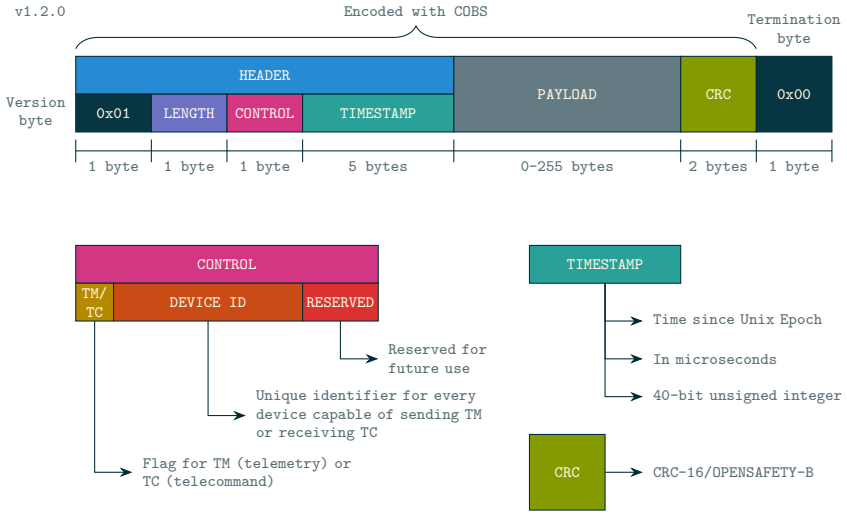
OrbiPacket Protocol Specification - v1.2.0

The OrbiSat Oeiras Team



Contents

1 Introduction **3**

2 Protocol Overview **3**

3 Packet Structure **4**

3.1 Header 4
3.1.1 Version 4
3.1.2 Length 4
3.1.3 Control 5
3.1.4 Timestamp 5
3.2 Payload 5
3.3 CRC 6
3.4 Termination Byte 6
3.5 Representation 6
3.5.1 Payload Storage 7
3.6 Packet Overhead 8

4 Encoding **8**
4.1 Stuffing 8
4.2 Implementation details 9

5 Decoding **9**
5.1 Single Packet Decoding 9
5.2 Framing 10
5.2.1 Stateless Framing 11
5.2.2 Stateful Framing 11

6 Synchronization **12**
6.1 Timestamp Synchronization 12

1 Introduction

OrbiPacket is a packet-based communication protocol. It was created specifically for communication with CanSat[1] devices. These devices can be fitted with very diverse electronics, and are often resource-constrained. Thus, this protocol has the goals of being simple and easy to use while providing decent efficiency and minimal overhead.

> **A bit of history.** The OrbiSat Oeiras team first participated in the CanSat Portugal competition in 2023/2024, being one of the finalists of the 11th edition. Several issues on all fronts meant no data was received during the CanSat's descent. The next year, wanting to get everything right, the team's firmware developer (aka me, Levi Gomes) decided sending data over text was neither efficient nor rigorous enough. So, after some effort, the first version of OrbiPacket was born during the 12th edition of the competition, in 2024/2025. And in case you're curious... no, we still didn't get any data.

2 Protocol Overview

The OrbiPacket protocol enables communication between a CanSat and one or more groundstations. This communication is carried out by exchanging **packets**, the basic units of the protocol. Packets are autonomous entities, which have a logical, high-level structure. These packets can be converted into raw bytes – a process known as **encoding** – adequate for transmission over a physical layer[2] (e.g. radio). At the other end of the link, the received bytes can then be **decoded** back into packets.

For communication to be possible, messages aren't enough – one also requires *communicators*. In OrbiPacket, these communicators are referred to as **devices**. A device is any discrete entity capable of sending and receiving packets. Note that OrbiPacket devices don't need to correspond to physical devices. In fact, a single piece of hardware can act as several devices, or, conversely, many components can come together under one device.

The protocol distinguishes between two kinds of packets: **telemetry** (TM) and **telecommand** (TC). The former represent data being sent *from* a device, whereas the latter encodes instructions sent *to* a device.

[1] CanSat's are small electronic devices, sized and shaped similarly to a soda can. They are used to perform various kinds of scientific missions, usually being released by rockets or other devices at varying altitudes. The *European Space Education Resource Office* (ESERO) holds CanSat competitions in various member states, challenging teams of secondary education students to build their own CanSat.

[2] In this specification, the designation *physical layer* is used very liberally, referring to any layers responsible for transmitting raw bytes. In that sense, the physical layer refers to the set of all layers below OrbiPacket – which might be a single layer or multiple layers.

As an example, consider a temperature sensor which is able to send readings and receive a command to alter the frequency of said readings. The readings will be wrapped in TM packets, while the command will be a TC packet. Since the sensor is able to send and receive packets, it is considered a device.

It is important to note that devices are more than a formalism. Each packet contains information about the device it relates to (i.e. the sender for TM packets and the receiver for TC packets), in the form of a **device identifier** (ID). The ID is a 5-bit number uniquely identifying a device in the CanSat.

3 Packet Structure

A packet is made up of several *fields*. Table 1 shows an overview of the packet structure. The fields before the payload are called the **header**, they contain metadata about the packet. When encoding a packet, fields must be concatenated in order. Fields which span multiple bytes are to be encoded in **little endian**, i.e., least-significant byte first.

Field	Size (bytes)
Version	1
Length	1
Control	1
Timestamp	5
Payload	0-255
CRC	2
Termination Byte	1

Table 1: Packet structure overview

3.1 Header

3.1.1 Version

SIZE 1 byte

Indicates the protocol version. Breaking updates to the protocol increment this field to ensure backwards compatibility. The current version of the protocol (1.2.0) specifies `0x01` as the version byte.

3.1.2 Length

SIZE 1 byte

Specifies the size of the payload in bytes, ranging from 0 to 255.

3.1.3 Control

SIZE 1 byte

Encodes metadata about the packet kind and device identifier. The following fields are specified from the highest bit to the lowest bit.

- **TM/TC Flag**: 1 bit, where `0` indicates telemetry and `1` indicates a telecommand.
- **Device ID**: 5 bits, uniquely identifying the source device for TM or target device for TC. Note that device IDs should remain consistent across TM and TC packets.

Reserved bits. The two remaining bits of the control byte are currently unused. Future versions of the protocol may use them to include additional information. They must therefore be ignored – do not use them to encode application specific data.

Interpreting the control byte. One can think of the control byte as *saying* the following about the packet:

- `0b0DDDDDRR`: TM packet from device `0bDDDDD` to the groundstation
- `0b1DDDDDRR`: TC packet from the groundstation to device `0bDDDDD`

3.1.4 Timestamp

SIZE 5 bytes

Represents the time since CanSat boot in microseconds as a 40-bit unsigned integer. The maximum representable timestamp is 1099511627776 microseconds, which corresponds to approximately 12.7 days – more than enough for typical CanSat operation times.

3.2 Payload

SIZE 0–255 bytes

Contains application-specific data, such as telemetry readings or command instructions. The structure of the payload is currently up to the application, but that may be subject to change in future versions.

3.3 CRC

SIZE 2 bytes

Cyclic redundancy check value computed over all preceding fields. The generator polynomial used by the protocol is CRC-16/OPENSAFETY-B, or `0xbaad` in Koopman's notation. The maximum packet length (excluding the CRC and termination bytes) is 255 + 11 = 266 bytes or 2128 bits. According to Koopman's research, this polynomial can protect 7985 bits at a Hamming distance of 4 and 108 bits (equivalent to 2 bytes of payload data) at a Hamming distance of 5, with negligible protected lengths for higher Hamming distances. This is deemed sufficient for this protocol. Table 2 lists the parameters for computing the CRC.

Parameter	Value
Width	`16`
Polynomial	`0x755b`
Initial Value	`0x0000`
Reflect Input	No
Reflect Output	No
Xor Output	`0x0000`
Check	`0x20fe`
Residue	`0x0000`

Table 2: CRC Parameters

3.4 Termination Byte

SIZE 1 byte

Inserted at the end of every packet to frame it, i.e, delimit it from adjacent packets. Fixed at `0x00`.

3.5 Representation

In their unencoded form, packets should be represented by a high-level construct. The specifics of this will invariably depend on the concrete language. Nonetheless, the recommended way of representing packets is through the use of a custom data type or object.

In the following sections, the pseudo-code data type defined in Listing 1 will be used. Note that the timestamp is stored as an unsigned 64-bit integer – when creating a packet from user data, it must be checked that the value doesn't surpass the 40-bit limit imposed by the protocol. The CRC field isn't included in the representation, since it is not part of the packet's information,

and should only exist in the encoded form. For simplicity, the length field isn't included, under the assumption that is can be retrieved from the array holding the payload. If such is not possible or practical due to language-specific constraints, it should be included in the packet representation.

Listing 1 Packet Representation

```
1: struct PACKET
2:     version : uint8
3:     kind : TM or TC
4:     deviceId : uint8
5:     timestamp : uint64
6:     payload : uint8[]
7: end struct
```

3.5.1 Payload Storage

A payload can contain arbitrary data – as long as it doesn't surpass the 255 byte limit. This data could assume many forms, and thus be of many different data types, which all have to somehow be stored by the same representation.

Perhaps the most obvious solution for many would be the use of a generic type (or a type template, or whichever other name such a construct might have). Thus the representation presented in Listing 1 would become something like Listing 2. While it might seem good at first, this solution presents two major drawbacks. Firstly, there are languages that don't support generics. Out of those that do, not all provide a mechanism to restrict the generic argument based on its size, so payloads over 255 bytes would only be detected when attempting to encode them. Furthermore, generics tend to *pollute* APIs – if packets were generic, any function or type interacting with them would also have to be generic. Because of these downsides, this solution is deemed unfitting.

Listing 2 A generic packet

```
1: struct PACKET<P>
2:     version : uint8
3:     kind : TM or TC
4:     deviceId : uint8
5:     timestamp : uint64
6:     payload : P
7: end struct
```

Another possibility would be to use a *universal* type[3] which can hold anything. Once again, this approach has the issue of not being able to

reject payloads larger than 255 bytes when they're created. On top of that, universal types are usually unsafe to handle, since all type information is lost.

The third option, and the one currently chosen by OrbiPacket, is to store the raw bytes of the payload in the packet representation, as a byte array. This way, any form of data can be stored – and by using fixed-size arrays, a payload larger than 255 bytes can't ever be created. In effect, this solution offsets the conversion of a payload to its bytes from the time of packet encoding to the time of packet creation. However, this doesn't mean the end-user should be responsible for performing the conversion – that would be potentially unsafe.[4] Instead, an implementation should provide an API to create payloads out of common data types.

3.6 Packet Overhead

The fixed fields (header and CRC) result in a total, unstuffed overhead of 10 bytes, so packet sizes range from 10 to 265 bytes, depending on payload length. COBS stuffing has a maximum overhead of 1 byte per 254 bytes of unstuffed data. Thus, accounting for the termination byte, the maximum overhead is 12 or 13 bytes per packet.

4 Encoding

Encoding a packet is the process of converting it into a stream of bytes. This stream of bytes is written into a buffer (byte array), either provided by the end-user or managed by the implementation.

Firstly, the packet's header fields are written to the buffer in order. The control byte is constructed from the packet kind and the device ID, using appropriate bitwise operations. The timestamp is written in little-endian, i.e., least-significant byte first, discarding the extra three bytes (required for a 64-bit integer but unused by the protocol). As previously discussed, payloads are stored as a byte array, which is simply copied to the buffer. Then, a CRC is computed over the buffer. The resulting 2 bytes are appended to the buffer, in little-endian. The buffer is then stuffed – see Section 4.1, and finally a termination byte is appended.

4.1 Stuffing

To prevent the occurrence of the termination byte (`0x00`) within the packet, which would lead to framing errors, the protocol employs **COBS (Con-**

[3] This is known by many names. In type theory, it's often referred to as *top* or *any*; object oriented languages usually call it *object*.

[4] For instance, an end-user could mistakenly provide the bytes in big-endian.

sistent Overhead Byte Stuffing). This stuffing is applied to all fields except the termination byte itself.

A full explanation of COBS is beyond the scope of this specification – if necessary, refer to the literature on the subject. In short, COBS works by replacing every null byte with the *distance* to the next null byte – this also includes prepending a byte indicating the distance to the first null byte. Furthermore, if the distance to the next null byte is larger than 255 (the maximum value of a single byte), then a distance of 255 will be indicated, and the 255^{th} byte will then indicate the distance to the next null byte (which can again be 255). Not only is this process remarkably simple, it also guarantees *consistent overhead*, i.e., the best-case, average-case and worst-case overheads are quite similar. The worst-case overhead is

$$\left\lceil \frac{N}{254} \right\rceil ,$$

where N is the length of the input.

4.2 Implementation details

Listing 3 presents an encoding procedure in pseudo-code, to serve as a reference for implementations. The procedure should be part of the public API of the implementation, taking in an instance of a packet (according to the representation presented in Section 3.5), and possibly a buffer. It should return a buffer (or a view into one) containing the encoded packet, possibly wrapped in a result type if any operations it performs are fallible. For implementations in object-oriented languages, it is recommended this method is a member of the representation object type.

5 Decoding

Decoding is the process converse to encoding. Decoding can be thought of as the operation of converting a stream of bytes into one or more valid packets. This encompasses three core steps: **separating** the stream into individual packets, **reconstructing** a packet from the raw bytes and **validating** it. The first step is to be handled separately from the other two, since it represents a logically separate task.

5.1 Single Packet Decoding

Decoding a single packet is a somewhat straightforward task. It mostly consists of performing the encoding process *backwards*. However, validity checks are interleaved, so that ill-formed packets are discarded.

Firstly, the incoming bytes must be unstuffed using COBS. Immediately, the version byte can be compared against the implementation's version.

Listing 3 Encoding Procedure

```
procedure ENCODE(packet : Packet)
    buffer ← []
    write packet → version to buffer
    write (length of packet → payload) to buffer
    control ← (packet → deviceId ≪ 2)
    if packet → kind is TC then
        control ← control | 0b10000000
    end if
    write control to buffer
    write (little-endian bytes of packet → timestamp) to buffer
    write packet → payload to buffer
    write (compute CRC of buffer) to buffer
    stuff buffer
    return buffer
end procedure
```

Then, the CRC checksum is reconstructed from the last two bytes and compared to the checksum computed over the remainder of the buffer. If they are different, the packet is considered invalid. The last item to be checked is the packet length. Since packets have 10 bytes of overhead – see Section 3.6 – it suffices to check that the buffer's second byte equals its length minus 10. To avoid integer underflow errors, implementations may first check that the packet is at least 10 bytes long. If either check fails, the packet is considered invalid. The last step is to parse the remaining packet fields. The third byte is analysed to retrieve both the device ID and the packet kind. The timestamp is reconstructed from bytes 4 through 8, and all remaining bytes constitute the payload. Listing 4 outlines this procedure.

Single packet decoding must be exposed through a top-level API – either as a standalone function which consumes a buffer and returns a packet or, in object oriented languages, as a constructor of the packet type. In either case, the operation is fallible.

5.2 Framing

To properly decode a complete stream, it is necessary to identify packet boundaries, i.e, to frame the packets. This is achieved by splitting the incoming byte stream at the termination byte, thus obtaining a set of byte sequences corresponding to single packets. There is, however, a catch: the last such sequence might be incomplete, in the event that not all bytes of the latest packet have yet been received (as exemplified in Fig. 1). These *trailing bytes* can only be decoded once the packet is complete, so they need to be temporarily stored. This storage can be controlled either by the end user

Listing 4 Single packet decoding procedure

```
procedure DECODE(buffer : uint8)
    unstuff buffer
    version ← buffer[0]
    if buffer[0] is not equal to protocol version then
        return invalid packet
    end if
    crc ← (get CRC from buffer)
    if crc is not equal to (compute CRC of buffer) then
        return invalid packet
    end if
    length ← buffer[1]
    if length is not equal to (length of buffer − 10) then
        return invalid packet
    end if
    kind ← (buffer[2] & 0b10000000)
    id ← (buffer[2] & 0b01111100) ≫ 2
    timestamp ← buffer[3 through 7]
    payload ← buffer[8 through length − 2]
    return Packet {version, kind, id, timestamp, payload}
end procedure
```

or by the implementation, resulting in the *stateless* and *stateful* methods described below. Note that both must internally invoke the single packet decoding routine. An implementation must provide both methods as part of its top-level API.

5.2.1 Stateless Framing

Stateless framing, as the name implies, doesn't keep track of any internal state: all decoding calls are independent and idempotent, and trailing bytes have to be managed by the end user. Thus, a decoding call receives a buffer, extracts all complete packets from it (regardless of their validity) and returns the trailing bytes. It is then up to the end user to prepend these trailing bytes to the received bytes before the next decoding call. Listing 5 outlines the control flow of this approach.

5.2.2 Stateful Framing

Stateful framing internally keeps track of trailing bytes, through a dedicated object. This object exposes a push-pop interface: it can be *fed* incoming bytes and *polled* for decoded packets, which it returns one at a time. Note that byte sequences are only processed when a packet is required. Listing 6 outlines the control flow of this approach. Clearly, using this approach is

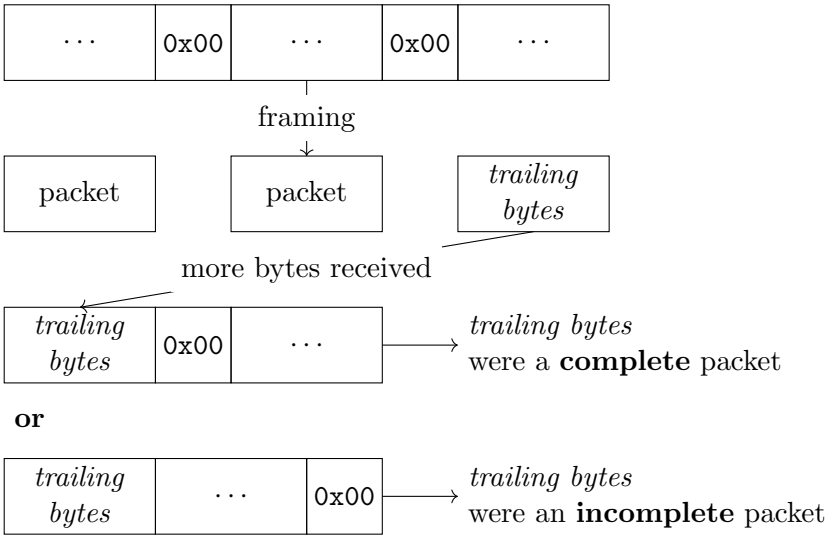


Figure 1: Trailing bytes

Listing 5 Control Flow of Stateless Framing

```
initialise buffer, trailing
while read incoming bytes into buffer do
    packets, trailing ← stateless decode (trailing + buffer)
    for each packet in packets do
        handle packet
    end for
end while
```

much simpler in terms of control flow. However, it takes memory control away from the end user, which might be undesirable.

6 Synchronization

6.1 Timestamp Synchronization

For accurate testing, logging and reporting, it is of utmost importance to know the absolute timestamp associated with a packet (e.g., in UTC). However, the microcontrollers typically used in CanSat devices are usually incapable of keeping track of such timestamps, so that only the time since boot (or program startup) is available. Thus, it becomes necessary to synchronize such timestamps to a real time clock, usually provided by the groundstation's system clock. As a result, the timestamps of all packets – both those sent by the CanSat and, for consistency and monotonicity, those sent by the groundstation – are measured in microseconds since CanSat startup, and a synchronization algorithm based on the Simple Network Time Pro-

Listing 6 Control Flow of Stateful Framing

```
initialise decoder
while read incoming bytes into decoder do
    while get packet from decoder do
        handle packet
    end while
end while
```

tocol (SNTP) is employed in the groundstation to recover absolute times from these timestamps. Figure 2 outlines the packet exchange that happens when a groundstation first connects to a CanSat.

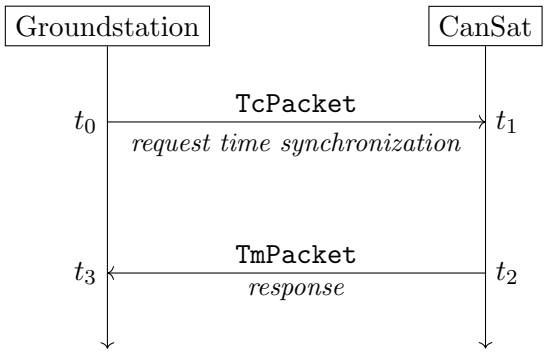


Figure 2: Time synchronization packet exchange.

Firstly, the groundstation sends a packet to the CanSat, with a payload containing the timestamp registered when constructing the packet, which is called the original timestamp (t_0). The CanSat receives the packet and registers its own timestamp – the receiving timestamp (t_1). Then it creates a response packet, in which it sends the original timestamp, the receiving timestamp and a transmitting timestamp (t_2), measured when creating the packet. Finally, the groundstation receives this response and registers its timestamp as the destination timestamp (t_3). The offset between the two clocks is calculated according to Eq. (1). For all further communications, the groundstation should add this offset to the timestamps it sends to the CanSat, and subtract it from the timestamps it receives.

$$\text{offset} = \frac{(t_1 - t_0) + (t_2 - t_3)}{2} \tag{1}$$