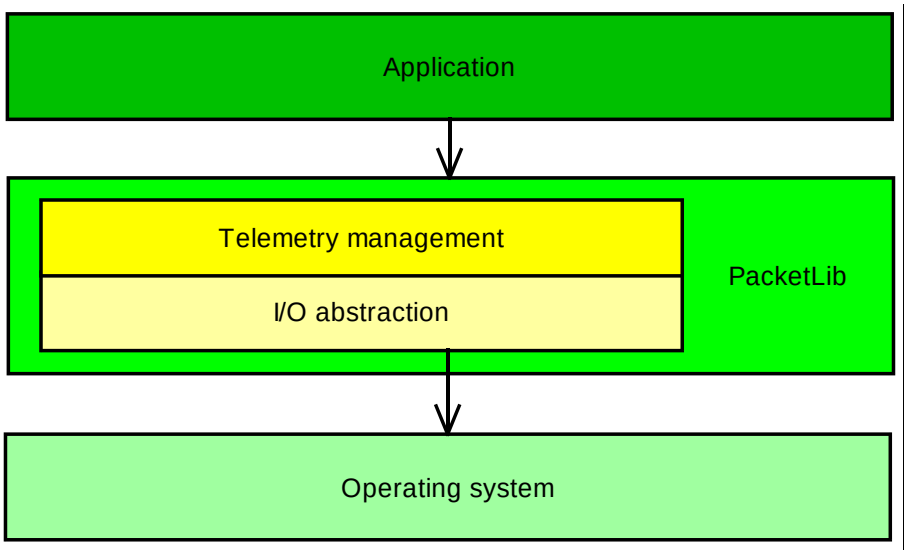


Figura 3.1: PacketLib layers

PacketLib is structured into two main layers: the Telemetry Management layer, which interfaces the application, and the I/O Abstraction layer, which interfaces the Operating System. The former allows the application to address the various elements forming the packet data stream, without having to deal with the specific structure. The latter abstracts from the specific input and output mechanisms (e.g. sockets, files, and shared memories).

This approach allows for a more rapid development of the user applications without having to deal with the kind of input and output sources, that could be changed at run time. Indeed the library could be used either to process or to generate a telemetry stream.

3.2 Package

PacketLib library should be divided into the following packages:

Main: contains the classes that manages the telemetry

IO: classes for I/O mechanism. See section 4.1 and chapter 3 and 8.

PacketLibException: class of exceptions

Utility: class with static methods of general utility. See [1].

Basic type: package with the basic type of the library.

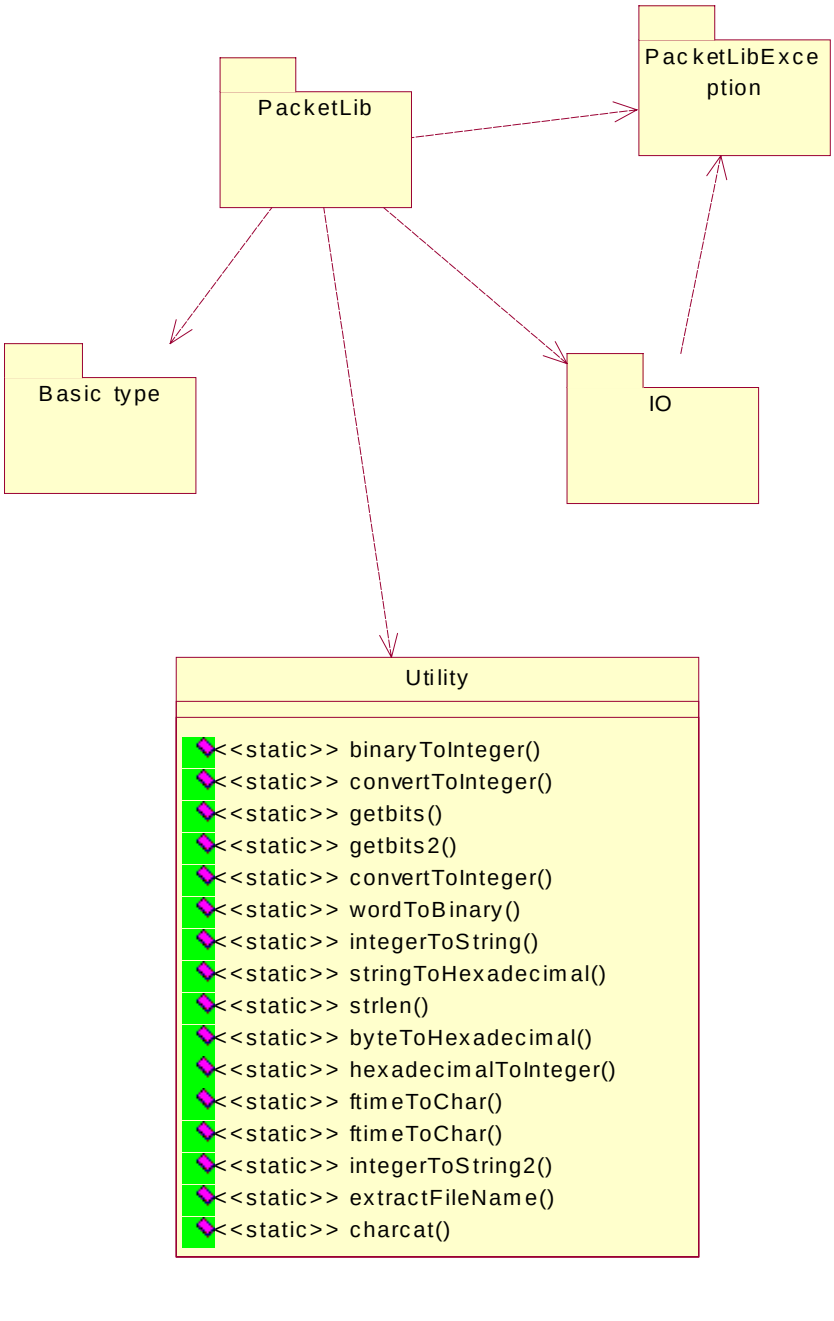
3.2.1 BASIC TYPE

The PacketLibDefinition.h file includes the definition of the following basic types:

```
#include <string>
#include <string.h>
#include <errno.h>
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#define TRUE 1
#define FALSE 0
#define EOI -1
#define DEBUGMODE 0
#define NULL 0

//#define PRINTDEBUG(strprint) if(DEBUGMODE) cout << strprint << endl;
#define PRINTDEBUG(strprint) if(DEBUGMODE) printf("%s\n", strprint);

typedef unsigned char byte;      //1 byte

typedef unsigned short word;     //2 byte

typedef unsigned long dword;     //4 byte

typedef bool boolean;

using namespace std;

#define BIGENDIANITY 0
```

DEBUGMODE=1 enable us to switch on all the functionalities useful during the development of the library, for example the use of the PRINTDEBUG macro.



Figur2 3.2: PacketLib architecture

4 DESIGN OVERVIEW

4.1 I/O Abstraction layer

This section presents the general architecture of the I/O system embedded into PacketLib. This layer allows to develope the applications independently of the device which generates/receives the telemetry. This means that it is possible to choose the specific I/O device at run time.

The main components are:

- Device hierarchy
- Input hierarchy
- Output hierarchy

In addition, the ByteStream class represents a stream of byte.

4.1.1 BYTESTREAM CLASS

This class is one of the core classes of the library. This class represents both the byte stream to be read and the byte stream to be written. This class manages also the Endianity.

Figure 4.1: ByteStream class

4.1.2 DEVICE HIERARCHY

This hierarchy includes all the classes representing an I/O system, with their characteristics. It is noted that only few elements are common through the class interfaces. This is because the possible devices (file, socket, shared memory and so on) may be very different.

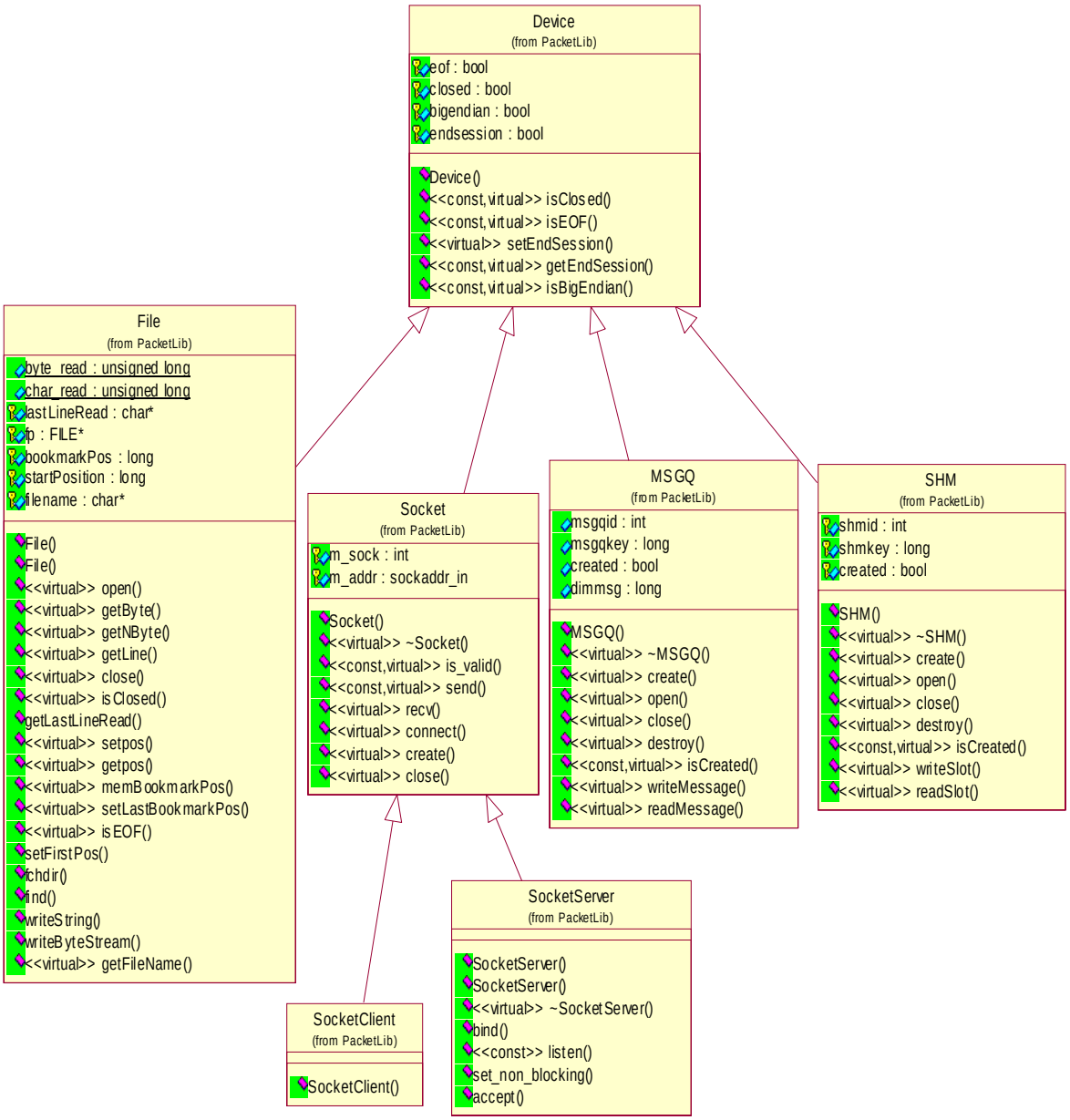


Figure 4.2: Device hierarchy

4.1.2.1 FILE

This class represents a generic file with all the required methods (open, close and create). Some methods are present for reading and writing the strings and ByteStream. See Chapter 6 for some example.

4.1.2.2 DISCOSSHM

This device is used to connect the DISCoS shared memory (usable only with –D WITHDISCOS compile option).

4.1.2.3 SOCKET

This device represents a generic socket (client o server).

4.1.2.4 SHM

With this class it is possible to create, connect, read and write from/to a shared memory. It is necessary to create a *struct* that represents the structure to be read or to be written. It manages a structure organized in slots. This means that it is possible to manage a list of data of the same format. The dimension of each slot is determined by the definition of the *struct*.

In order to write, it is necessary to assign to this *struct* the value to be written, and to use the *writeSlot()* method. With *readSlot()* it is possible to read a slot.

4.1.3 INPUT HIERACHY

Within this hierarchy are present all the classes that represent an I/O system for the PacketLib. With these classes it is possible to read a ByteStream or a Packet from Input. See chapter 8 for some examples.

The main methods are:

- *open(char** parameters)*: open the input with the correct parameters (file name for the InputFile, IP address and port for InputSocket)
- *close()*: close the input
- *ByteStream* readByteStream(int dim)*
- *char* readString()*: read a string

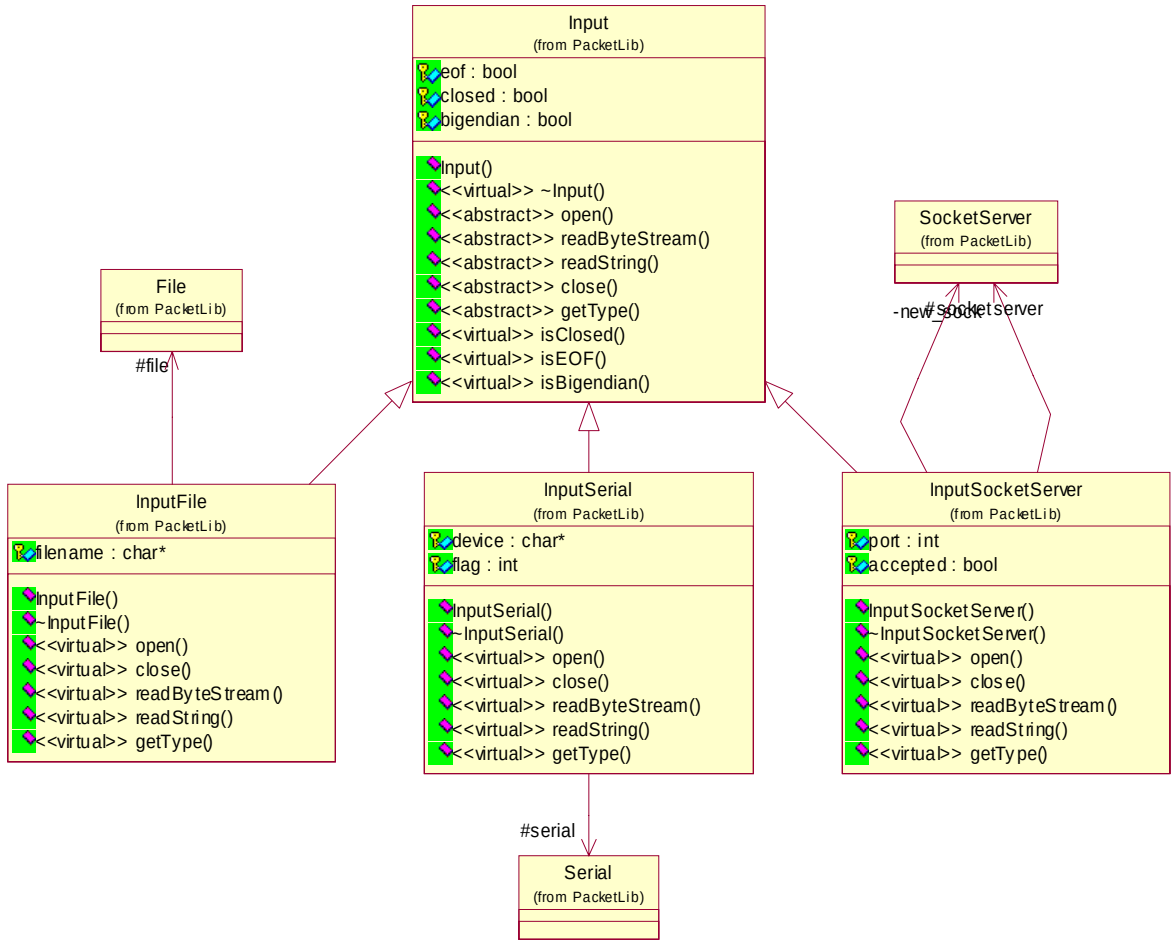


Figure 4.3: Input hierarchy

4.1.4 OUTPUT HIERARCHY

Output abstraction layer. See Chapter 6 for some example.
The main methods are:

- *open(char** parameters)*: open the output (see 4.1.3)
- *close()*: close the output
- *writeByteStream(ByteStream* b)*: write a ByteStream to the output
- *writeString(char* c)*: write a string

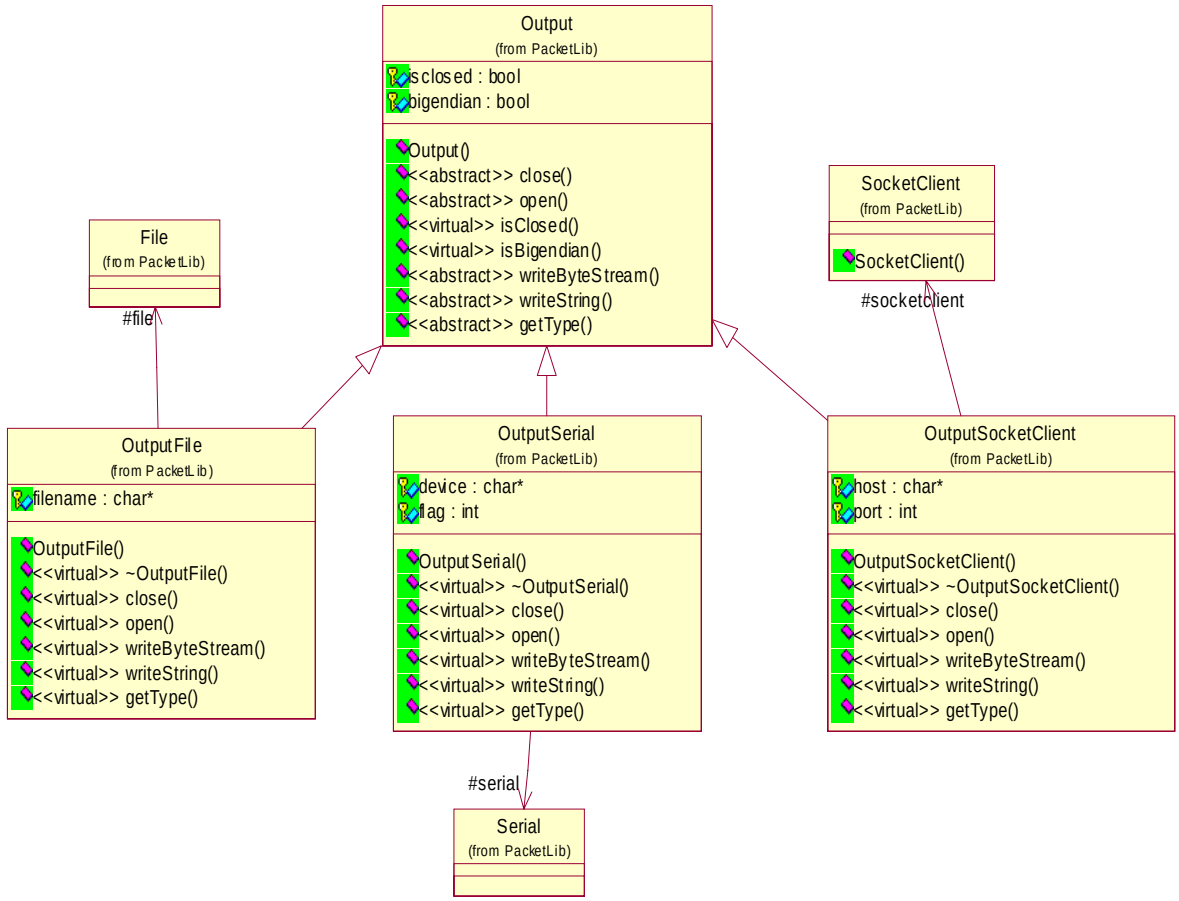


Figure 4.4: Output hierarchy

4.2 Main Layer

4.2.1 PACKETSTREAM

PacketStream is the hierarchy that represents the telemetry stream. Two main classes are present:
InputPacketStream: it represents a packet stream as input. This is useful for reading a stream
OutputPacketStream, it represents a packet stream as output. This is useful for writing a stream.
Before using a PacketStream, it is necessary to create the structure of the stream in memory, by means of the `createStreamStructure(*.stream file name)` method.

The list of packets is contained into the *.stream file. In the following example (more details in the Interface Control Document) 3 types of packets are created into the memory.

```
[Configuration]
--prefix
true
--stream format bigendian
true
--dimension of prefix
2
[Header Format]
headerESATM.header
[Packet Format]
BURST.packet
TC_CAL_start.packet
TC_CAL_stop.packet
```

Before to use a packet, it is necessary to get a reference to it:

```
Packet* p = ops.getPacketType(2);
```

The index that is passed to the method depends on the content of the .stream. The first packet (in this case the BURST.packet) has index 1. The index 0 is reserved to an object of the class PacketNotRecognized.

The input and output objects of the Input and Output hierarchy should be connected to OutputPacketStream and to the InputPacketStream via setOutput() or setInput() methods (see example).

The method InputPacketStream::readPacket() read a telemetry packet from the input.
The method OutputPacketStream::writePacket() write a telemetry packet to the output.

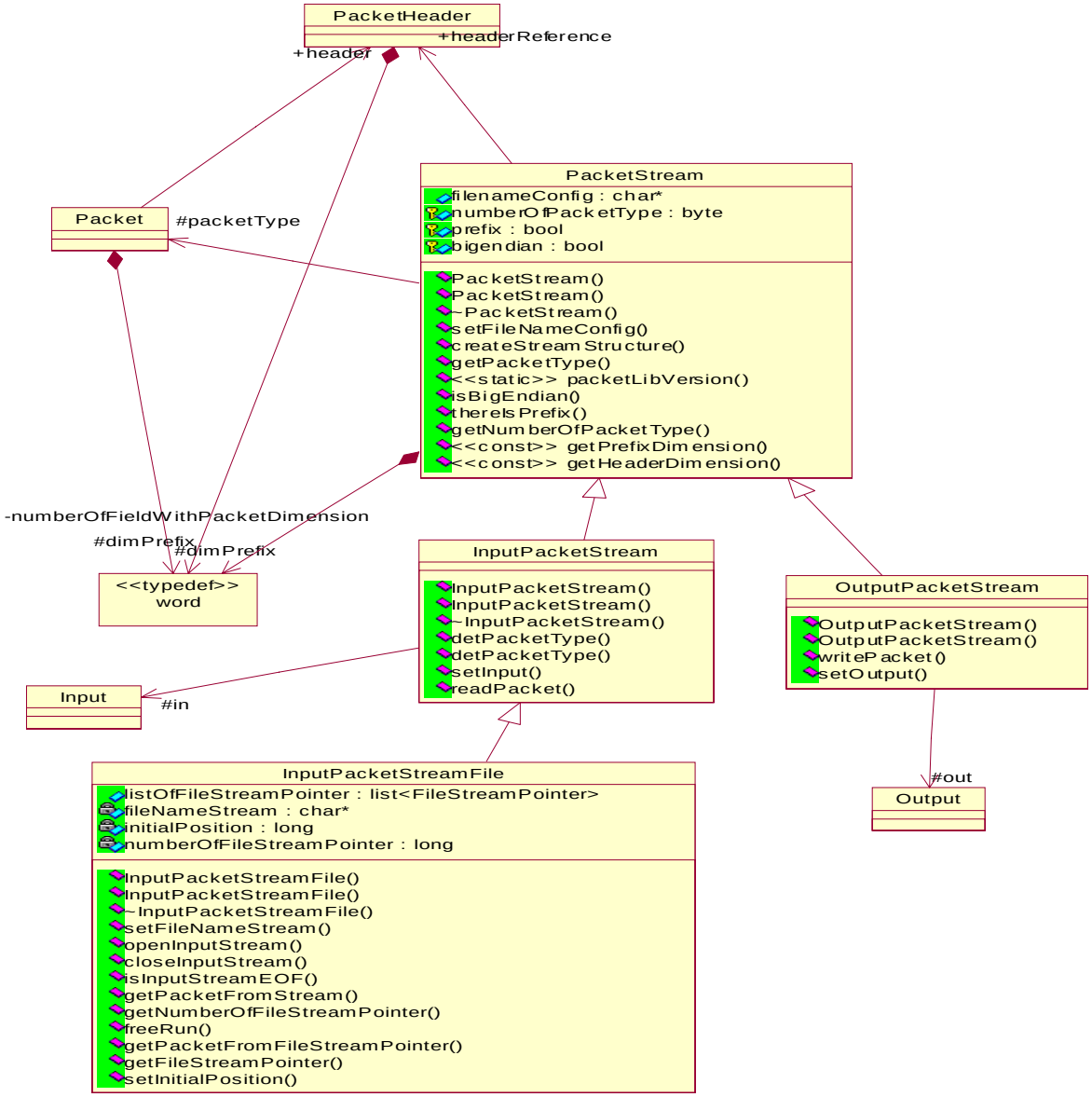


Figura 4.5: PacketStream hierarchy

4.2.2 PACKET

Packet class is the core of the library because it represents a TM or TC packet. The structure of a packet is described by some configuration files (with .header, .packet and .rblock extension, see Interface Control Document) provided by the Programmer.

4.2.2.1 COMMON STRUCTURE: PARTOFPACKET

A telemetry packet should be basically divided into sections. Each section is basically compound by a list of fields.

This structure is represented by the PartOfPacket class that contains all the methods for the basic elaboration of a list of fields or single fields.

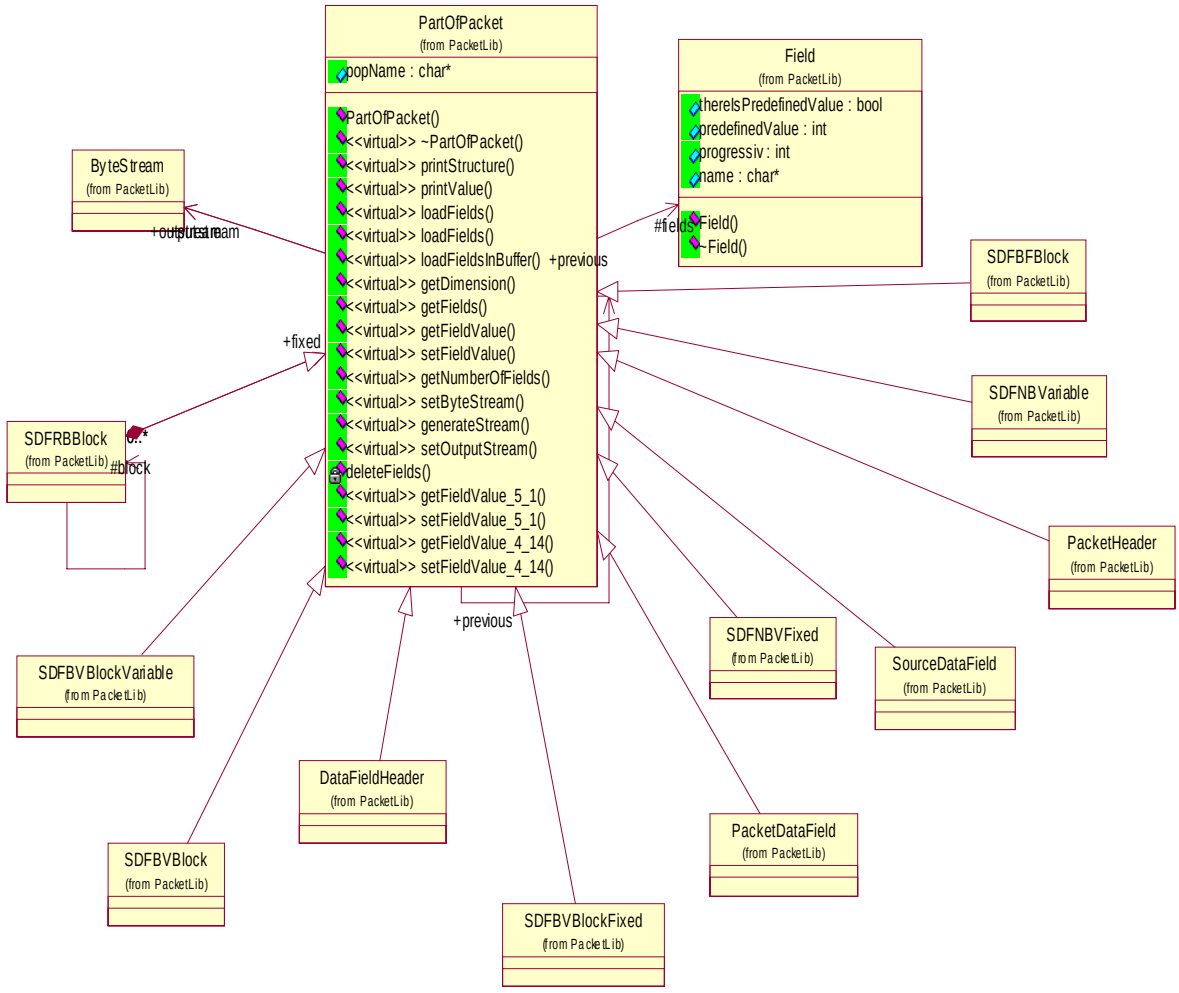


Figura 4.6: PartOfPacket and the derived class

In the Figura 4.6 are shown all the methods and the sections foreseen for a telemetry packet. For more details about these section see 4.2.2.5.

4.2.2.2 THE CLASS PACKET

The class Packet represents a telemetry packet.

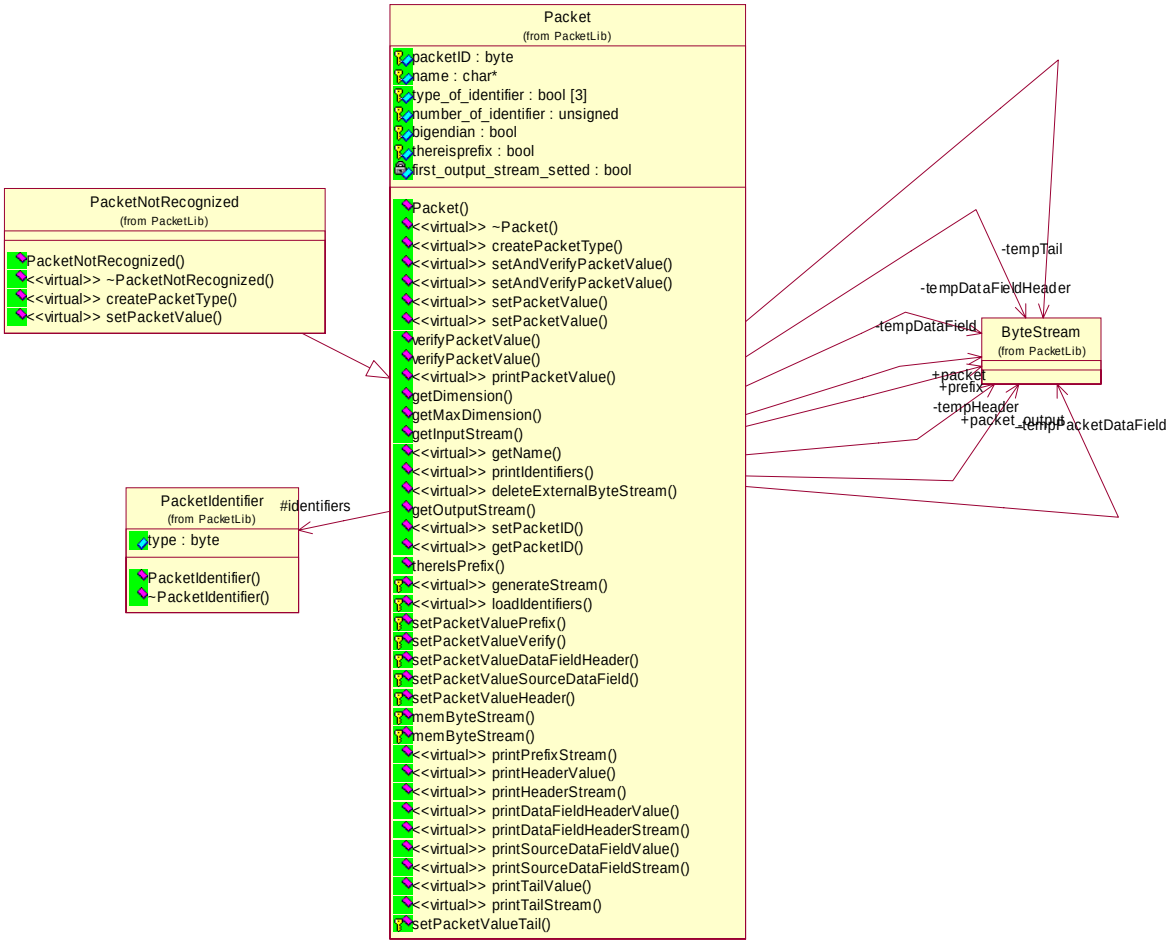


Figure 4.7: The Packet class hierarchy

The method print() are useful to print to stdout the content of a section of the packet.

4.2.2.3 NAVIGATION METHOD OF THE PACKET AND HOW TO WORK WITH FIELD VALUES

It is possible to set and get the value of the fields by means of the setFieldValue() and getFieldValue() methods applied to the various sections of the packet.
For example, if p is a pointer to a packet obtained from a PacketStream,

p->header: enables us to access to the header of the packet
p->dataField->dataFieldHeader: enables us to access to the data field header of the packet
p->dataField->sourceDataField: enables us to access to the source data field of the packet
p->dataField->tail: enables us to access to the tail of the packet.

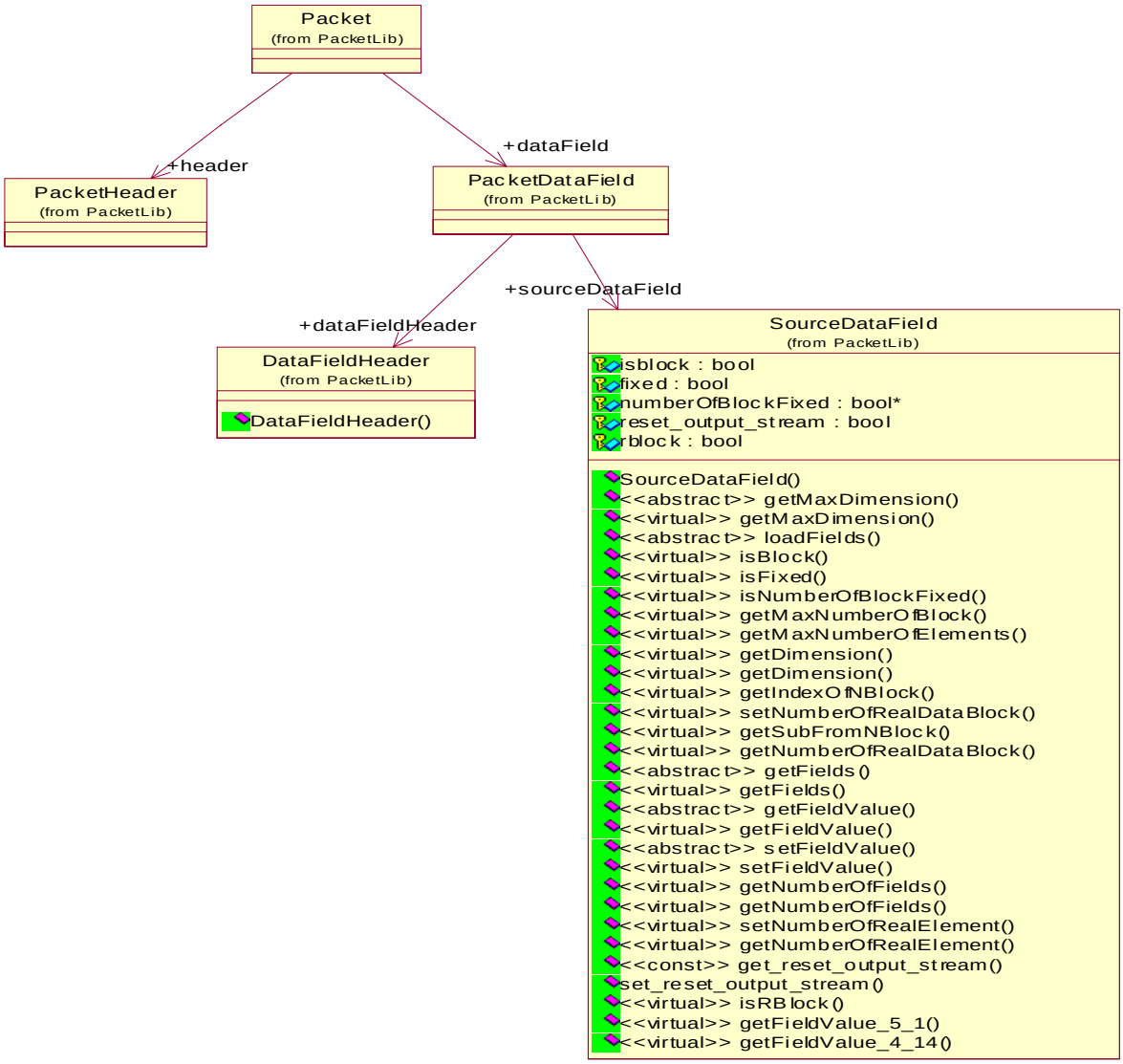


Figure 4.8: Packet and main sections

Some example follows. If we want to set the value 257 to the field 3 of the header:

```
p->header->setFieldValue(3, 257);
```

It is noted that if a predefined value is specified in the .stream, the setFieldValue has no effect, and the value contained in the .stream is used.

Set the value 1257 to the field 2 of the data field header and the value 127 to the field 32 of the source data field:

```
p->dataField->dataFieldHeader->setFieldValue(2, 1257);
p->dataField->sourceDataField->setFieldValue(32, 127);
```

Get the value of the field 10 of the data field header:

```
word w = p->dataField->dataFieldHeader->getFieldValue(2);
```

Some specialized methods exists for some particular PTC, PFC:
virtual float getFieldValue_5_1 (word index)
virtual void setFieldValue_5_1 (word index, float value)
virtual long getFieldValue_4_14 (word index)
virtual void setFieldValue_4_14 (word index, long value)
See Reference Guide for more details.

In addition, if a source data field is organized into blocks, it is possible to have the following methods:
virtual word getFieldValue (word block, word index)
virtual void setFieldValue (word block, word index, word value)
These methods works for Layout 2, 3.

To work with the fixed part of the source data field of the Layout 4 it is enough to use the basic version of the getFieldValue() and setFieldValue() methods.

If we want to access a rblock, it is necessary to use the following method:

```
virtual SDFRBBlock *  getBlock (word nblock, word rBlockIndex)
```

We obtain the block of number nblock of the group of blocks of the rblock with the index rBlockIndex (see the Interface Control Document). Now, we can use the method setFieldValue() and getFieldValue() of the SDFRBBlock class to access to the fixed part of this rblock, and use the getBlock() to access to the variable part. See the related example in this document.

4.2.2.4 GET THE FIELD

It is possible to get the entire field using the method getField() applied to the various part of packet shown in the section 4.2.2.3.

4.2.2.5 PACKET STRUCTURE

In the Figure 4.9 is shown the overall view of the packet. The class orange is the class for the telemetry layout 1, the red classes are used for the telemetry Layout 2, the yellow classes are used for the telemetry Layout 3 and the green classes are used for the telemetry Layout 4 (see Interface Control Document).

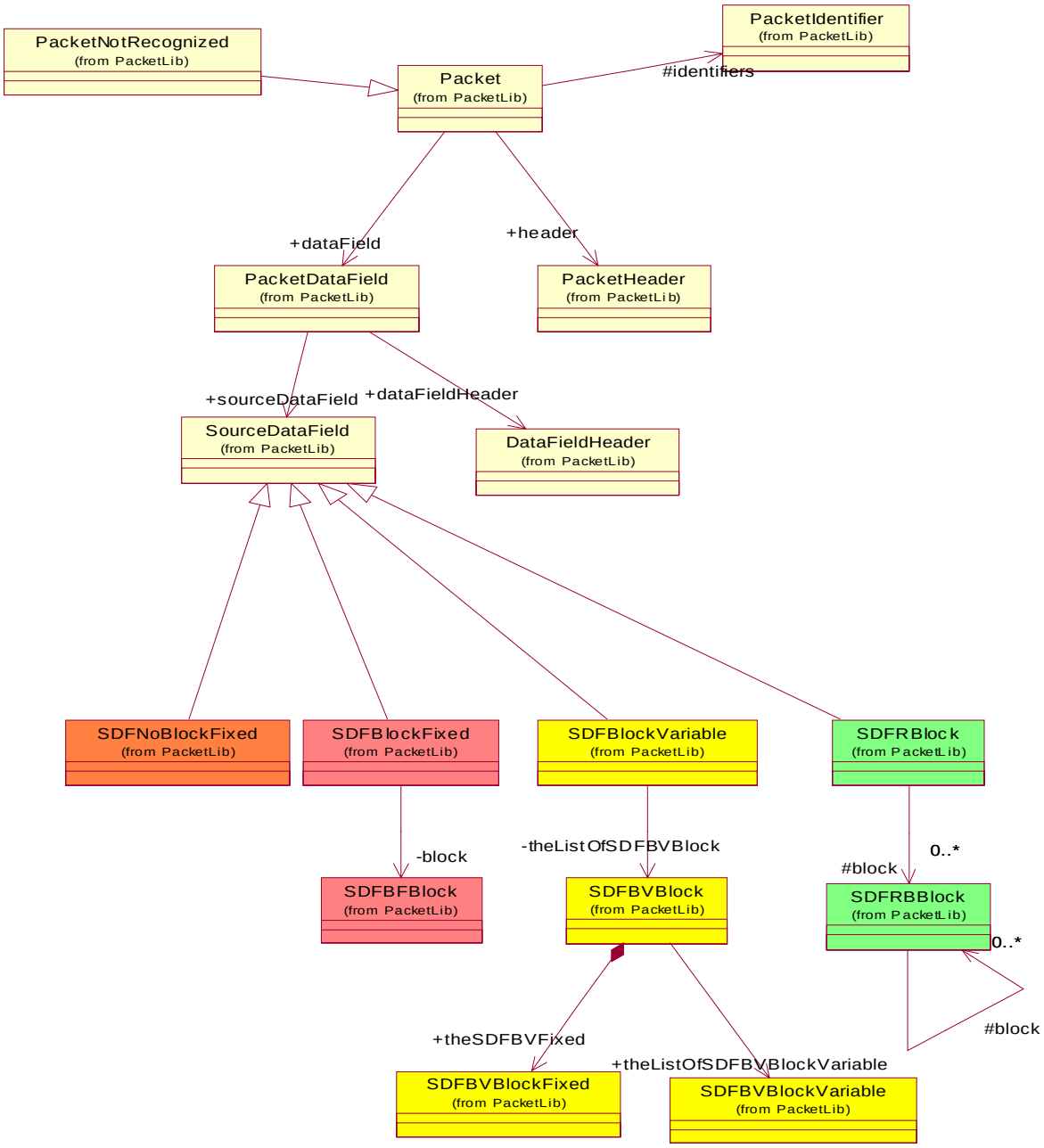


Figure 4.9: Overall view of the packet

5 OVERVIEW: HOW THE LIBRARY WORKS

The real working of library can be easily understand with a little coding example that process the input.

```
1.  InputPacketStream ps;

2.  char* parameters; //set the parameters of input

3.  Input *in = new InputSocket(parameters);

4.  in.open();

5.  ps.createStreamStructure("conf.stream");

6.  ps.setInput(in);

7.  Packet* p = ps.getPacket();

8.  cout << "The packet reads from input is" << p->name << endl;

9.  cout << "The value of field 4 of header is " << p->header->getFieldValue(4) << endl;

10. cout << "The value of field 2 of source data field is " << p->dataField->sourceDataField->getFieldValue(2) << endl;
```

Line 1 instantiates the object that represents the input byte stream, and line 5 loads the configuration files and creates into memory the byte stream structure and the packets structure (header, data field and fields). Lines 2, 3, and 4 create an input source and open it. To change the input source without modifying the rest of the code, it is only necessary to modify the line 3 by instantiating another type of input (e.g. InputFile instead of InputSocket), and opening it with the correct parameters.

The line 6 links the input byte stream with the input source, and the remaining code extracts the required packet information from the input stream. In particular, line 7 reads a complete packet, whereas lines 8-10 print the value of the various fields.

Summarizing, lines 2-4,6 provide the I/O Abstraction layer, while lines 1,5,7-10 are related to the Telemetry Management layer.

From this example it is evident that with a few lines of code it is possible to connect with an input source and obtain an object representing a packet with all the field value decoded starting from a byte flow. The library is able to manage either big-endian and little-endian format.

For a full comprehension of how the library works, it is necessary to understand how the input is processed. After having read the packet Header (which is of fixed length), the library knows the length of the following Data Field. The actual structure of the Data Filed is derived by its identifier which consists of one or more fields containing some predefined values. Typically, the APID field of CCSDS standard is used, but the library has not limitation, and allows identifiers which include others fields (e.g.: in the AGILE case the Type/Subtype fields of the Data Field Header).

The identification of the various packets is performed by looking in the packets for one of the possible identifier defined in the *.packet ASCII file. Identified packets are unpacked and the various fields are read into the packet structure defined in the *.packet files, where can be directly addressed as shown in lines 9,10. In the negative case, an object representing a generic not recognized packet is created, where the Data Field byte sequence is copied without any structure.

6 DEVICE EXAMPLES

This section presents some examples that use the device hierarchy. Note that the device behaviour depends by the device.

6.1 File

The file declaration is the following:

```
bool bigendian;
bigendian = ...   //true or false

File f(bigendian);
```

The bigendian parameter is useful only if we work with a stream of byte..

Open the file:

```
char* filename = “prova”;
f.open(filename);
```

Open the file in writing mode:

```
f.open(filename, “w”);
```

If a file is opened in reading mode, it is possible to read the content with the following methods:

```
int val;
ByteStream* b;
char* str;

val = f.getByte();      //read single byte

b = f.getNByte(200);    //read 200 bytes and return a ByteStream

str = f.getLine();      //if it is a text file, read a string
```

If the file is opened in writing mode:

```
f.writeString(str);     //write a string

f.writeByteStream(b);   //write a ByteStream
```

Use isEOF() to know if we are a the end of file, or use bool isClosed() to know if the file is closed.

Methods that are useful to change the position into the file:

- long getpos()

- long setpos(long offset)
- bool memBookmarkPos(): saves the current position in the file
- bool setLastBookmarkPos(): moves in the last saved current position

Close the file with:

```
f.close()
```

These method should be closed into a try…catch block and should be managed the exception PacketExceptionIO*.

This is an example that came from the QpacketViewer. This code write into a log file the content of a telemetry packet.

```
fo.writeString("\n\nHeader:\n ");
c1 = p->header->stream->printStreamInHexadecimal();
fo.writeString(c1);
fo.writeString("\n");
c = p->header->printValue("\r\n");
for(int i=0; c[i] != 0; i++)
      fo.writeString(c[i]);
```

fo is the input file. p is an object of class Packet. c1 is a char*.

6.2 SHM

This example is divided into two parts: SHM Server that creates and writes the shared memory, SHM Client that connects and reads the shared memory.

6.2.1 SHM SERVER

```
#include <iostream.h>
#include <stdlib.h>
#include "PacketExceptionIO.h"
#include "SHM.h"

using namespace PacketLib;

typedef struct s {
      long pointer;
      char filename[200];
} str_idl;



#define SHM_PROCESSOR_SYNC 10000

#define KEY_SYNC(acq_type, ch)  SHM_PROCESSOR_SYNC+#acq_type*100 + #ch

int main(int argc, char *argv[])
```

```
{
try {
      cout << "Create the shared memory" << endl;
      SHM* shm = new SHM(true);
      int i;
      shm->create(100000, 1, sizeof(shm_sync));
      shm->open();
      str_idl* s = (str_idl*) new str_idl;

      s->pointer=10;

      shm->writeSlot(0, (void*) s);

      scanf("writed ... %d", &i);
      shm->close();
      scanf("%d", &i);

      delete shm;
}
catch(PacketExceptionIO* e) {
      cout << e->geterror() << endl;
}

}
```

6.2.2 SHM CLIENT

```
#include <iostream.h>
#include <stdlib.h>
#include "PacketExceptionIO.h"
#include "SHM.h"
#include <unistd.h>

using namespace PacketLib;

typedef struct s1 {
      long pointer;
      char filename[200];
} str_idl;

typedef struct s {
      unsigned short status;
      unsigned long fits_raw_number;
      unsigned long run_id;
      char filename[500];
} shm_sync;

int main(int argc, char *argv[])
{
```

```
try {
      //2
      cout << "Connect with the shared memory" << endl;
      SHM* shm = new SHM(true);
      int i;

      shm->open(10301, 1, sizeof(shm_sync));
      while(true) {
            shm_sync* arr = (shm_sync*) shm->readSlot(0);
            cout << arr->status << endl;
            cout << arr->fits_raw_number << endl;
            cout << arr->run_id << endl;
            cout << arr->filename << endl;
            cout << "-------------------" << endl;
            sleep(1);
      }
      scanf("%d", &i);
      shm->close();
      delete shm;
}
catch(PacketExceptionIO* e) {
      cout << e->geterror() << endl;
}

}
```

6.3 MSGQ

6.3.1 MSGQ SERVER

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <iostream.h>
#include <stdlib.h>
#include "MSGQ.h"
#include "PacketException.h"

using namespace PacketLib;

#define  SENDER_MSGQ_LEN  255
#define  SENDER_MSG_TYPE  10

int main(int argc, char *argv[])
{
  try {
    char* buf;
```

```
    MSGQ msgq(true);
    msgq.create(10000, SENDER_MSGQ_LEN);
    cout << "Wait for a message..." << endl;

    buf = msgq.readMessage(SENDER_MSG_TYPE);

    for(int i=0; i< SENDER_MSGQ_LEN; i++)
            cout << buf[i];

    msgq.destroy();
    cout << "End of the program" << endl;
  }
  catch(PacketException* e) {
    cout << e->geterror() << endl;
  }
}
```

6.4 MSGQ Client

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <iostream.h>
#include <stdlib.h>
#include "MSGQ.h"
#include "PacketException.h"

using namespace PacketLib;

#define  SENDER_MSGQ_LEN  255
#define  SENDER_MSG_TYPE  10

int main(int argc, char *argv[])
{
  try {
    char* mtext = new char[SENDER_MSGQ_LEN];

    MSGQ msgq(true); //create the object

    msgq.open(10000, SENDER_MSGQ_LEN); //open the msgq

    //set the values
    for(int i = 0; i < SENDER_MSGQ_LEN; i++)
            mtext[i] = 35;

    //write the message
    msgq.writeMessage(mtext, (long)SENDER_MSG_TYPE);
```

```
    cout << "End of the program" << endl;
  }
  catch(PacketException* e) {
    cout << e->geterror() << endl;
  }
}
```

7 OUTPUT EXAMPLES

7.1 General description

The main steps to create an output data flow are the following:
1) create an OutputPacketStream object;
2) create an object of the Output hierarchy
3) open the .stream file. This create the structure of the telemetry packets that should be generated.
4) open the Output object with the correct parameters
5) connect the OutputPacketStream object with the Output object created and opened
6) elaborate the Packets and ByteStreams objects
7) close the output (if it is necessary).

7.2 Example 1

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <iostream.h>
#include <stdlib.h>
#include "OutputSocketClient.h"
#include "OutputPacketStream.h"
#include "ByteStream.h"
#include "Output.h"
#include "OutputFile.h"

using namespace PacketLib;

int main(int argc, char *argv[])
{
        Output* out;
        OutputPacketStream ops;

        //create the structure of the telemetry in memory
        ops.setFileNameConfig("../../../CAL-DFE-TE_configuration/BURST.stream");
        ops.createStreamStructure();

        //parameter for the output: socket client
        out = (Output*) new OutputSocketClient(ops.isBigEndian());
        cout << "OUTPUT: SOCKET CLIENT 20001" << endl;
        char** param = (char**) new (char*)[3];
        param[0] = "localhost"; //host
        param[1] = "20002";     //port
        param[2] = 0;
```

```
        //open output
        out->open(param);

        //connect the output
        ops.setOutput(out);

        //get a packet and set some fields
        Packet* p = ops.getPacketType(2);
        p->header->setFieldValue(3, 257);
        p->header->setFieldValue(5, 1);

        //send the packet
        ops.writePacket(p);

        cout << p->packet_output->printStreamInHexadecimal()    << endl;
        //close the output
        out->close();


}
```

If we want to send a ByteStream:

```
        //send a byte stream to the output
        ByteStream* bl = new ByteStream(10, ops.isBigEndian());
        for(int i=0; i< 10; i++)
                bl->setByte(i, i);

        out->writeByteStream(bl);
```

If we want to use a file as output:

```
        //parameter for the output: file
        out = (Output*) new OutputFile(ops.isBigEndian());
        char** param = (char**) new (char*)[2];
        param[0] = "../../test.raw"; //file name
        param[1] = 0;
```

8 INPUT EXAMPLES

8.1 General description

The main steps to create an output data flow are the following:
1) create an InputPacketStream object;
2) create an object of the Input hierarchy
3) open the .stream file. This create the structure of the telemetry packets that should be generated.
4) open the Input object with the correct parameters
5) connect the l'InputPacketStream object with the Input object created and opened
6) elaborate the Packets or ByteStreams objects
7) close the input (if it is necessary).

8.2 Example 1

Iin this example the stream is read from file.

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <iostream.h>
#include <stdlib.h>
#include "InputSocketServer.h"
#include "InputPacketStream.h"
#include "Input.h"
#include "InputFile.h"
#include "Packet.h"

int main(int argc, char *argv[])
{
      Input* in;

      InputPacketStream ips;
      ByteStream* bs;
      bool endcycle = false;

      //2) create input
      in = (Input*) new InputFile(false);

      //3) create the structure of the telemetry in memory
      ips.setFileNameConfig("../../../CAL-DFE-TE_configuration/BURST.stream");
      ips.createStreamStructure();


      //parameter of the input: file
      cout << "INPUT: FILE" << endl;
      char** param = (char**) new (char*)[2];
```

```
	param[0] = "../../test.raw";
	param[1] = 0;

	//open input
	in->open(param);

	//5) set a particular input
	ips.setInput(in);

	//6) input data elaboration

	...

	//7) close input
	in->close();

}
```

8.3 Reading from a socket

Modification required in order to read the input from a socket::

```
	//2) create input
	in = (Input*) new InputSocketServer(true);

	//4) parameter for input: socket server
	cout << "INPUT: SOCKET SERVER 20001" << endl;
	char** param = (char**) new (char*)[2];
	param[0] = "20001";	//port
	param[1] = 0;
```

8.4 Reading a ByteStream

```
	//6) elaboration of the input data

	//read the byte stream from the input
	while(!endcycle) {
		cout << "Waiting data..." << endl;
		bs = in->readByteStream(10);
		if (bs != 0) {
			cout << bs->printStreamInHexadecimal() << endl;
			cout << "------------------------" << endl;
		}
		else {
			if(in->isEOF())
				endcycle = true;
		}
	}
```

8.5 Reading a Packet

To read a packet from input:

```
Packet* p = ips.readPacket();
```

If the data read don't contain a recognized packet, a pointer to PacketNotRecognized is obtained.
If p is correct, it is possible to elaborate the packet:

```
cout << p->getName() << endl;

cout << p->header->getFieldValue(4) << endl;
```

8.6 InputPacketStreamFile example

See the code of the QPacketViewer as example of the use of the InputPacketStreamFile class.

9 EXAMPLE OF LAYOUT 4

The Layout 4 (see Interface Control Document) is the most complex layout of the PacketLib. The code provided in this Chapter is useful to understand how to work with this type of layout. The example provided comes from a telemetry generator.

This example illustrates in particular how to navigate a Layout 4 packet.

The example is based on the Layout 4 example and .packet and .rblock files described in the Interface Control Document.

```
ByteStream* make3901() {
ByteStream* b;
SDFRBlock* sdf;
SDFRBBlock* bi, *bcX, *bcZ, *TAA1, *mcal;
static unsigned int sec = 0;
static unsigned short msec = 0;

        //get the packet to elaborate
        Packet* p = opstream->getPacketType(PACKET_3901); //3901

        //-------------------------------------------------------
        //set the header values ---------------------------------
        //-------------------------------------------------------
        p->header->setFieldValue(3, APIDTM);

        //-------------------------------------------------------
        //set the data field header values ----------------------
        //-------------------------------------------------------
        p->dataField->dataFieldHeader->setFieldValue(3, 39);  //type: the value 39 in the field 3
        p->dataField->dataFieldHeader->setFieldValue(4, 1);   //subtype

        //-------------------------------------------------------
        //work with source data field (sdf) ---------------------
        //-------------------------------------------------------

        //Get a pointer to the source data field
        sdf = (SDFRBlock*) p->dataField->sourceDataField;

        //set the number of blocks of the sdf
        sdf->setNumberOfRealDataBlock(block_3901_events, 0); //N blocks for rtype 0

        //set the fixed part of the source data field ----------------------
        sdf->setFieldValue(0, 10);    //set the value 10 in the field 0
        sdf->setFieldValue_5_1(0, 12.3);
        sdf->setFieldValue_4_14(8, 123456789);

        sdf->setFieldValue(15, 1);  //orbital phase
        sdf->setFieldValue(17, 3);  //Grid conf mode

        //work with variable part of the sdf ---------------------------------

        //for each block of the sdf, set the number of sub-blocks
        for(int i=0; i < block_3901_events; i++) {
                //get the block i of type 0 of the source data field
                bi = sdf->getBlock(i, 0);
                //set the number of blocks of sub-block of type 0 of block i of the sdf
```

```
        bi->setNumberOfRealDataBlock(block_3901_1, 0);
        //set the number of blocks of sub-block of type 1 of block i of the sdf
        bi->setNumberOfRealDataBlock(block_3901_2, 1);
        //set the number of blocks of sub-block of type 2 of block i of the sdf
        bi->setNumberOfRealDataBlock(block_3901_3, 2);
        //set the number of blocks of sub-block of type 3 of block i of the sdf
        bi->setNumberOfRealDataBlock(block_3901_4, 3);
}

//EXAMPLE: work with the first block of the sdf

//get the first block of the sdf, bi
bi = sdf->getBlock(0,0);

//set the fixed part of the of the first block of the sdf
bi->setFieldValue(13, 2730);   //ACTOPCON
bi->setFieldValue(15, 182);   //SAIE 1 Upper thres

//set the fixed part of the sub-blocks of type 0 (called ST cluster)
for(int i=0; i<block_3901_1; i++) {
        bcX = bi->getBlock(i,0);
        bcX->setFieldValue(0,0);    //FEB 0
        bcX->setFieldValue(1,2);    //CHIP 2
        bcX->setFieldValue(2,100);    //strip
        bcX->setFieldValue(3,101);    //total charge
        bcX->setFieldValue(4,102);    //total width
        bcX->setFieldValue(5,105);    //c3
        bcX->setFieldValue(6,103);    //c1
        bcX->setFieldValue(7,104);    //c2
        bcX->setFieldValue(8,104);    //c4
        bcX->setFieldValue(9,103);    //c5
}

//set the fixed part of the sub-blocks of type 2 (called MCAL zero suppressed)
static int bar;
static int energy;
for(int i=0; i<block_3901_3; i++) {
        mcal = bi->getBlock(i, 2);
        mcal->setFieldValue(1, bar++);
        if(bar > 29) bar = 0;
        mcal->setFieldValue(3, energy++);
        mcal->setFieldValue(4, energy++);
}

//set the fixed part of the sub-blocks of type 2 (called TAA1 triggered)
static int feb=0;
static int chip =0;
for(int i=0; i<block_3901_4; i++) {
        TAA1 = bi->getBlock(i, 3);
        TAA1->setFieldValue(1, feb++);
        if(feb>11) feb = 0;
        TAA1->setFieldValue(2, chip++);
        if(chip>23) chip = 0;
}

//-------------------------------------------------------
//set the tail of the packet (the CRC value) -------------
```

```
        //-------------------------------------------------------
        p->dataField->tail->setFieldValue(0, 11);

        //set the source sequence counter
        p->header->setFieldValue(5, ssc++);

        //generate the packet
        b = p->getOutputStream();

        //-------------------------------------------------------
        //if a prefix is present, set the prefix ------------------
        //-------------------------------------------------------
        if(opstream->thereIsPrefix()) {
                if(opstream->getPrefixDimension() == 2)
                        b->setWord(0, p->getDimension());
        }

        return b;
}
```

10 ANNEX A – CONSIDERATION ABOUT BYTESTREAM (ITALIAN)

Per gestire la memoria allocata da questa classe è necessario comprendere in dettaglio il funzionamento di questa classe e di tutti i suoi metodi. In particolare la classe può lavorare in due modalità distinte dal punto di vista della memoria:

- la classe ha allocato la memoria necessaria per memorizzare lo stream di byte
- la classe non ha allocato tale memoria, ma gestisce solamente il puntatore byte*: la responsabilità dell'allocazione di tale memoria è esterna alla classe

A queste due modalità di funzionamento (allocazione/non allocazione) corrispondono anche due categorie di costruttori:

1) costruttori che non allocano memoria

- ByteStream(bool bigendian=false): semplice costruttore.
- ByteStream(byte* stream, long dim, bool bigendian, bool memory_sharing): costruttore che richiede il byte stream, la sua dimensione e la sua rappresentazione in bigendian format. Memory sharing indica se la memoria deve essere condivisa tra più ByteStream o meno. Se true la memoria viene condivisa e quindi quando verrà invocato il distruttore questo non deallocherà memoria. Effettua lo swap dei byte se necessario e memory_sharing=false.

2) costruttori che allocano memoria

- ByteStream(long size, bool bigendian): costruttore che alloca un byte stream di dimensione dim e con una specifica rappresentazione
- ByteStream(ByteStream* b0, ByteStream* b1, ByteStream* b2): costruttore che unisce i ByteStream passati come argomento (che può anche essere 0, in questo caso non ha effetto) in un unico ByteStream, allocando la memoria necessaria. La deallocazione dei ByteStream passati come argomento rimane esterna alla classe.

Il distruttore dealloca la memoria occupata dal byte stream solamente se questa è stata da lui creata oppure è stato passato dall'esterno un byte* con l'attributo memroy_sharing = false.

Sono presenti alcuni attributi statici per il monitoraggio del comportamento del programma

ByteStream risolve anche l'endianity. Questo implica che quando un byte* viene passato come argomento di metodo o di costruttore, se l'architettura è little endian e si vuole un file bigendian viene effettuato lo swap delle coppie di byte. Lo stesso nel caso di architettura bigendian e formato file little endian. Quando viene restituito un byte*, prima di effettuare l'operazione viene rieffettuato lo swap.

Formato file	Little-endian	Big-endian
Architettura		
Little-endian	No	Yes
Big-endian	Yes	no

L'attributo byte rimane per un accesso rapido, ma se si vuole inviare su output utilizzare

byte* getStream().

Non assegnare mai direttamente o leggere direttamente il byte* stream.

Questa classe è anche in grado di generare nuovi ByteStream a partire da ByteStream esistenti:

- ByteStream* getSubByteStream(word first, word last): non viene creata memoria. Consente a più ByteStream di condividere la stessa area di memoria, ottenendo un reference ad un sottoinsieme dello stream iniziale (è attivo il memory sharing)

- ByteStream* getSubByteStreamCopy(word first, word last): viene creata nuova memoria. Si ottiene un ByteStream che dispone di una copia di parte dello stream (il memory sharing è disattivo)
- void ByteStream:: setStreamCopy (byte* b, unsigned dim): elimina lo stream presente e copia quello passato come parametro. La classe ha la responsabilità di deallocare la sua copia locale, ma non il byte* b passato come argomento. Effettua lo swap dei byte se necessario.
- bool ByteStream::setStream(byte* b, unsigned dim, bool bigendian, bool memory_sharing=true): elimina lo stream presente e assegna direttamente il puntatore. La responsabilità di deallocare la memoria dipende dal valore del parametro memory_sharing. Effettua lo swap dei byte se necessario e memory_sharing=false.
- bool ByteStream::setStream(ByteStream* b, word first, word last): elimina lo stream corrente e condivide lo stream con un altro ByteStream. La responsabilità della gestione della memoria rimane del ByteStream passato come argomento

mem_allocation indica se il distruttore deve deallocare il puntatore al byte*. True allora dealloca.

Per accedere direttamente allo byte* stream:

- byte* getStream(): restituisce il byte* in rappresentazione big endian

Per le operazioni di output, sono presenti due metodi di basso livello, utilizzabili dai Device:

- byte* getOutputStream(): restituisce il byte* in rappresentazione dipendente dalla macchina
- void endOutputStream(): rimette il byte* in rappresentazione big endian

Richiamare il primo metodo, inviare lo stream ottenuto su output e quindi richiamare il secondo metodo.