PacketLib 3.4.0 Programmer's Guide

R.I. CIWS-IASFBO-TN-010

Custodian:	Name:	Andrea Bulgarelli	Signature:		Date:	
Prepared by:	Name:	Andrea Bulgarelli	Signature:		Date:	
Reviewed by:	Name:		Signature:		Date:	
Approved by:	Name:		Signature:		Date:	

All information contained in this document is property of **INAF.** All rights reserved.

AUTHOR LIST

Andrea Bulgarelli, Andrea Zoli	INAF/IASF Bologna, Italy

DISTRIBUTION LIST

CIWS e-mail list	ciws@iasfbo.inaf.it

All information contained in this document is property of **INAF.** All rights reserved.

DOCUMENT HISTORY

Version	Date	Modification
d0.1	31 March 2014	First draft

All information contained in this document is property of **INAF.** All rights reserved.

TABLE OF CONTENTS

1. INTRODUCTION 3
1.1 SCOPE AND PURPOSE OF THE DOCUMENT 3
1.2 REFERENCE DOCUMENTS 3
2. ARCHITECTURAL OVERVIEW 4
2.1 THE MAIN LAYERS 4
2.2 PACKAGE 4
2.3 BASIC TYPE 5
3. DESIGN OVERVIEW 7
3.1 EXCEPTION MANAGEMENT 7
3.2 I/O ABSTRACTION LAYER 7
3.2.1 ByteStream class 7
3.2.2 Device hierarchy 7
3.2.2.1 File 7
3.2.2.2 DISCoSSHM 7
3.2.2.3 Socket 7
3.2.2.4 SHM 7
3.2.3 Input hierachy 10
3.2.3.1 Output Hierarchy 10
3.3 MAIN LAYER 12
3.3.1 PacketStream 12
3.3.1.1 InputPacketStream::readPacket() method 13
3.3.2 Packet 14
3.3.2.1 Common structure: PartOfPacket 14
3.3.2.2 The class packet 15
3.3.2.3 Navigation method of the packet and how to work with field values 16
3.3.2.4 Get the field 19
3.3.2.5 Packet structure 20
3.3.2.6 Memory management 20
3.3.2.7 Direct access to the byte stream 20
4. OVERVIEW: HOW THE LIBRARY WORKS 23
5. DEVICE EXAMPLES 25
5.1 FILE 25
5.2 SHM 26
5.2.1 SHM Server 26
5.2.2 SHM Client 28
5.3 MSGQ 29
5.3.1 MSGQ Server 29
5.4 MSGQ CLIENT 30
6. OUTPUT EXAMPLES 32
6.1 GENERAL DESCRIPTION 32
6.2 EXAMPLE 1 32
7. INPUT EXAMPLES 35
7.1 GENERAL DESCRIPTION 35
7.2 EXAMPLE 1 35
7.2.1 InputFILE 36
7.3 READING FROM A SOCKET 37
7.4 READING A BYTESTREAM 38
7.5 READING A PACKET 38
7.6 INPUTPACKETSTREAMFILE EXAMPLE 39

All information contained in this document is property of **INAF**. All rights reserved.

7.6.1 NAVIGATION OF THE STREAM POINTER 40

8. PERFORMANCE CONSIDERATIONS 41

8.1 MEMORY MANAGEMENT 41
8.2 MULTITHREADING CONSIDERATIONS 41
8.3 PACKET STREAM MANAGEMENT 42
8.3.1 Build a list of pointer 43
8.3.2 Build a list of Bytestream objects 45
8.3.3 Build a list of packets objects 48

9. EXAMPLES ABOUT LAYOUTS 50

9.1 EXCEPTION MANAGEMENT 50
9.2 LAYOUT 2 50
9.3 LAYOUT 3 56
9.4 LAYOUT 4 59

10. ANNEX A – CONSIDERATION ABOUT BYTESTREAM (ITALIAN) 65

All information contained in this document is property of **INAF.** All rights reserved.

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].

1.2 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, et al. “PacketLib Interface Control Document”,

All information contained in this document is property of **INAF.** All rights reserved.

2. ARCHITECTURAL OVERVIEW

2.1 The main layers

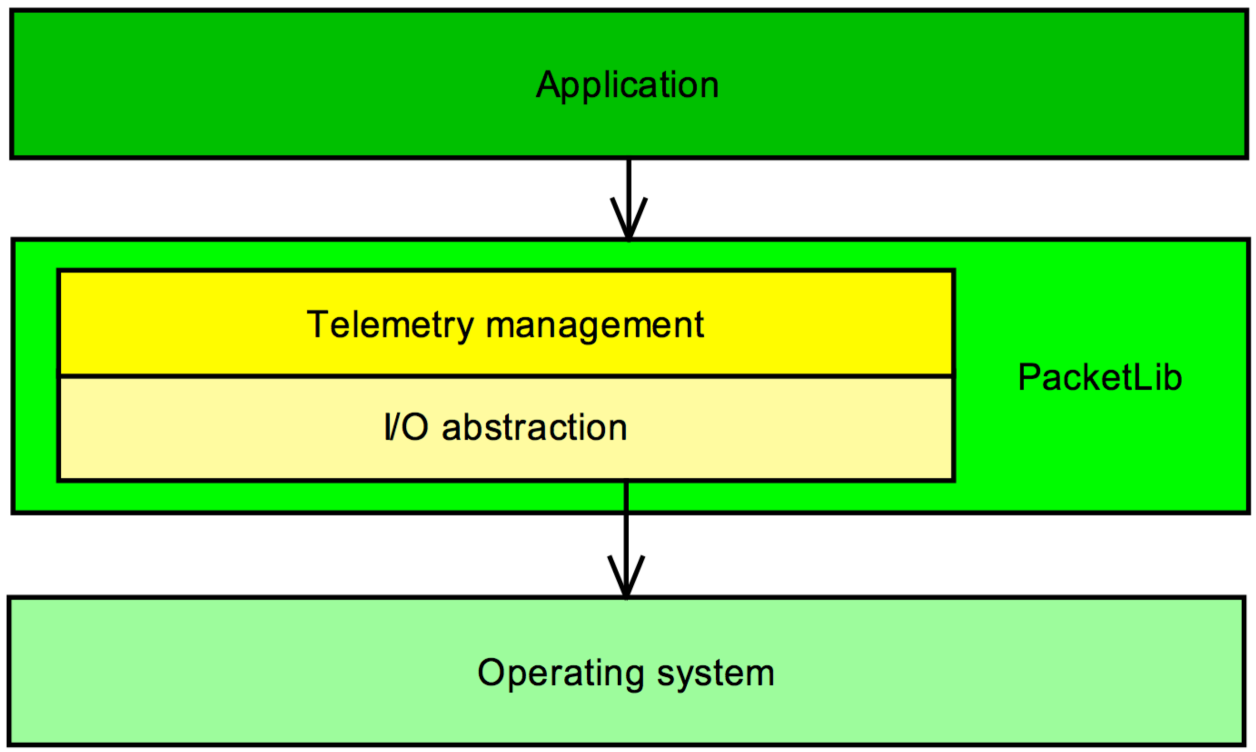


Figura 3.1: PacketLib layers

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.

2.2 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 4.2 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.

All information contained in this document is property of **INAF.** All rights reserved.

2.3 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.

All information contained in this document is property of **INAF.** All rights reserved.

All information contained in this document is property of **INAF.** All rights reserved.

3. Design overview

3.1 Exception management

PacketLib provides some exception classes, that are used to generate some exceptions during the I/O operations or during the telemetry packets elaboration. The classes involved are the following:

- the base class PacketException
- the derived classes PacketExceptionIO and PacketExceptionFileFormat.

When an exception is generated, use the geterror() or geterrorcode() methods to get the message and error code.

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.2.1 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.

3.2.2.2 DISCoSSHM

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

3.2.2.3 Socket

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

3.2.2.4 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

All information contained in this document is property of **INAF.** All rights reserved.

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.

```
ByteStream
+stream : byte*
-byteInTheStream : dword
-bigendian : bool
-mem_allocation : bool
-mem_allocation_constructor : bool
+ByteStream(bigendian : bool = false)
+ByteStream(size : dword, bigendian : bool)
+ByteStream(stream : byte *, dim : dword, bigendian : bool, memory_sharing : bool = true)
+ByteStream(b0 : ByteStreamPtr, b1 : ByteStreamPtr, b2 : ByteStreamPtr)
+ByteStream(b0 : ByteStreamPtr, start : dword, end : dword = -1, memory_sharing : bool = true)
+~ByteStream()
+getStream() : byte *
+getOutputStream() : byte *
+endOutputStream() : void
+getSubByteStream(first : dword, last : dword) : ByteStreamPtr
+getSubByteStreamCopy(first : dword, last : dword) : ByteStreamPtr
+setStream(b : byte *, dim : dword, bigendian : bool, memory_sharing : bool = true) : bool
+setStream(b : ByteStreamPtr, first : dword, last : dword) : bool
+setStreamCopy(b : byte *, dim : dword) : void
+setWord(start : dword, value : word) : bool
+setByte(start : dword, value : word) : void
+getByte(byteNumber : dword) : byte
+getValue(start : dword, dim : word) : long
+getDimension() : dword
+printStreamInHexadecimal() : char *
+getMemAllocation() : bool
+isBigendian() : bool
+swapWordIfStreamIsLittleEndian() : void
+swapWordIfStreamIsBigEndian() : void
+swapWordForIntel() : void
+swapWord() : void
#setMemoryAllocated(allocated : bool) : void
#deleteStreamMemory() : void
```

Figure 1: ByteStream class

All information contained in this document is property of **INAF.** All rights reserved.

Device
#eof : bool
#closed : bool
#bigendian : bool
#endsession : bool
+Device(bigendian : bool)
+isClosed() : bool
+isEOF() : bool
+setEndSession(_newVal : bool &) : void
+getEndSession() : bool &
+isBigEndian() : bool

File
+byte_read : dword
+char_read : dword
#lastLineRead : char*
#fp : FILE*
#bookmarkPos : long
#startPosition : long
#filename : char*
#mode : char*
+File()
+File(b : bool, startP : long = 0)
+~File()
+open(filename : char *, mode : char * = "r") : bool
+setFileName(filename : char *) : void
+getFileName() : char *
+getByte() : int
+getNByte(N : dword = 1) : ByteStreamPtr
+getLine() : char *
+close() : void
+isClosed() : bool
+getLastLineRead() : char *
+setpos(offset : long) : long
+getpos() : long
+memBookmarkPos() : bool
+setLastBookmarkPos() : bool
+isEOF() : bool
+setFirstPos() : long
+fchdir() : bool
+find(b : byte) : long
+writeString(str : char *) : bool
+writeByteStream(b : ByteStreamPtr) : bool
+getNumberOfStringLines() : long
+fsize() : long

SHM
#shmid : int
#shmkey : long
#num_slot : dword
#slot_dim : dword
#created : bool
#shm : byte*
+SHM(bigendian : bool)
+~SHM()
+create(shmkey : long, num_slot : dword, slot_dim : dword) : bool
+open(shmkey : long = 0, num_slot : dword = 0, slot_dim : dword = 0) : bool
+close() : bool
+destroy() : bool
+isCreated() : bool
+writeSlot(n_slot : dword, element : void *) : void
+readSlot(n_slot : dword) : void *

MSGQ
+msgqid : int
+msgqkey : long
+created : bool
+dimmsg : long
+buf : msgbuf
+MSGQ(bigendian : bool)
+~MSGQ()
+create(shqkey : long, dim : long) : bool
+open(shqkey : long, dim : long) : bool
+close() : bool
+destroy() : bool
+isCreated() : bool
+writeMessage(msg : char *, mtype : long) : void
+readMessage(mtype : long) : char *

Socket
#m_sock : int
#m_addr : sockaddr_in
+Socket(bigendian : bool)
+~Socket()
+is_valid() : bool
+send(b : ByteStreamPtr) : bool
+recv(dim : word, status : int &) : ByteStreamPtr
+connect(host : string, port : int) : bool
+create() : bool
+close() : bool

SocketClient
+SocketClient(bigendian : bool, host : string, port : int)

SocketServer
+SocketServer(bigendian : bool)
+SocketServer(bigendian : bool, port : int)
+~SocketServer()
+bind(port : int) : bool
+listen() : bool
+set_non_blocking(bool) : void
+accept(SocketServer &) : bool

Figure 2: devices

All information contained in this document is property of **INAF**. All rights reserved.

3.2.3 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

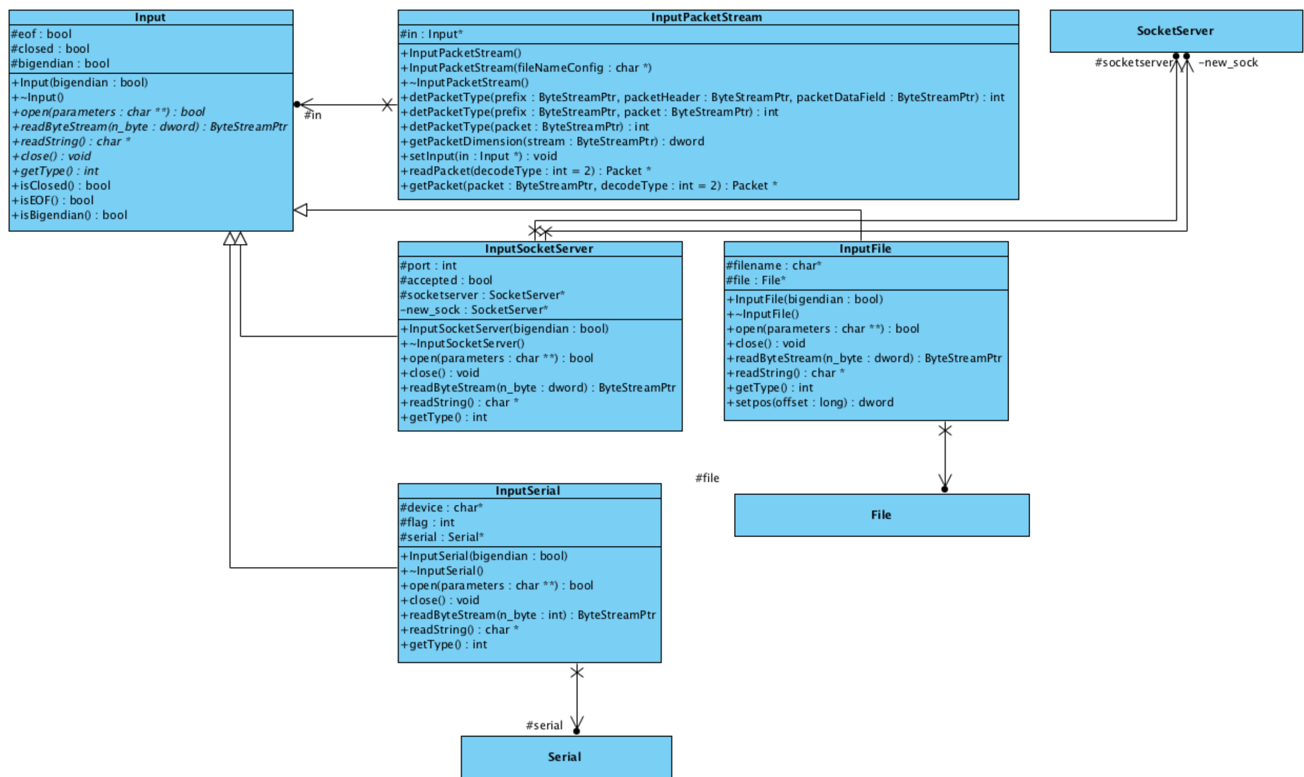


Figure 3: inputs

3.2.3.1 Output Hierarchy

Output abstraction layer. See Chapter 6 for some example.

The main methods are:

- *open(char** parameters)*: open the output (see 4.2.3)
- *close()*: close the output
- *writeByteStream(ByteStream* b)*: write a ByteStream to the output

All information contained in this document is property of **INAF**. All rights reserved.

- *writeString(char* c)*: write a string

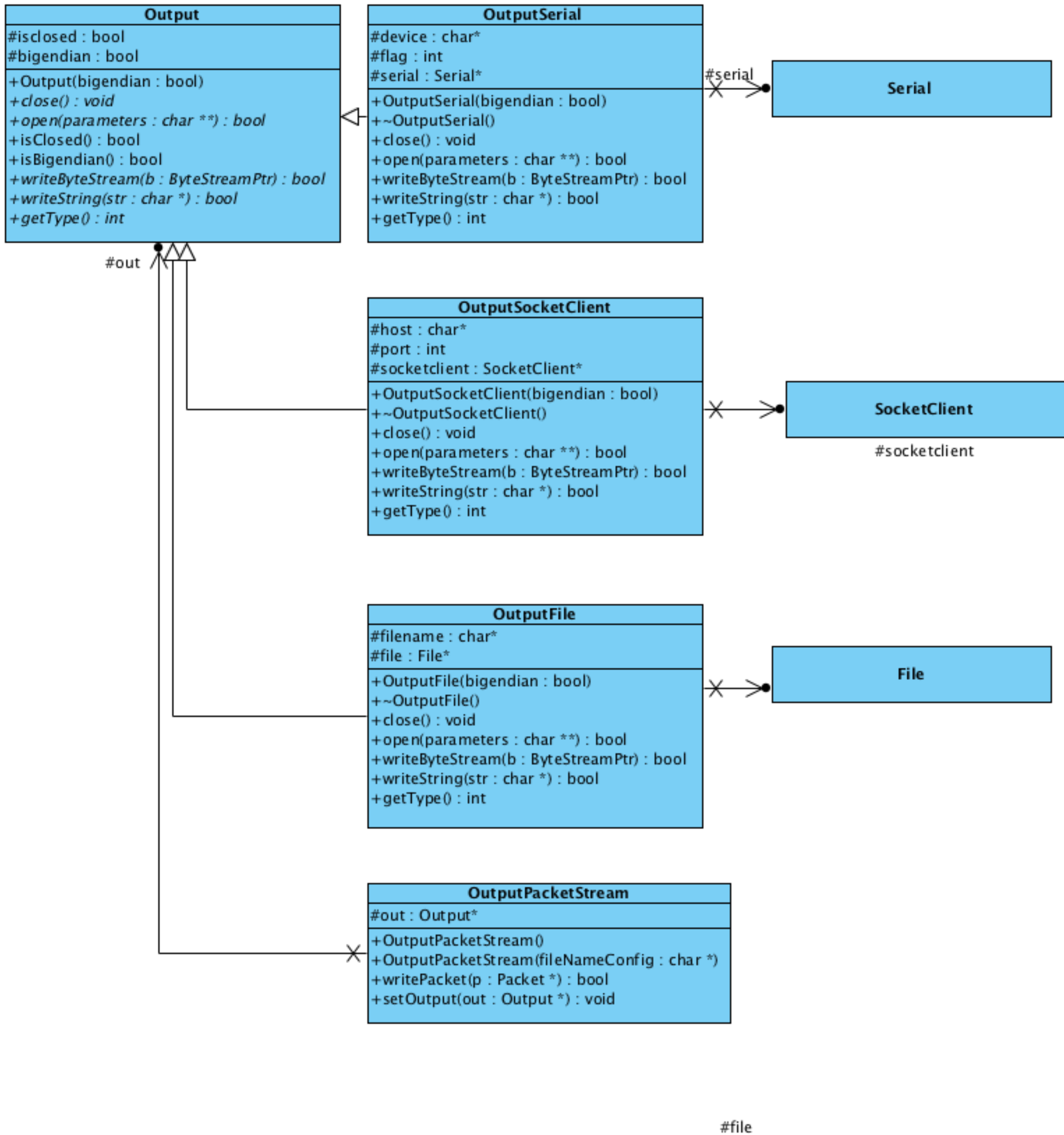


Figure 4: output

All information contained in this document is property of **INAF**. All rights reserved.

3.3 Main Layer

3.3.1 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.

All information contained in this document is property of **INAF.** All rights reserved.

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.

3.3.1.1 InputPacketStream::readPacket() method

The readPacket() method of the InputPacketStream class return a decoded telemetry or telecommand packet obtained from an input source (see Chapter 5 for an example).

If the packet is not recognized (depending on the list of identifier associated with a packet) the packet returned is a not recognized packet.

If there are some problems during the decoding operations, a NULL pointer is returned. A possible problem can occur if the total dimension of a section of the packet (header, source data field, block) are greater than the number of bytes read from input. Controls are not foreseen if the total dimension of a section is less of the number of bytes read from input. This enable to decode a telemetry packet even if the definition of this packet is not complete.

All information contained in this document is property of **INAF.** All rights reserved.

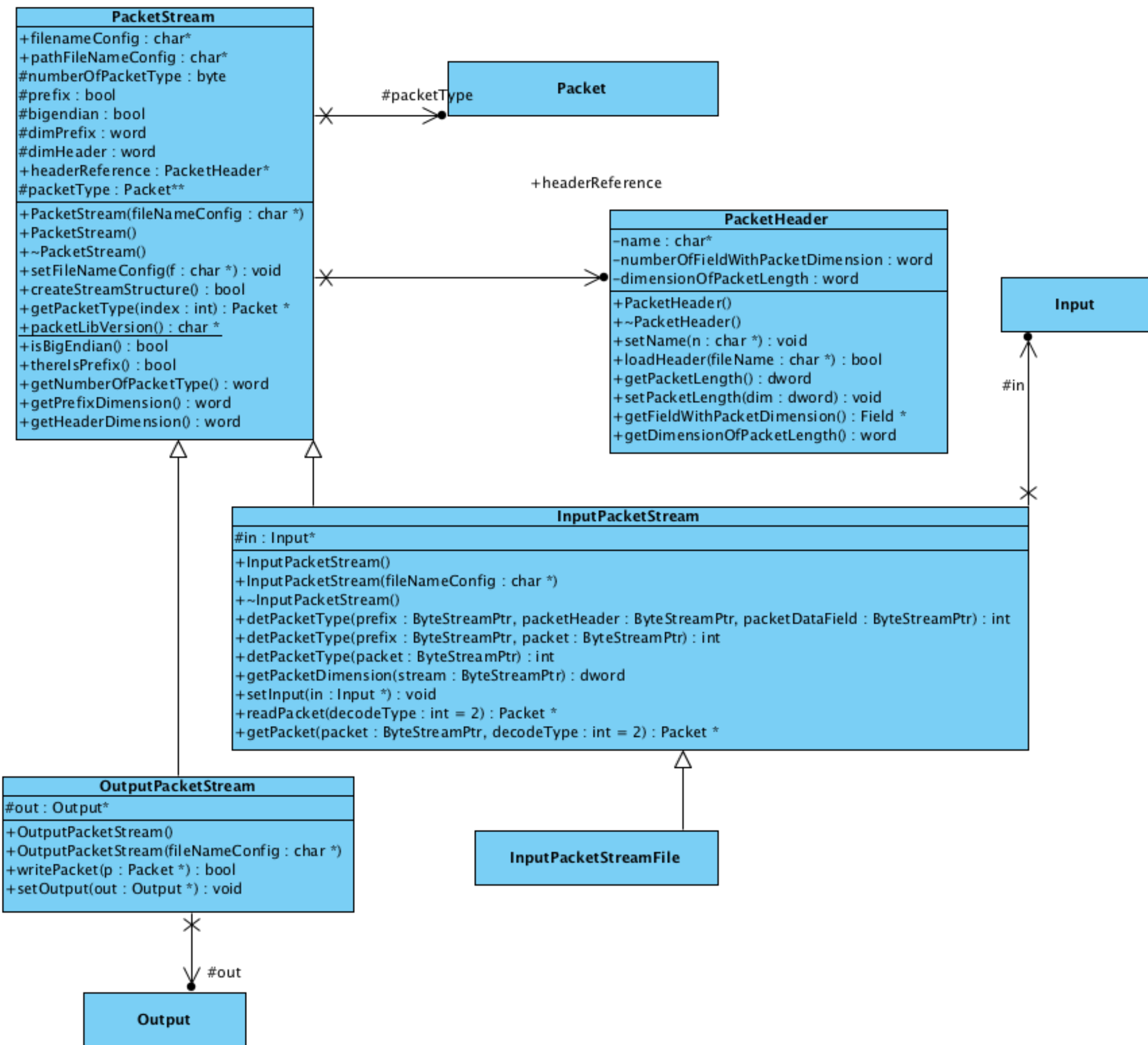


Figure 5: packet Stream

3.3.2 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.

3.3.2.1 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.

All information contained in this document is property of **INAF**. All rights reserved.

Figure 6: part of packet

In the Figure 4.6 are shown all the methods and the sections foreseen for a telemetry packet. For more details about these section see 4.3.2.5.

3.3.2.2 The class packet

The class Packet represents a telemetry packet.

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

All information contained in this document is property of **INAF**. All rights reserved.

```
PacketHeader
-name : char*
-numberOfFieldWithPacketDimension : word
-dimensionOfPacketLength : word
+PacketHeader()
+~PacketHeader()
+setName(n : char *) : void
+loadHeader(fileName : char *) : bool
+getPacketLength() : dword
+setPacketLength(dim : dword) : void
+getFieldWithPacketDimension() : Field *
+getDimensionOfPacketLength() : word

Packet
+prefix : ByteStreamPtr
+packet : ByteStreamPtr
+packet_output : ByteStreamPtr
#packetID : byte
#name : char*
#type_of_identifier : bool[3]
#number_of_identifier : unsigned
#bigendian : bool
#thereisprefix : bool
#dimPrefix : word
#filename : char*
-tempHeader : ByteStreamPtr
-tempDataField : ByteStreamPtr
-tempDataFieldHeader : ByteStreamPtr
-tempPacketDataField : ByteStreamPtr
-tempTail : ByteStreamPtr
-first_output_stream_setted : bool
+header : PacketHeader*
+dataField : PacketDataField*
#identifiers : PacketIdentifier**
+Packet(bigendian : bool)
+~Packet()
+createPacketType(fileName : char *, prefix : bool, dimprefix : word) : bool
+setAndVerifyPacketValue(prefix : ByteStreamPtr, packetHeader : ByteStreamPtr, packetDataField : ByteStreamPtr) : bool
+setAndVerifyPacketValue(prefix : ByteStreamPtr, packet : ByteStreamPtr) : bool
+setByteStreamPointers(prefix : ByteStreamPtr, packetHeader : ByteStreamPtr, packetDataField : ByteStreamPtr) : void
+setByteStreamPointers(prefix : ByteStreamPtr, packet : ByteStreamPtr) : void
+setPacketValue(prefix : ByteStreamPtr, packetHeader : ByteStreamPtr, packetDataField : ByteStreamPtr, onlySections : bool = false) : bool
+setPacketValue(prefix : ByteStreamPtr, packet : ByteStreamPtr, onlySections : bool = false) : bool
+verifyPacketValue(prefix : ByteStreamPtr, packet : ByteStreamPtr) : bool
+verifyPacketValue(stream : byte *) : bool
+setPacketValue(stream : byte *, onlySections : bool = false) : bool
+verifyPacketValue(prefix : ByteStreamPtr, packetHeader : ByteStreamPtr, packetDataField : ByteStreamPtr) : bool
+printPacketValue() : void
+printPrefixStream() : char *
+printHeaderValue() : char **
+printHeaderStream() : char *
+printDataFieldHeaderValue() : char **
+printDataFieldHeaderStream() : char *
+printSourceDataFieldValue() : char **
+printSourceDataFieldStream() : char *
+printTailValue() : char **
+printTailStream() : char *
+printPacketOutputStream() : char *
+getDimension() : dword
+getMaxDimension() : dword
+getInputStream() : ByteStreamPtr
+getOutputStream() : ByteStreamPtr
+getName() : char *
+getFileName() : char *
+printIdentifiers() : void
+setPacketID(value : byte &) : void
+getPacketID() : byte &
+thereIsPrefix() : bool
+isBigendian() : bool
#generateStream() : void
#loadIdentifiers(file : ConfigurationFile &) : bool
#setPacketValuePrefix(prefix : ByteStreamPtr) : bool
#setPacketValueVerify(prefix : ByteStreamPtr, packetHeader : ByteStreamPtr, packetDataField : ByteStreamPtr) : bool
#setPacketValueDataFieldHeader(packetDataField : ByteStreamPtr) : bool
#setPacketValueSourceDataField(packetDataField : ByteStreamPtr, onlySections : bool = false) : bool
#setPacketValueHeader(header : ByteStreamPtr) : bool
#setPacketValueTail(packetDataField : ByteStreamPtr) : bool
#memByteStream(prefix : ByteStreamPtr, packetHeader : ByteStreamPtr, packetDataField : ByteStreamPtr) : void
#memByteStream(prefix : ByteStreamPtr, packet : ByteStreamPtr) : void

+header

PacketDataField
+dataFieldHeader : DataFieldHeader*
+sourceDataField : SourceDataField*
+tail : PartOfPacket*
+PacketDataField()
+~PacketDataField()
+getNumberOfRealDataBlock() : word
+setNumberOfRealDataBlock(number : word) : void
+getDimension() : dword
+getMaxDimension() : dword
+setOutputStream(os : ByteStreamPtr, first : word) : bool
+generateStream(bigendian : bool) : ByteStreamPtr

+dataField

PacketNotRecognized
+PacketNotRecognized(bigendian : bool)
+~PacketNotRecognized()
+createPacketType(fileName : char *, prefix : bool, dimprefix : word) : bool
+setPacketValue(prefix : ByteStreamPtr, packetHeader : ByteStreamPtr, packetDataField : ByteStreamPtr) : bool

PacketStream
+filenameConfig : char*
+pathFileNameConfig : char*
#numberOfPacketType : byte
#prefix : bool
#bigendian : bool
#dimPrefix : word
#dimHeader : word
+headerReference : PacketHeader*
#packetType : Packet**
+PacketStream(fileNameConfig : char *)
+PacketStream()
+~PacketStream()
+setFileNameConfig(f : char *) : void
+createStreamStructure() : bool
+getPacketType(index : int) : Packet *
+packetLibVersion() : char *
+isBigEndian() : bool
+thereIsPrefix() : bool
+getNumberOfPacketType() : word
+getPrefixDimension() : word
+getHeaderDimension() : word

#packetType

PacketIdentifier
+fieldNumber : word
+type : byte
+value : word
+PacketIdentifier(fn : int, t : byte, v : word)
+~PacketIdentifier()

#identifiers
```

Figure 7: Packet

3.3.2.3 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.

All information contained in this document is property of **INAF.** All rights reserved.

Packet

+header

+dataField

PacketHeader
-name : char*
-numberOfFieldWithPacketDimension : word
-dimensionOfPacketLength : word
+PacketHeader()
+~PacketHeader()
+setName(n : char *) : void
+loadHeader(fileName : char *) : bool
+getPacketLength() : dword
+setPacketLength(dim : dword) : void
+getFieldWithPacketDimension() : Field *
+getDimensionOfPacketLength() : word

PacketDataField
+dataFieldHeader : DataFieldHeader*
+sourceDataField : SourceDataField*
+tail : PartOfPacket*
+PacketDataField()
+~PacketDataField()
+getNumberOfRealDataBlock() : word
+setNumberOfRealDataBlock(number : word) : void
+getDimension() : dword
+getMaxDimension() : dword
+setOutputStream(os : ByteStreamPtr, first : word) : bool
+generateStream(bigendian : bool) : ByteStreamPtr

+sourceDataField

PartOfPacket

+tail

+dataFieldHeader

DataFieldHeader
+DataFieldHeader()

SourceDataField
#isblock : bool
#fixed : bool
#rblock : bool
#numberOfBlockFixed : bool*
#maxNumberOfBlock : word*
#numberOfRealDataBlock : word*
#indexOfNBlock : word*
#subFromNBlock : word*
#reset_output_stream : bool
+SourceDataField(sdfName : char * = 0)
+~SourceDataField()
+setByteStream(s : ByteStreamPtr, onlySections : bool = false) : bool
+getMaxDimension() : dword
+getMaxDimension(nblock : word) : dword
+loadFields(fp : InputText &) : bool
+isBlock() : bool
+isFixed() : bool
+isRBlock() : bool
+isNumberOfBlockFixed(rblockIndex : word = 0) : bool
+getMaxNumberOfBlock(rblockIndex : word = 0) : word
+getMaxNumberOfElements(nblock : word) : word
+getDimension() : dword
+getDimension(block : word) : dword
+getIndexOfNBlock(rblockIndex : word = 0) : word
+setNumberOfRealDataBlock(value : word, rblockIndex : word = 0) : void
+getSubFromNBlock(rblockIndex : word = 0) : word
+getNumberOfRealDataBlock(rblockIndex : word = 0) : word
*+getFields(index : word) : Field **
+getFields(block : word, index : word) : Field *
+getFieldValue(index : word) : word
+getFieldValue(block : word, index : word) : word
+getFieldValue_5_1(block : word, index : word) : float
+getFieldValue_5_1(index : word) : float
+setFieldValue_5_1(block : word, index : word, value : float) : void
+setFieldValue_5_1(index : word, value : float) : void
+getFieldValue_4_14(block : word, index : word) : signed long
+getFieldValue_4_14(index : word) : signed long
+setFieldValue_4_14(block : word, index : word, value : signed long) : void
+setFieldValue_4_14(index : word, value : signed long) : void
+getFieldValue_4_13(block : word, index : word) : signed long
+getFieldValue_4_13(index : word) : signed long
+setFieldValue_4_13(block : word, index : word, value : signed long) : void
+setFieldValue_4_13(index : word, value : signed long) : void
+getFieldValue_3_14(block : word, index : word) : unsigned long
+getFieldValue_3_14(index : word) : unsigned long
+setFieldValue_3_14(block : word, index : word, value : unsigned long) : void
+setFieldValue_3_14(index : word, value : unsigned long) : void
+setFieldValue_3_13(block : word, index : word, value : unsigned long) : void
+setFieldValue_3_13(index : word, value : unsigned long) : void
+getFieldValue_3_13(block : word, index : word) : unsigned long
+getFieldValue_3_13(index : word) : unsigned long
+setFieldValue(index : word, value : word) : void
+setFieldValue(block : word, index : word, value : word) : void
+getNumberOfFields(block : word) : word
+getNumberOfFields() : word
+setNumberOfRealElement(block : word, value : word) : void
+getNumberOfRealElement(block : word) : word
+get_reset_output_stream() : bool
+set_reset_output_stream(value : bool) : void
+printInHexadecimal() : char *

Figure 8: packet and main sections

All information contained in this document is property of **INAF**. All rights reserved.

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)
virtual long getFieldValue_4_13 (word index)
virtual void setFieldValue_4_13 (word index, long value)
virtual long getFieldValue_3_14 (word index)
virtual void setFieldValue_3_14 (word index, long value)
virtual long getFieldValue_3_13 (word index)
virtual void setFieldValue_3_13 (word index, long value)
```

In particular, 3,13 and 4,13 are 24 bit data. The setFieldValue 3,13 and 4,13 returns an exception if an incorrect out of range value is passed. The range are the following (see PacketLibDefinition.h):

```
#define U24BITINTGEGERUNSIGNED_MAX 16777215
```

All information contained in this document is property of **INAF**. All rights reserved.

```
#define U24BITINTGEGERSIGNED_MIN -8388607
#define U24BITINTGEGERSIGNED_MAX 8388607
```

The 24 bit field is the only setXY method that generate an exception, because in the other cases the out of range is managed by the type of the argument of the method itself.

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)
```

also with the associated PTF and PFC.

These methods works for Layout 2 and 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.

3.3.2.4 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 4.3.2.3.

For example:

```
Field* field = p->header->getFields(3);
```

All information contained in this document is property of **INAF.** All rights reserved.

```
cout << "field name " << field->name << endl;
cout << "field dimension " << field->dimension << endl;
cout << "field value " << field->value << endl;
```

3.3.2.5 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).

3.3.2.6 Memory management

During the reading operations, when a packet is recognized the external ByteStream read is adopted by the packet. For this reason, the responsibility to deallocate the memory used by the packet is of the same packet.

When we have finished to use the packet, this memory should be deallocated with

```
p->deleteExternalByteStream();
```

3.3.2.7 Direct access to the byte stream

With the PacketLib it is also possible to access directly to the byte stream read from input (or generated to output). To do this, it is possible to use the

```
ByteStream* b = p->packet
```

for the packet obtained from an input and

```
ByteStream* b = p->packet_output
```

for an ALREADY generated packet as output (that set the value of the stream from the value of the fields).

In addition, each PartOfPacket (header, blocks, data field, etc) has it own pointer to the stream. For example, to obtain the pointer of the header

```
p->header->stream
```

or

```
p->header->outputstream
```

respectively and start the elaboration from this instead of the full ByteStream of the packet.

After this you can use the methods of the ByteStream class to read (from p->packet) or modify (to p->packet_output) the streams. You can use the

All information contained in this document is property of **INAF.** All rights reserved.

```
ByteStream::getStream()
```

to get a byte* (unsigned char*) pointer to the stream. It is possible to use this technique to access to some part of packets with a particular structure. In particular, some methods are provided to get a byte or a word directly from the ByteStream (with the endianity management). For example, to get a single byte (b is a ByteStream*) of index 2:

```
byte* b1 = b->getByte(2);
```

To get a value into the stream of bytes starting from the position 4 of dimension 2 (a word):

```
word w = b->getValue(4, 2);
```

Only dimension 1 and 2 are foreseen.

There are also methods the set the values:

```
bool setWord (unsigned start, word value)
void setByte (unsigned start, word value)
```

All information contained in this document is property of **INAF.** All rights reserved.

Figure 9: overall view of the packet

All information contained in this document is property of **INAF**. All rights reserved.

4. 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.readPacket();
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.

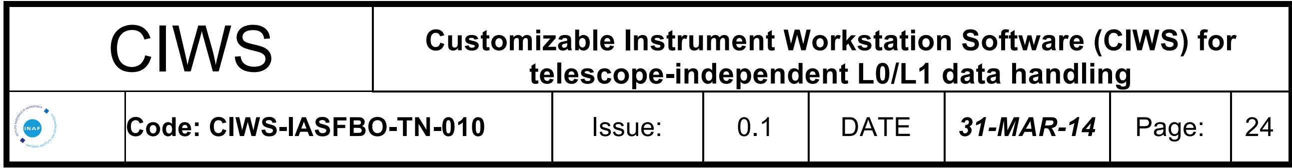
Summarizing, lines 2-4,6 provide the I/O Abstraction layer, while lines 1,5,7-10 are related to the Telemetry Management layer.

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

All information contained in this document is property of **INAF**. All rights reserved.



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.

All information contained in this document is property of **INAF.** All rights reserved.

5. Device examples

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

5.1 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
```

All information contained in this document is property of **INAF.** All rights reserved.

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*.

5.2 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.

5.2.1 SHM Server

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

All information contained in this document is property of **INAF**. All rights reserved.

```
#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) {
```

All information contained in this document is property of **INAF**. All rights reserved.

```
cout << e->geterror() << endl;
}

}
```

5.2.2 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));
```

All information contained in this document is property of **INAF.** All rights reserved.

```
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;
}

}
```

5.3 MSGQ

5.3.1 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[])
```

All information contained in this document is property of **INAF**. All rights reserved.

```
{
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;
}
}
```

5.4 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
```

All information contained in this document is property of **INAF.** All rights reserved.

```
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;
}
}
```

All information contained in this document is property of **INAF.** All rights reserved.

6. Output examples

6.1 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).

6.2 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
```

All information contained in this document is property of **INAF**. All rights reserved.

```
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:

All information contained in this document is property of **INAF**. All rights reserved.

```
//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;
```

All information contained in this document is property of **INAF.** All rights reserved.

7. Input examples

7.1 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).

7.2 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;
```

All information contained in this document is property of **INAF**. All rights reserved.

```
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();
}
```

7.2.1 InputFILE

With the following example it is possible to see how to read a list of packets into a file.

All information contained in this document is property of **INAF**. All rights reserved.

```
int i = 0;
while (! in->isEOF()) {
Packet* p = ips.readPacket();
if( in->isEOF())
break;
cout << i++ << " INPUT: " << p->getName() << endl;
//when you have finished to use the packet, it is important to
//deallocate the memory allocated during the reading operation from input
p->deleteExternalByteStream();
}
```

The internal if(in->isEOF()) is should be used because if we read the last packet, the pointer into the file is positioned in the last byte, not in the EOF.

Remember that if ips.readPacket() has reached the EOF a NULL pointer is returned. The end of cycle condition should be rewritten:

```
while (! in->isEOF()) {
Packet* p = ips.readPacket();
if( !p )
break;
}
```

An additional consideration about the memory management. When a packet is read some memory is allocated. When we have finished to use the packet, this memory should be deallocated with

```
p->deleteExternalByteStream();
```

7.3 Reading from a socket

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

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

All information contained in this document is property of **INAF**. All rights reserved.

```
//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;
```

7.4 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;
}
}
```

7.5 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:

All information contained in this document is property of **INAF.** All rights reserved.

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

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

7.6 InputPacketStreamFile example

The InputPacketStreamFile a a class that enable us to open and iterate into a telemetry file.

Follows an example:

```
//create the input packet stream
InputPacketStreamFile* ips = new InputPacketStreamFile();
//set the .stream and create the structure
ips->setFileNameConfig("CAL-DFE-TE.stream");
ips->createStreamStructure();

//open the telemetry file
ips->setFileNameStream("mcal.raw");
ips->setInitialPosition(0);
ips->openInputStream();
//create the list of packets recognized into the telemetry file. remember that a packet not recognized
//has the type 0
ips->freeRun();
```

Now is possible to access to the list of the packets recognized (and not recognized) into the file.

```
for(int index=0; index< ips->getNumberOfFileStreamPointer(); index++)
{
Packet *p = ips->getPacketFromFileStreamPointer(index);
//work with the packet
int apid = p->header->getFieldValue(3);
...
}
```

All information contained in this document is property of **INAF.** All rights reserved.

Remember that with the ips->getPacketFromFileStreamPointer(index) a pointer to the packet of the list of packets of the packet stream is obtained. If you want to obtain a local copy of the packet use the following method:

```
Packet *p = ips->getPacketFromFileStreamPointer(index, true);
```

But the performance of the PacketLib becomes slow (see the Chapter about the performance considerations).

7.6.1 NAVIGATION OF THE STREAM POINTER

The InputPacketStreamFile class store a list of all the loaded packets from a file. This list contains only a pointer into the file and some additional information. The pointer class is called FileStreamPointer, with the followings public attributes:

```
//## Type of packet. This is the index in the list of packetType
//## attribute of InputPacketStream class
int typeOfPacket;

//## Pointer of the start position into the file (FileStream)
long pointerStart;
//## A global index of this pointer into the full list of pointers
long index;
//## The name of the found packet
char* nameOfPacket;

//## Pointer of the end position into the file (FileStream)
long pointerEnd;
```

It is possible to access and work with this list of file pointers:

```
long nfsp = ips->getNumberOfFileStreamPointer();
for(long i=0; i<nfsp; i++) {
FileStreamPointer* fsp = ips->getFileStreamPointer(i);
if(fsp) {
cout << fsp->nameOfPacket << endl;
}
}
```

All information contained in this document is property of **INAF.** All rights reserved.

8. Performance considerations

The PacketLib is written with the main purpose the optimize the speed elaboration and the memory allocation. Do to the constraints, some consideration should be taken into account for writing PacketLib applications.

8.1 Memory management

Each Packet class has its own local copy of the byte stream read from input (the prefix and packet members) or generated to output (the prefix and packet_output members), represented by ByteStream objects.

As reported in the Par. 4.3.2, a Packet is compound of a lot of PartOfPacket objects. Each of these objects represent a part of the telemetry packet (the header, the blocks, and so on). Each part of packet contains a list of fields and two ByteStream objects that represents the raw representation of the part of packet, the members stream (for the input stream) and outputstream (for the output stream).

For memory optimization considerations, a unique byte stream for input (and one for output) is present in memory and it is shared by all the part of packets of the Packet object. In the following schema is showed this concept:

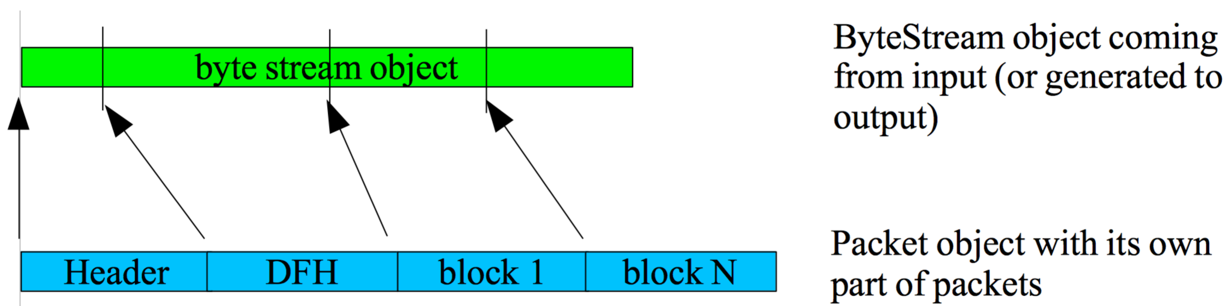


Figure 10: memory management

8.2 Multithreading considerations

PacketLib is not thread-safe, but there are some solutions to use the PacketLib in a like-trhead-safe mode, in addition to the mutexes/semaphore technique. The problem is that each Packet class has its own ByteStream (one for input, one for output) and if more than one thread access to the same Packet*, it is possible to write in the same area of the ByteStream. But there are some solutions:

1) create one PacketStream (input or output) for each thread (the PacketStream has its copy of the Packets, and share only the Input or Output objects between the thread (using a semaphore). In this way it is possible to work one the same Packet structure in parallel mode, and serialize only the read and write operations.

2) Use only one PacketStream but use more than one copy of the same .packet in the .stream. For example

```
[Packet Format]
```

All information contained in this document is property of **INAF**. All rights reserved.

```
ATM_FULLWINDOW.packet
ATM_FULLWINDOW.packet
```

and for each thread get a pointer of its own packet (index 1 for one thread, index 2 for the second thread) and put the mutex before a read and write operation.

8.3 Packet Stream management

Each packet stream object (input or output) has its own list of Packet objects generated from the .stream and .packet files. For example, if the .packet is the following:

```
[Configuration]

--prefix

true

--file format bigendian

true

--dimension of prefix

2

[Header Format]

headerESATM.header

[Packet Format]

TC_CAL_start.packet

TC_CAL_stop.packet

BURST.packet

GRID.01.packet

HK.packet
```

in the list of packets of the packet stream are contained 5 Packet objects, one for each type of packet.

All information contained in this document is property of **INAF.** All rights reserved.

To recognize a packet from the input (or generated to the output, but it follows only the considerations about the input, the output considerations are similar) the following streps are done:

1. open the input telemetry files
2. read a number of byte of the same dimension of the prefix (if present) and of the same dimension of the header of the first packet
3. decode the header and, from the header, reads the dimension of the packet
4. read a number of byte of the same dimension of the packet (without the header, that has been read in the above steps)
5. recognize (by means of the identificator specified in the .packet) and decode all the fields of the packet

The above operations are performed with the InputPacketStream::readPacket() and InputPacketStreamFile::getPacketFromFileStreamPointer().

The packet returned from these two methods are the pointer of the packet object contained in the list of packets of the packet stream.

The main reason of this choice is for performance considerations. In fact,

- the description of the structure of the telemetry packet in memory is expensive
- the storage of all the byte stream reads from input is expensive.

It seems that an application can work with only on packet of the same type at the same time. But also with this solution it is possible to get a list of packets of the same type in memory. It follows some solutions. These examples are reported for InputPacketStreamFile class. Similar solutions should be adopted for InputPacketStream base class.

8.3.1 Build a list of pointer

The InputPacketStream class contains the list of pointer of all the packets recognized into a file (after the freeRun() call).

It is possible to use the FileStreamPointer* InputPacketStreamFile::getFileStreamPointer(int index) to get the pointer of this list.

```
InputPacketStreamFile* ips = new InputPacketStreamFile();

ips->setFileNameConfig("CAL-DFE-TE.stream");

ips->createStreamStructure();

ips->setFileNameStream("mcal.raw");

ips->setInitialPosition(0);

ips->openInputStream();
```

All information contained in this document is property of **INAF**. All rights reserved.

```
ips->freeRun();


//build a list of file stream pointer

list<FileStreamPointer*> fileStreamList_1294;


for(int index=0; index< ips->getNumberOfFileStreamPointer(); index++)

{

//from the general list of packets of the file, build different

//lists for each type of packets

Packet *p = ips->getPacketFromFileStreamPointer(index);

int apid = p->header->getFieldValue(3);

if (apid == 1294) {

//make a copy of the byte stream

FileStreamPointer* fsp = ips->getFileStreamPointer(index);

fileStreamList_1294.push_back(fsp);

}

if (apid == ...) {

//build another list

}

}


// Iterate through list and output each element.

list<FileStreamPointer*>::const_iterator iter;

int i = 0;
```

All information contained in this document is property of **INAF.** All rights reserved.

```
for (iter=fileStreamList_1294 .begin(); iter !=
fileStreamList_1294.end(); iter++)

{

FileStreamPointer* fsp = (FileStreamPointer*)(*iter);



Packet *p = ips->getPacketFromFileStreamPointer(fsp->index);

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

}
```

The main advantages of this solution are the followings:

- no additional memory allocation to store the byte stream in memory read from input. The memory are allocated only when the ips object access to the file with the getPacketFromFileStreamPointer() method
- no additional memory allocation to store the structure of the packet. It is possible to use only one type of packet for all the telemetry coming from the input

But, on the other hand

- it is necessary to access th the file every time it is necessary to work with a packet object (when we call getPacketFromFileStreamPointer() method)

About multi threading application, it is also necessary to pass the ips object to each thread that receive the list of file stream pointers.

8.3.2 Build a list of Bytestream objects

The solution presented in this paragraph enable us to optimize the file access and the memory allocation.

```
InputPacketStreamFile* ips = new InputPacketStreamFile();

ips->setFileNameConfig("CAL-DFE-TE.stream");

ips->createStreamStructure();

ips->setFileNameStream("mcal.raw");

ips->setInitialPosition(0);
```

All information contained in this document is property of **INAF.** All rights reserved.

```
ips->openInputStream();
ips->freeRun();

//build a list of byte stream objects
list<ByteStream*> byteStreamPacketList_1294;

//from the general list of packets of the file, build different
//lists for each type of packets
for(int index=0; index< ips->getNumberOfFileStreamPointer(); index++)
{
Packet *p = ips->getPacketFromFileStreamPointer(index);
int apid = p->header->getFieldValue(3);
if (apid == 1294) {
//make a copy of the byte stream
ByteStream *b = new ByteStream(p->packet, 0, 0);
//store the byte stream in the list
byteStreamPacketList_1294.push_back(b);
}
if (apid == ...) {
//build another list
}
}

// Iterate through list and output each element.
list<ByteStream*>::const_iterator iter;
```

All information contained in this document is property of **INAF.** All rights reserved.

```
for (iter=byteStreamPacketList_1294.begin(); iter !=
byteStreamPacketList_1294.end(); iter++)

{

ByteStream* b = (ByteStream*)(*iter);



//get the correct type of packet

Packet* p = ips->getPacketType(3);

cout << p->getName() << " " << b->getDimension() << endl;

//set the packet object with the byte stream.

//If a prefix is present, get the prefix from input

//and call p->setAndVerifyPacketValue(prefix, b);

bool ok = p->setAndVerifyPacketValue(0, b);

if(ok) {

cout << p->header->getFieldValue(5) << endl; //get the source sequence
counter

}

}
```

The main advantages of this solution are the followings:

- no additional memory allocation to store the structure of the packet. It is possible to use only one type of packet for all the telemetry coming from the input
- it is not necessary to access to the file after the first freeRun()

On the other hand

- it is necessary to allocate memory for each byte stream read from file. The total amount of memory allocated is of the same dimension of the file.

About multithreading application, it is not necessary to pass the ips object to each thread, but it is possible for each thread to have its own packet stream object with a pointer to the correct type of packet.

All information contained in this document is property of **INAF.** All rights reserved.

At the time of writing, this is the best solution for the PacketLib architecture.

8.3.3 Build a list of packets objects

```
InputPacketStreamFile* ips = new InputPacketStreamFile();

ips->setFileNameConfig("CAL-DFE-TE.stream");

ips->createStreamStructure();

ips->setFileNameStream("mcal.raw");

ips->setInitialPosition(0);

ips->openInputStream();

ips->freeRun();


list<Packet*> packetList;

for(int index=0; index< ips->getNumberOfFileStreamPointer(); index++)

{

Packet *p = ips->getPacketFromFileStreamPointer(index, true);

int apid = p->header->getFieldValue(3);

if (apid == 1294) {

packetList.push_back( p );

}

if (apid == ...) {

//build another list

}

}

list<Packet*>::const_iterator iter;

// Iterate through list and output each element.
```

All information contained in this document is property of **INAF.** All rights reserved.

```
for (iter=packetList.begin(); iter != packetList.end(); iter++) {

Packet* p = (Packet*)(*iter);

cout << p->getName() << " " << b->getDimension() << endl;

}
```

This solution is the simple solution, but

- it is necessary to allocate memory for each byte stream read from file. The total amount of memory allocated is of the same dimension of the file.
- it is necessary to allocate memory to store the structure of the packet. This is a memory and time consumption operation.

All information contained in this document is property of **INAF.** All rights reserved.

9. Examples ABOUT LAYOUTS

9.1 Exception management

PacketLib can generate some exceptions. It is necessary to inserted the code into a try-catch block to manage these exception (when an exception is generated without this block, a segmentation fault should be obtained).

```
try {


//read and write from/to the I/O layers

//elaborate the packets

...


} catch (PacketException* e) {

cout << e->geterror() << endl;

}
```

9.2 Layout 2

In this example we build a Layout 2 packet. Use as reference the Layout 2 example of the PacketLib ICD. In this example it is explained the difference between a fixed and a variable number of block.

Let us consider the following extract from the .packet (see PacketLib ICD for the full text) (first case):

```
[SourceDataField]

block

fixed

--type of number of block: fixed = number of block fixed equals to max number of block (fixed | variable)

fixed
```

All information contained in this document is property of **INAF**. All rights reserved.

```
--supposed number (or max number) of block

6
```

In the above example we build a Layout 2 packet (the first 2 keyword are block and fixed) with a fixed number of blocks (the keyword 3): the number of blocks are 6.

Another case (second case) should be:

```
[SourceDataField]

block

fixed

--type of number of block: fixed = number of block fixed equals to max number of block (fixed | variable)

variable

--supposed number (or max number) of block

6
```

In the above example the number of blocks are variable. This means that it is possible to build a packet with 6 or less blocks.

From a programming point of view, the only difference between the two cases is the use of the method SourceDataField::setNumberOfRealDataBlock() with the number of blocks as argument. This is useful only in the second case.

In the following example is explained how to use the Layout 2 packets.

The first example is related with the generation of the telemetry.

```
int generate_packet() {

try {
```

All information contained in this document is property of **INAF**. All rights reserved.

```
Output* out;

OutputPacketStream ops;


//create the structure

ops.setFileNameConfig("conf_agile/stream.stream");

ops.createStreamStructure();


//parameter for output: file

out = (Output*) new OutputFile(ops.isBigEndian());

char** param = (char**) new (char*)[2];

param[0] = "test.raw"; //nome file

param[1] = 0;


//open the output

out->open(param);

ops.setOutput(out);


Packet* p = ops.getPacketType(2);

p->header->setFieldValue(3, 1294);

p->header->setFieldValue(5, 1);


//use the following command only if the number of blocks is not fixed

//p->dataField->setNumberOfRealDataBlock(10);


//set some fields
```

All information contained in this document is property of **INAF.** All rights reserved.

```
p->dataField->dataFieldHeader->setFieldValue_3_13(4, 3133393);

p->dataField->sourceDataField->setFieldValue_4_14(1, 0, 3133392);

//get the source data field

SDFBlockFixed* sdf = (SDFBlockFixed*) p->dataField->sourceDataField;


//get a block of the source data field

SDFBFBlock* block = sdf->getBlock(0);


//set some values of the block

for(int i = 0; i< block->getNumberOfFields(); i++)

block->setFieldValue(i, 1);

block->setFieldValue_5_1(0, 3.12);


//write the packet

ops.writePacket(p);


//close output

out->close();


} catch (PacketException* e) {

cout << e->geterror() << endl;

}

}
```

In the following example the packets are read.

All information contained in this document is property of **INAF**. All rights reserved.

```
int readpacket() {

Input* in;

try {

InputPacketStream ips;

ByteStream* bs;

bool endcycle = false;


//create the structure

ips.setFileNameConfig("conf_agile/stream.stream");

ips.createStreamStructure();

in = (Input*) new InputFile(ips.isBigEndian());

cout << "INPUT: FILE" << endl;

char** param = (char**) new (char*)[2];

param[0] = "test.raw";

param[1] = 0;


//open input

in->open(param);


ips.setInput(in);


//read input

int i = 0;

while (! in->isEOF()) {
```

All information contained in this document is property of **INAF**. All rights reserved.

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

if( in->isEOF())

break;

//check if the packet received is a particular type

if(p->getPacketID() == 2) {

cout << i++ << " INPUT: " << p->getName() << endl;

cout << "Num of blocks: " << p->dataField->getNumberOfRealDataBlock() <<
endl;



//get some values

cout << p->dataField->dataFieldHeader->getFieldValue_3_13(4) << endl;

cout << p->dataField->sourceDataField->getFieldValue_4_14(1, 0) << endl;

//get the source data field

SDFBlockFixed* sdf = (SDFBlockFixed*) p->dataField->sourceDataField;



//get a block

SDFBFBlock* block = sdf->getBlock(0);

cout << block->getFieldValue_5_1(0) << endl;

//print the list of values

char **c = p->dataField->sourceDataField->printValue();

int i = 0;

while(c[i])

cout << c[i++] << endl;

}
```

All information contained in this document is property of **INAF.** All rights reserved.

```
//when you have finished to use the packet, it is important to
//deallocate the memory allocated during the reading operation from input
p->deleteExternalByteStream();
}
in->close();
return 0;
}
```

9.3 Layout 3

In this example we build a Layout 3 packet. The ops is an object of OutputPacketStream type.

```
ByteStream* make_layout3(void* parameters) {
ByteStream* b;
static unsigned int sec = 0;
static unsigned short msec = 0;

OutputPacketStream* ops = param->ops;
Packet* p = ops->getPacketType(1);
//set the number of blocks at the data field level
p->dataField->setNumberOfRealDataBlock(number_of_block);

//make an iteration for each block of the packet. In this layout each
//block is compound by sub-blocks (called elements)
for(int i=0; i<p->dataField->getNumberOfRealDataBlock(); i++) {
```

All information contained in this document is property of **INAF.** All rights reserved.

```
//set the number of elements for each block. i is the number of the
block.

p->dataField->sourceDataField->setNumberOfRealElement(i,
number_of_elements);



//get the total number of fields of the source data field

word nfields = p->dataField->sourceDataField->getNumberOfFields(i);



//set a value for each field of the source data field

for(int j=0; j<nfields ; j++)

if(j!=2)

p->dataField->sourceDataField->setFieldValue(i, j, j);



p->header->setFieldValue(3, 1294);

p->dataField->dataFieldHeader->setFieldValue(2, 1);

p->dataField->dataFieldHeader->setFieldValue(3, 15);

//Time calculation

msec += 50;

if(msec >= 1000) {

msec -= 1000;

sec += 1;

}

short secM = (short) sec >> 16;

short secL = (short) sec;

p->dataField->dataFieldHeader->setFieldValue(4, secM);

p->dataField->dataFieldHeader->setFieldValue(5, secL);
```

All information contained in this document is property of **INAF**. All rights reserved.

```
p->dataField->dataFieldHeader->setFieldValue(6, msec);

}

burst_packet_number++;

if(burst_packet_number > max_burst_packet_number)

burst_packet_number = 0;

//SSC

p->header->setFieldValue(5, ssc++);

b = p->getOutputStream();

if(ops->thereIsPrefix()) {

if(ops->getPrefixDimension() == 2)

b->setWord(0, p->getDimension());

}

return b;

}
```

In the last example all the fields of the source data field should be seen as an unique list.

It is possible to work with a single block:

```
SDFBlockVariable* sdf = (SDFBlockVariable*) p->dataField->sourceDataField;

SDFBVBlock* block = sdf->getBlock(0); //get the block number 0 of the total number of blocks

for(int i = 0; i< block->getNumberOfFields(); i++)

block->setFieldValue(i, 1);
```

With sdf->getNumberOfRealDataBlock() it is possible to get the number of blocks.

It is also possible to access to each block and each element of the block:

All information contained in this document is property of **INAF.** All rights reserved.

```
//get the block number 1

SDFBVBlock* block1 = sdf->getBlock(1);

//get the fixed part of the block number 1

SDFBVBlockFixed* block1_fixed = block1->getFixedPart();


//set the number of elements of the block 1

block1_fixed->setNumberOfRealElement(3);

//it is possible to set (or get) a field using the setFieldValue

block1_fixed->setFieldValue(3, 111);

//access to a single element of the current block. Use the

//block1_fixed->getNumberOfRealElement() to get the number of elements
for each block

SDFBVBlockVariable* block1_el0 = block1->getElement(0);

block1_el0->setFieldValue(3, 150);
```

9.4 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;
```

All information contained in this document is property of **INAF**. All rights reserved.

```
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 ----------------------
```

All information contained in this document is property of **INAF.** All rights reserved.

```
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
```

All information contained in this document is property of **INAF**. All rights reserved.

```
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++);
```

All information contained in this document is property of **INAF.** All rights reserved.

```
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()) {
```

All information contained in this document is property of **INAF.** All rights reserved.

```
if(opstream->getPrefixDimension() == 2)

b->setWord(0, p->getDimension());

}

return b;

}
```

All information contained in this document is property of **INAF.** All rights reserved.

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.

All information contained in this document is property of **INAF**. All rights reserved.

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.

All information contained in this document is property of **INAF**. All rights reserved.

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.

All information contained in this document is property of **INAF.** All rights reserved.