Parrot®

For Developers

ARSDK Protocols

Parrot SA

November 4, 2015

Contents

Purpose of this document

This document describes the binary protocols used by the ARSDK 3, and specifically by the `ARNetworkAL`, `ARNetwork`, `ARDiscovery`, `ARStream` and `ARCommands` libraries.

This document is made for people who want to implement an ARSDK-compatible framework, without reading all the ARSDK source code and trying to figure out "how it works"

Conventions

In this document, products are referred to by their network connection type, rather by their name, so the Bebop Drone and Jumping Sumo (and variants) are "Wifi products", while the Rolling Spider (and variants) is a "BLE product". If something is specific to a given product, its name will be written directly.

The connection is done between a controller (the computer, phone, tablet ...) and a device (the Bebop Drone, Rolling Spider ...). In the libraries, some objects are called "c2d" or "d2c", these notations means "controller to device" and "device to controller", respectively.

Inside the ARSDK, the products are sometimes called with other names than the actual product name. Here is a simple list of ARSDK alternative names for real names, which should help when reading the ARSDK source code:

Real Name	Alternatives
Bebop Drone	ARDrone, ARDrone 3
Jumping Sumo	JS
Rolling Spider	MiniDrone, RS
SkyController	SC
Airborne Night	MiniDroneEvoLight
Airborne Cargo	MiniDroneEvoBrick
Hydrofoil	MiniDroneEvoHydrofoil
Jumping Night	JSEvoLight
Jumping Race	JSEvoRace

Table 1: Name correspondence between product names and ARSDK internal names

Data endianness

For the `ARNetwork`, `ARNetworkAL`, `ARStream` and `ARCommands` library, all the data are sent on network in LITTLE_ENDIAN byte order. This is done because most of the devices actually are in LITTLE_ENDIAN mode, and thus avoids a lot of byteswapping.

The `ARSAL` library provides conversion macros (similar to the `htonl()`-family) in the
`<SDK>/libARSAL/Includes/libARSAL/ARSAL_Endianness.h` file.

1 ARDiscovery protocol

The `ARDiscovery` library is separated in two parts: Discovery, and Connection. The Discovery part is in charge of discovering the devices on the network (for Wifi products) or nearby (for BLE Products), while the Connection part is responsible for negociating the connection parameters (which are used by the `ARNetworkAL` library, among others)

1.1 Discovery

1.1.1 Wifi

The ARSDK Wifi Products use the mDNS protocol to advertise themselves on the wifi network. You can use any compatible implementation (Apple Bonjour, Avahi, Android NSDManager...) to discover the products.

The service types for the different products are:

Product Name	Service Type
Bebop Drone	`._arsdk-0901._udp`
Jumping Sumo	`._arsdk-0902._udp`
SkyController	`._arsdk-0903._udp`
Jumping Night	`._arsdk-0905._udp`
Jumping Race	`._arsdk-0906._udp`

Table 2: mDNS Service type for the Wifi Products

These identifiers can be found in the `<SDK>/libARDiscovery/Sources/ARDISCOVERY_Discovery.c` file.

You can find the following information from the mDNS Service:

Information	Source	Description
Name	Service Name	Display name of the product
IP	Service resolution	IP address of the product
Port	Service port	Discovery Connection port
Extras	Service TxtData	A json string containing extra informations (currently contains the product serial number)

Table 3: Informations available in the mDNS service

1.1.2 BLE

To identify ARSDK BLE Products, you need to get the product Advertising Data. You can use any BLE-Compliant implementation to do this (bluez, Apple CoreBluetooth, Android BluetoothAdapter...).

In the manufacturer specific data part of the advertising data, the first 6 bytes should be the following:

- Vendor ID (2 bytes) : `0x0043` (Parrot Bluetooth ID)
- USB Vendor ID (2 bytes) : `0x19cf` (Parrot USB ID)
- USB Product ID (2 bytes) : Depends on product

Product Name	USB Product ID
Rolling Spider	`0x0900`
Airborne Night	`0x0907`
Airborne Cargo	`0x0909`
Hydrofoil	`0x090a`

Table 4: USB Product IDs of BLE Products

These identifiers can be found in the `<SDK>/libARDiscovery/Sources/ARDISCOVERY_Discovery.c` file.

1.2 Connection

1.2.1 Wifi

For wifi products, further negociation is needed: this is done over a TCP socket, using the mDNS Service port.

To negociate the connection, connect a socket to this TCP port, and send a JSON string containing the following informations:

Key	Mandatory	Description
`d2c_port`	Yes	The UDP port you will use to read data
`controller_type`	Yes	The type of the controller (e.g. “Phone”, “Tablet”...)
`controller_name`	Yes	The name of the controller (e.g. Application name)
`device_id`	No	The product serial number

Table 5: Available keys for connection JSON

Here is an example of a valid connection JSON string:
{ `"d2c_port":43210, "controller_type":"Phone", "controller_name":"com.example.arsdkapp"` }

The ‘‘`device_id`’’ field is useful when reconnecting to a product: If a product receives a connection request with the ‘‘`device_id`’’ field, the connection will only be accepted if it matches the product serial number.

The device will answer a connection request with anoter JSON string sent on the same TCP socket. This response JSON string can contain the following information:

Key	Mandatory	Description
`status`	Yes	If different from 0, it means that the connection is refused (see below)
`c2d_port`	Yes	The UDP port you will send data to (If the connection is refused, this value will be 0)
`arstream_fragment_size`	No	The size of `ARStream` fragments
`arstream_fragment_maximum_number`	No	The maximum number of `ARStream` fragments per video frame.
`arstream_max_ack_interval`	No	The maximum time between ARStream ACKs
`c2d_update_port`	Yes	The FTP port for updating the product
`c2d_user_port`	No	Another FTP port for other uses
`skycontroller_version`	SC only	The SkyController version

Table 6: Available keys for connection answer JSON

Here is an example of a valid connection answer JSON string:

```
{ "status":0, "c2d_port":54321,
"arstream_fragment_size":65000,
"arstream_fragment_maximum_number":4,
"arstream_max_ack_interval":-1, "c2d_update_port":51,
"c2d_user_port":61 }
```

If the `status` field is not 0, then it is an `eARDISCOVERY_ERROR` enum value. These values can be found in the `<SDK>/libARDiscovery/Includes/libARDiscovery/ARDISCOVERY_Error.h` file.

1.2.2 BLE

There is no Connection part for BLE products. Just use your BLE adapter methods to connect to the device. Product pairing is not required for BLE connectivity.

2 ARNetworkAL protocol

The `ARNetworkAL` library is responsible for the network abstraction of the `ARNetwork` library. Both libraries are really tied together, and the structure of a network packet is composed of header from both, but for the sake of clarity, this section will describe the entire `ARNetworkAL+ARNetwork` headers.

Naming conventions: The full block of data sent by `ARNetworkAL` is called a `packet`. A `packet` is composed of one or more `frame`. Each `frame` is composed of an `header` (Described in this section), and some `data` (Described in either `ARCommand` or `ARStream` section).

2.1 Wifi

On Wifi network, a `packet` is sent on an UDP socket, and corresponds to an UDP packet. The destination ports were negotiated during the `ARDiscovery.Connection`, and the sending ports are free.

A `packet` can contain multiple `frames`. `frames` are simply added one after another in the UDP packet.

A `frame` contains the following information:

- Data type (1 byte)
- Target buffer ID (1 byte)
- Sequence number (1 byte)
- Total size of the `frame` (4 bytes, Little endian)
- Actual data (N bytes)

2.1.1 Data types

The `ARNetworkAL` library supports 4 types of data:

- Ack(1): Acknowledgment of previously received data
- Data(2): Normal data (no ack requested)

- Low latency data(3): Treated as normal data on the network, but are given higher priority internally
- Data with ack(4): Data requesting an ack. The receiver must send an ack for this data !

The full list of data types can be found in the `<SDK>/libARNetworkAL/Include/libARNetworkAL/ARNETWORKAL_Frame.h` file.

To acknowledge data, simply send back a `frame` with the Ack data type, a buffer ID of `128+Data_Buffer_ID`, and the data sequence number as the data.
E.g. : To acknowledge the `frame` `"(hex) 04 0b 42 0b000000 12345678"`, you will need to send a `frame` like `"(hex) 01 8b 01 08000000 42"`

2.1.2 Target Buffer IDs

Buffers IDs are separated into three main sections:

- [0; 9]: Reserved values for `ARNetwork` internal use.
- [10; 127]: Data buffers.
- [128; 255]: Acknowledge buffers.

By convention, Controller to Product buffers starts at 10 and grow up, while Product to Controller buffers starts at 127 and go down. A buffer number can be used in both direction.

A complete description of the buffers will be done in the `ARNetwork` section.

2.1.3 Sequence number

Each buffer has its own independant sequence number, which should be increased on new data send, but not on retries. The `ARNetwork` library will ignore out of order and duplicate data, but will still send Acks for them if requested. If the back-gap in sequence number is too high (we use 10 in `ARNetwork` library), then the `frame` is not considered out of order, and instead is accepted as the new reference sequence number.

2.1.4 Multiple `frames` in one `packet`

The `packet "(hex)01ba270800000042020bc30b00000012345678"` can be split in the follwing way:

- Read the type of the first `frame`: (0x01), or Ack
- Read the buffer id of the first `frame`: (0xba)
- Read the size of the first `frame`: `0x00000008` (written in human readable endian): 7 bytes of header + 1 byte of data
- Read the data of the first `frame`: `0x42`
- Since there is remaining data in the buffer, start again the process for a second frame

For this exmample, we have an ack for buffer `0x0a`, sequence number `0x42`, and non acknowelged data for buffer `0x0b`, with content `0x78563412`.

The `ARNetworkAL/ARNetwork` libraries should never send ill-formed packets (i.e. every packet you receive consists of [1..N] `frames`, without garbage data at the end. These libraries are ill-formed packets resistant: Before reading a `frame`, we make sure that there is at least 7 bytes of data remaining in the buffer, before reading data, we make sure that there is at least `data_size - 7` bytes of data in the buffer, and so on.

2.1.5 Disconnection detection

The `ARNetworkAL` library will consider the remote to be disconnected if no data was read from the socket for the last 5 seconds.

2.2 BLE

On BLE networks, a `packet` will only encapsulate a single `frame`. A `frame` is caracterized by the BLE characteristic address it belongs to.

A `frame` contains the following information:

- Data type (1 byte)
- Sequence number (1 byte)
- Actual data (up to 18 bytes)

2.2.1 Data types

The `ARNetworkAL` library supports 4 types of data:

- Ack(1): Acknowledgment of previously received data
- Data(2): Normal data (no ack requested)
- Low latency data(3): Treated as normal data on the network, but are given higher priority internally
- Data with ack(4): Data requesting an ack. The receiver must send an ack for this data unit!

The full list of data types can be found in the `<SDK>/libARNetworkAL/Include/libARNetworkAL/ARNETWORKAL_Frame.h` file.

To acknowledge a data unit, simply send back a `frame` with the Ack data type, to the characteristic of id `16+characteristic_id`, and the data sequence number as the data.
E.g. : To acknowledge the `frame "(hex) 04 42 12345678"` in characteristic `0xf00a`, you will need to send a `frame` like `"(hex) 01 01 42"` in characteristic `0xf01a`.

2.2.2 Characteristic IDs

The products should declare 32 characteristics for `ARNetwork`, plus 2 for ftp-like purpose (update, media download...).

The `ARNetwork` characteristics are numbered from `0xf000` to `0xf01f`. They will be referred as 0 to 31 (just add `0xf000` to get the real id).

Characteristics are separated into three sections:

- [0; 9]: Reserved for `ARNetwork` internal use.
- [10; 15]: Data.
- [16; 31]: Acknowledges.

By convention, the Controller to Product characteristics start a 10 and grow upward, while the Product to Controller characteristics start at 15 an

grow downward. A characteristic can be used for two-way communication, but that is not advised (and not used on products).

The two characteristics for ftp-like transmission (using `ARDataTransfer` library) are `0xfd23` and `0xfd53`.

2.2.3 Sequence number

Each characteristic has its own independent sequence number, which should be increased on new data send, but not on retries. The `ARNetwork` library will ignore out of order and duplicate data, but will still send Acks for them if requested. If the back-gap in sequence number is too high (we use 10 in `ARNetwork` library), then the `frame` is not considered out of order, and instead is accepted as the new reference sequence number.

3 ARNetwork protocol

Each product has its own `ARNetwork` configuration. This configuration defines the number, type and direction of multiple buffers. For BLE products, each buffer is mapped on a characteristic, for wifi product, the buffer ids are sent inside the `ARNetworkAL` packets.

3.1 Definition of a Buffer

From an external point of view, a buffer is a fifo which will be duplicated on the remote end. Each buffer has 8 parameters. Some of these parameters are for internal use only, while some have an effect on the binary data transmitted on network. The parameters are:

Parameter	Type	Description
`ID`	`int`	The ID of the buffer ([0-255] for wifi, [0-31] for BLE)
`dataType`	`eARNETWORKAL_FRAME_TYPE`	The type of the buffer This type will be sent in the `ARNetworkAL` frames.
`sendingWaitTimeMs` `ackTimeoutMs`	`int` `int`	Deprecated (use 0) Time (in milliseconds) before considering a frame lost (Only used for acknowledged buffers)
`numberOfRetry`	`int`	Number of retries before considering a frame lost (Only used for acknowledged buffers)
`numberOfCell`	`int32_t`	Size of the internal fifos
`dataCopyMaxSize`	`int32_t`	Maximum size of an element in the fifo
`isOverwriting`	`int`	Boolean value indicating what to do when data is received while the fifo is full

Table 7: Parameters for an `ARNetwork` buffer

The full configuration parameters for `ARNetwork` buffers can be found in `<SDK>/libARNetwork/Includes/libARNetwork/ARNETWORK_IOBufferParam.h`.

Each buffer is unidirectional (i.e. can be used to send or receive data, not both), but it is possible to have two buffers (one in each direction) sharing the same ID. Buffers are thus separated in the "sending" and "receiving" buffers list for each product.

On these parameters, only `ID` and `dataType` are actually sent on network. The other ones only dictate the `ARNetwork` internal behavior for these buffers. External implementations should follow this behavior too.

In reception buffers, the `dataCopyMaxSize` parameter indicates the maximum size of data (excluding `ARNetworkAL` header) that can be read by the library. If a buffer is defined with a `dataCopyMaxSize` of 128 in the product, then it will be unable to read a data of 129 or more bytes.

For emission buffers, the `isOverwriting` parameter dictates whether trying to send a data while the sending fifo is full will do a drop of an old data, or a refusal of the new data. For reception buffers, it has almost the same effects, except that refused acknowledged data are NOT acknowledged to the sender (so the sender might retry them later, when the buffer is no longer full).

For BLE Networks, the `ID` is converted to a characteristic number by adding `0xf000`.

3.2 Internal buffers

Regardless of the network type, the buffer with ID 0 to 9 are reserved for `ARNetwork` internal use. The current implementation uses buffers 0 and 1 to implement a ping with the following protocol:

- Send a `struct timespec` of the current time to buffer 0
- Wait for reception of the same `struct timespec` on buffer 1
- Calculate the difference to estimate the ping

For this to work, the remote end should immediately send the data on buffer 1 when something is received on buffer 0. If the remote does not implement it, then the estimated latency will always be 1sec, with no further effect on the product. (TL;DR : It is not mandatory for a custom implementation to do this)

This function is **disabled** on BLE networks, to avoid sending useless packets.

Buffers from 2 to 9 are reserved for future use.

3.3 Acknowledge buffers

Acknowledge buffers are automatically created by `ARNetwork` for each buffer (regardless of their type) with the following configuration:

```
{
.ID = baseId + 128; // (+16 for BLE)
.dataType = ARNETWORKAL_FRAME_TYPE_ACK;
.sendingWaitTimeMs = 0; // Deprecated
.ackTimeoutMs = 0; // Unused
```

```
.numberOfRetries = 0; // Unused
.numberOfCell = 1; // Never more than one ack at a time
.dataCopyMaxSize = 1; // One byte of data:  the sequence number
.isOverwriting = 0; // Useless by design:  there is only one ack
waiting at a time
}
```

Acknowledge buffers of sending buffers are added to the receiving buffers list, while acknowledge buffers for receiving buffers are added to the sending buffers list.

3.4 Per device buffers

3.4.1 Bebop Drone & SkyController

The Bebop Drone and the SkyController share the same buffer configuration:

Controller to Device buffers

- Non ack data (periodic commands for piloting and camera orientation).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 10
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 2; // PCMD + Camera
  .dataCopyMaxSize = 128;
  .isOverwriting = 1; // Periodic data:  most recent is better
  }
  ```
- Ack data (Events, settings ...).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 11
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_WITH_ACK;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 150;
  .numberOfRetries = 5;
  .numberOfCell = 20;
  ```

```
.dataCopyMaxSize = 128;
.isOverwriting = 0; // Events should not be dropped
}
```

- Emergency data (Emergency command only).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 12
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_WITH_ACK;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 150;
  .numberOfRetries = -1; // Infinite
  .numberOfCell = 1;
  .dataCopyMaxSize = 128;
  .isOverwriting = 0; // Events should not be dropped
  }
  ```

- ARStream video acks.
  This buffers transports `ARStream` data.
  ```
  {
  .ID = 13
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_LOW_LATENCY;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 1000; // Enough space ...
  .dataCopyMaxSize = 18; // Size of an ack packet
  .isOverwriting = 1; // New is always better!
  }
  ```

Device to Controller buffers

- Non ack data (periodic reports from the device).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 127
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 20;
  .dataCopyMaxSize = 128;
  .isOverwriting = 1; // Periodic data:  most recent is better
  }
  ```

- Ack data (Events, settings ...).
  This buffers transports `ARCommands`.
  {
  `.ID = 126`
  `.dataType = ARNETWORKAL_FRAME_TYPE_DATA_WITH_ACK;`
  `.sendingWaitTimeMs = 0; // Deprecated`
  `.ackTimeoutMs = 150;`
  `.numberOfRetries = 5;`
  `.numberOfCell = 256;`
  `.dataCopyMaxSize = 128;`
  `.isOverwriting = 0; // Events should not be dropped`
  }

- ARStream video data.
  This buffers transports `ARStream` data.
  {
  `.ID = 125`
  `.dataType = ARNETWORKAL_FRAME_TYPE_DATA_LOW_LATENCY;`
  `.sendingWaitTimeMs = 0; // Deprecated`
  `.ackTimeoutMs = 0; // Unused`
  `.numberOfRetries = 0; // Unused`
  `.numberOfCell = <arstream_fragment_maximum_number>*2;`
  `// Read from ARDiscovery.Discovery part !`
  `.dataCopyMaxSize = <arstream_fragment_size>;`
  `// Read from ARDiscovery.Discovery part !`
  `.isOverwriting = 1; // New is always better!`
  }

3.4.2 Jumping Sumo, Jumping Night & Jumping Race

The Jumping Sumo, the Jumping Night and the Jumping Race share the same buffer configuration:

Controller to Device buffers

- Non ack data (periodic commands for piloting).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 10
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 1;
  .dataCopyMaxSize = 128;
  .isOverwriting = 1; // Periodic data:  most recent is better
  }
  ```
- Ack data (Events, settings ...).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 11
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_WITH_ACK;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 150;
  .numberOfRetries = 5;
  .numberOfCell = 20;
  .dataCopyMaxSize = 128;
  .isOverwriting = 0; // Events should not be dropped
  }
  ```
- ARStream video acks.
  This buffers transports `ARStream` data.
  ```
  {
  .ID = 13
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_LOW_LATENCY;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 1000; // Enough space ...
  .dataCopyMaxSize = 18; // Size of an ack packet
  ```

```
.isOverwriting = 1; // New is always better!
}
```

- ARStream audio data. (Only for newer models)
  This buffers transports `ARStream` data.
  ```
  {
  .ID = 15
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_LOW_LATENCY;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = <arstream_fragment_maximum_number>*2;
  // Read from ARDiscovery.Discovery part !
  .dataCopyMaxSize = <arstream_fragment_size>;
  // Read from ARDiscovery.Discovery part !
  .isOverwriting = 1; // New is always better!
  }
  ```

- ARStream audio acks. (Only for newer models)
  This buffers transports `ARStream` data.
  ```
  {
  .ID = 14
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_LOW_LATENCY;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 1000; // Enough space ...
  .dataCopyMaxSize = 18; // Size of an ack packet
  .isOverwriting = 1; // New is always better!
  }
  ```

Device to Controller buffers

- Non ack data (periodic reports from the device).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 127
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 20;
  .dataCopyMaxSize = 128;
  ```

```
.isOverwriting = 1; // Periodic data:  most recent is better
}
```

- Ack data (Events, settings ...).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 126
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_WITH_ACK;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 150;
  .numberOfRetries = 5;
  .numberOfCell = 256;
  .dataCopyMaxSize = 128;
  .isOverwriting = 0; // Events should not be dropped
  }
  ```

- ARStream video data.
  This buffers transports `ARStream` data.
  ```
  {
  .ID = 125
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_LOW_LATENCY;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = <arstream_fragment_maximum_number>*2;
  // Read from ARDiscovery.Discovery part !
  .dataCopyMaxSize = <arstream_fragment_size>;
  // Read from ARDiscovery.Discovery part !
  .isOverwriting = 1; // New is always better!
  }
  ```

- ARStream audio acks. (Only for newer models)
  This buffers transports `ARStream` data.
  ```
  {
  .ID = 123
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_LOW_LATENCY;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 1000; // Enough space ...
  .dataCopyMaxSize = 18; // Size of an ack packet
  .isOverwriting = 1; // New is always better!
  }
  ```

- ARStream audio data. (Only for newer models)
  This buffers transports `ARStream` data.
  {
  `.ID = 124`
  `.dataType = ARNETWORKAL_FRAME_TYPE_DATA_LOW_LATENCY;`
  `.sendingWaitTimeMs = 0; // Deprecated`
  `.ackTimeoutMs = 0; // Unused`
  `.numberOfRetries = 0; // Unused`
  `.numberOfCell = <arstream_fragment_maximum_number>*2;`
  `// Read from ARDiscovery.Discovery part !`
  `.dataCopyMaxSize = <arstream_fragment_size>;`
  `// Read from ARDiscovery.Discovery part !`
  `.isOverwriting = 1; // New is always better!`
  }

3.4.3 Rolling Spider, Hydrofoil, Airborne Night & Airborne Cargo

The Rolling Spider, the Hydrofoil, the Airborne Night and the Airborne Cargo share the same buffer configuration:

Controller to Device buffers

- Non ack data (periodic commands for piloting).
  This buffers transports `ARCommands`.
  {
  `.ID = 10`
  `.dataType = ARNETWORKAL_FRAME_TYPE_DATA;`
  `.sendingWaitTimeMs = 0; // Deprecated`
  `.ackTimeoutMs = 0; // Unused`
  `.numberOfRetries = 0; // Unused`
  `.numberOfCell = 1;`
  `.dataCopyMaxSize = 18; // Maximum size of data on a ble characteristic with a 2 byte header`
  `.isOverwriting = 1; // Periodic data:  most recent is better`
  }

- Ack data (Events, settings ...).
  This buffers transports `ARCommands`.
  {
  `.ID = 11`
  `.dataType = ARNETWORKAL_FRAME_TYPE_DATA_WITH_ACK;`
  `.sendingWaitTimeMs = 0; // Deprecated`
  `.ackTimeoutMs = 150;`
  `.numberOfRetries = 5;`
  `.numberOfCell = 20;`
  `.dataCopyMaxSize = 18;`
  `.isOverwriting = 0; // Events should not be dropped`
  }

- Emergency data (Emergency command only).
  This buffers transports `ARCommands`.
  {
  `.ID = 12`
  `.dataType = ARNETWORKAL_FRAME_TYPE_DATA_WITH_ACK;`
  `.sendingWaitTimeMs = 0; // Deprecated`
  `.ackTimeoutMs = 150;`
  `.numberOfRetries = -1; // Infinite`
  `.numberOfCell = 1;`

```
.dataCopyMaxSize = 18;
.isOverwriting = 0; // Events should not be dropped
}
```

Device to Controller buffers

- Non ack data (periodic reports from the device).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 15
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 0; // Unused
  .numberOfRetries = 0; // Unused
  .numberOfCell = 20;
  .dataCopyMaxSize = 18;
  .isOverwriting = 1; // Periodic data:  most recent is better
  }
  ```
- Ack data (Events, settings ...).
  This buffers transports `ARCommands`.
  ```
  {
  .ID = 14
  .dataType = ARNETWORKAL_FRAME_TYPE_DATA_WITH_ACK;
  .sendingWaitTimeMs = 0; // Deprecated
  .ackTimeoutMs = 150;
  .numberOfRetries = 5;
  .numberOfCell = 20;
  .dataCopyMaxSize = 18;
  .isOverwriting = 0; // Events should not be dropped
  }
  ```

4 ARCommands protocol

The `ARCommand` library is a simple codec for sending binary data on the network.

4.1 Command identifier

The command is identified by its first 4 bytes:

- Project or Feature ID (1 byte)
- Class ID in the project/feature (1 byte)
- Command ID in the class (2 bytes)

In this document, commands are noted in the `Project.Class.Command` notation. For example, the `PCMD` command, in class `Piloting` for project `ARDrone3` is `ARDrone3.Piloting.PCMD`.

4.2 Command arguments

The command arguments are directly packed after the command ID. Multi-bytes values are sent in little endian to avoid bytes swap in the product.

Here is a list of the supported argument types:

4.3 Commands definition

Commands are defined in xml files. All the source code of the `ARCommands` library is generated from these files.

The current xml files can be found in the `<SDK>/libARCommands/Xml` directory.

In this directory, each `.xml` file correspond to a project or a feature. Each product only understand a given list of features. Here is a list of features implemented per product:

Note that implementing a features means that at least a subset of the feature is useful for the product, not that all commands in the feature are actually implemented!

Type	Size (bytes)	Description
u8	1	unsigned 8bit value
i8	1	signed 8bit value
u16	2	unsigned 16bit value
i16	2	signed 16bit value
u32	4	unsigned 32bit value
i32	4	signed 32bit value
u64	8	unsigned 64bit value
i64	8	signed 64bit value
float	4	IEEE-754 single precision
double	8	IEEE-754 double precision
string	*	Null terminated string (C-String) (Variable size)
enum	4	Per command defined enum Coded as i32 on network

Table 8: Supported types for `ARCommand` arguments

Product	Features
Bebop Drone	ARDrone3, Common
Jumping Sumo	JumpingSumo, Common
Rolling Spider	MiniDrone, Common
SkyController	SkyController
Airborne Night	MiniDrone, Common
Airborne Cargo	MiniDrone, Common
Hydrofoil	MiniDrone, Common
Jumping Night	JumpingSumo, Common
Jumping Race	JumpingSumo, Common

Table 9: Features implemented by each product

Also note that the `xxx_debug.xml` files contain debug commands that should be avoided. These commands can change from one version to another without notice, and can have unwanted behavior.

4.3.1 The SkyController case

The SkyController implements its own set of commands, even for common ones, as it can be connected to another device. When a SkyController is connected to a Bebop Drone, it will forward all `ARDrone3.X.Y` and `Common.Z.W` commands to the Bebop Drone, and will forward to the controller all the data coming from the Bebop (including the `ARStream` data.

4.4 Command attributes

While not tied to the `ARCommand` codec, certain commands can have other xml attributes, namely `buffer`, `timeout` and `listtype`. These attributes are given as hints for an implementer on how the command is intended to be used.

4.4.1 buffer

The value of this attribute can be either `NON_ACK`, `ACK` or `HIGH_PRIO`, defaulting to `ACK` if not given. It gives a hint about the destination buffer for the command.

For the Bebop Drone, the `NON_ACK` buffers are 10 (c2d) and 127 (d2c), the `ACK` buffers are 11 (c2d) and 126 (d2c), and the `HIGH_PRIO` buffer is the 12 (c2d).

This is only a hint, and the product will decode any `ARCommand` on any `ARNetwork` buffer, as long as the buffer is not used for `ARStream`.

4.4.2 timeout

The value of this attribute can be either `POP`, `RETRY` or `FLUSH`, defaulting to `POP` if not given.

For acknowledged data, if a timeout happens (after the retries from `ARNetwork`), there is three possible answers:

- POP: Pop the data from the fifo, and continue with the next data (default).
- RETRY: Retry the data. This leads to infinite retries of the current data.
- FLUSH: Flush the entire `ARNetwork` buffer. This can be useful if the next data depends on the current one.

This information has no effect on non acknowledged data.

4.4.3 listtype

The value of this attribute can be either `NONE`, `LIST` or `MAP`, defaulting to `NONE` if not given.

`LIST` commands are sent multiple times, and a list of all the received value should be created. The `ARController` library uses a hash map with a locally generated integer key to emulate this.

`MAP` commands are sent multiple times, and their first argument should be used as a key in a map of received values.

In both cases, clearing the map/list before requesting the data is the responsibility of the receiver, unless stated otherwise in a specific command implementation.

4.5 Commands list

Listing all the commands in this document would be too long. The format used in the `.xml` files is designed to be human readable, with inline comments about every commands and arguments.

Some of the important commands are listed below.

4.5.1 `Common.Settings.AllSettings` & `Common.Common.AllStates`

Normally, the product will send state and settings updates on the go. To synchronize them, you will have to request a full snapshot of the settings and the states of the product during the initialization.

When receiving these commands, the product will send all its settings (or states), then it will send a final command, saying that all the settings or state were sent.

The SkyController uses the `SkyController.Settings.AllSettings` and `SkyController.Common.AllStates` commands instead.

Request	Final answer
Common.Settings.AllSettings	Common.SettingsState.AllSettingsChanged
Common.Common.AllStates	Common.CommonState.AllStatesChanged
SkyController.Settings.AllSettings	SkyController.SettingsState.AllSettingsChanged
SkyController.Common.AllStates	SkyController.CommonState.AllStatesChanged

Table 10: Final command for each Settings/State request

4.5.2 Common.Common.CurrentDate & Common.Common.CurrentTime

These commands set the date/time on the product, which in turn is set into the media and PUD (for ARDrone Academy files.

The argument to these commands is an ISO-8601 formatted string, with the following formatters:

- "yyyy-MM-dd" for Common.Common.CurrentDate. Ex: 2015-08-27.
- "'T'HHmmssZZZ" for Common.Common.CurrentTime. Ex: T101527+0200.

For compatibility purposes, you should always sent both commands to the product, not juste the Common.Common.CurrentDate one. The order is irrelevant on newer firmwares, but older ones need the Common.Common.CurrentTime to be sent after.

Note that you should generate both strings from a single timestamp to avoid any loop error at midnight.

4.5.3 ARDrone3.MediaStreaming.VideoEnable & JumpingSumo.MediaStreaming.VideoEnable

These commands will enable or disable the video streaming from the product. When connecting, the streaming is disabled (which lets a high bandwidth to transfer all the data needed by the Common.Settings.AllSettings and Common.Common.AllStates commands). You should enable the video only when needed (when you actually display it !).

In FreeFlight, the video is also disabled in the following cases:

- When the user browses the internal memory of the drone.
- When the user downloads media from the drone.

- When the user sends an update file to the drone.

4.6 Using the ARCommandsParser to generate your own code

The code generator is split into a parser, and a generator part. The parser is written in Python, and can be found in the `ARSDKBuildUtils` repository: `<SDK>/ARSDKBuildUtils/Utils/Python/ARCommandsParser.py`

It defines a `parseAllProjects` function, which takes 4 arguments:

- projects: A list of strings, list of the projects to parse. If this list contains the "all" string, then all projects are parsed, regardless of the content of the other elements.
- pathToARCommands: The path to the `ARCommands` library root (not the `Xml` directory !)
- genDebug: if True, then the `xxx_debug.xml` files are also parsed. Defaults to False. Should not be used
- mergeDebugProjectsInReleaseProjetcs: if True, then the debug commands will be merged with the release commands in single projects instead of being in separated debug projets. Defaults to False. Should not be used

And returns a list of `ARProject` objects. This class (and all other internally used class) are fairly straightforward to understand, and thus are not described in depth here.

A simple example of iterating on all commands can be seen at lines 519-554 of `<SDK>/libARCommands/Xml/generateLibARCommands.py`

5 ARStream protocol

The `ARStream` library is designed to send and receive arbitraty binary streams using `ARNetwork` as its network back-end. It is used to transport live audio and video data between the product and the controller.

The `ARStream` library used an acknowledge system on Bebop Drone firmwares before 2.0.17, while the Jumping Sumo never used them. This feature won't be used on newer firmwares (for all products), so implementing it is optionnal for an `ARStream` compatible library.

The `ARStream` library is not designed to be used on BLE networks.

5.1 Terminology

`ARStream` was designed to transport video data, thus the terms used in this documentation will match some video terms.

The input of `ARStream` is a `frame`. A `frame` can be a single video frame (for video streams), or multiple audio samples joined togeter (for audio streams). Each frame is also tagged as an `FLUSH_FRAME` or not.

`FLUSH_FRAMES` are mapped on I-Frames for `h.264` stream (e.g. on Bebop Drone). It is always set for `MJPEG` (e.g. Jumping Sumo) and audio streams, as these do not have the I-Frame/P-Frame difference.

5.2 Data

When sending a `frame`, it may be divided in multiple `fragments` on the network. The size and the maximum number of fragments are negociated during the ARDiscovery Connection part.

The data sent to `ARNetwork` consists of a packed header followed by the `fragment`.

The header format is:

- `frameNumber` (2 bytes): Identifier of the frame, if two fragments have the same frame number, they belong to the same frame. This counter loops every 2^{16} frames.

- `frameFlags` (1 byte): A bitfield indicating flags for the frame. Currenlty only support the `FLUSH_FRAME` flag in bit 0.
- `fragmentNumber` (1 byte): The index of the current fragment in the frame.
- `fragmentsPerFrame` (1 byte): The total number of fragments in the frame.

The size of the data is not specified, as it can be derived from the size of the `ARNetwork` data. The offset is not specified as the library consider that the N^{th} fragment should go at index $arstream\_fragment\_size * N$ (i.e. the library uses only full size fragments, except for the last one which can be shorter).

Fragments can be received in any order, or even multiple time each. All implementations should accept these cases and keep working. (i.e. do not append data to the output buffer without reading the header!)

A frame is considered as complete once we receive all of its fragments. If we receive a fragment for frame $N + 1$ while the frame N is not complete, then we discard the N^{th} frame and start working on the new one.

`FLUSH_FRAME`s are frames that can be immediately decoded. To reduce latency, `FLUSH_FRAME`s should flush any other waiting frame from the pipeline (including any fifo between `ARStream` and a video decoder). This behavior is also implemented on the `ARStream` sender on the product: if a new `FLUSH_FRAME` is available, but some previous frame were not sent, these frames are flushed and the new one is sent instead.

5.3 Acknowledges

`ARStream` acks are deprecated and are no longer enabled on most products. The description here is only for compatibility with older Bebop Drone firmwares.

`ARStream` ack policy is controlled by the `arstream_max_ack_interval` parameter, negociated during the `ARDiscovery` Connection. Most product will send a value of -1, which means that no acks are to be sent. Possible values are:

- -1 (or other negative value): No ack at all
- 0: No periodic acks, but send ack when receiving a `fragment`

- > 0: Maximum time (in ms) between two acks. Typically this is done by having a thread sending periodic acks. To keep ack latency as low as possible, keep also sending acks when receiving a `fragment`.

The Acknowledge packet has the following format (packed in memory, so no gap inside the structure):

- `frameNumber` (2 bytes): Id of the frame (must match the current sending frame).
- `highPacketsAck` (8 bytes): Bitfield for the upper 64 fragments of the frame.
- `lowPacketsAck` (8 bytes): Bitfield for the lower 64 fragments of the frame.

To acknowledge a fragment, the corresponding byte in the 128bits bitfield is set, when all fragments bits are set, then the full frame is acknowledged. This structure limits the maximum number of fragments to 128, but the current frimwares only use 4 at most.

When sending a full ack, it may be useful to set all the bits (even the ones for unused fragment numbers) in the bitfield. This allow the `ARStream` late-ack algorithm to work.