Details on the Sphero API

Protocol Definition

All Sphero API packets share a single structure defined below. Packets start/end with the special SOP/EOP control bytes. These control byte values are not allowed unencoded anywhere else in the packet (see: Packet Encoding). Packets are classified as either commands or responses, distinguished by a bit flag. Commands may be sent at any time by any node. Commands may optionally request a response from the receiver using a bit flag.

Packets can be directed to particular nodes using the “target ID” (TID) and “source ID” (SID). The TID and SID are each composed of two parts, a “port ID” and a “node ID” (See: Sending Information via Channels).

All packets have a two-byte ID code, separated into the “device ID” (DID) and “command ID” (CID). These bytes specify the command to execute. When sending a command, the sender uses the “sequence number” (SEQ) as a handle to link a particular command packet to its response. Responses echo the DID, CID, and SEQ to help the sender fully identify the corresponding command packet.

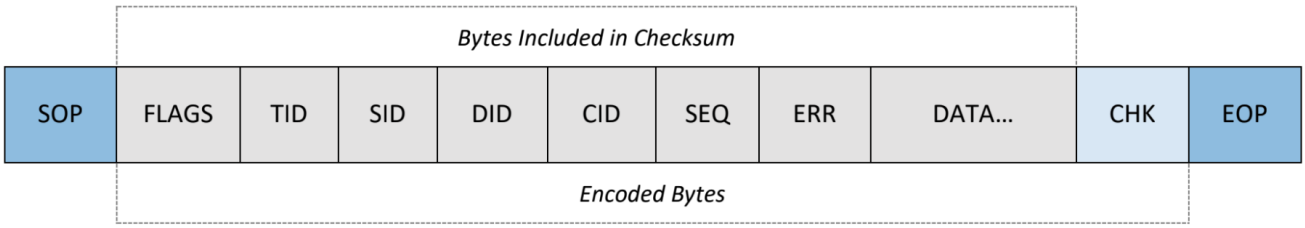
Responses contain a special “error code” field in the header that expresses the result of the corresponding command (see: Error Codes).

Both commands and responses may contain a data payload. In principle, this payload may be any size, though in practice the payload is limited by the receiver’s buffers. For forwarded commands, the data size may also be limited by intermediate nodes’ buffer sizes, depending on the implementation.

Finally, all packets contain a single byte checksum, which is computed over all other bytes in the packet (excluding SOP and EOP) before payload encoding.

Packet Structure

The Sphero API packet structure is shown below. All items are a single byte, except DATA, which is zero or more bytes.



SOP	Start of Packet	Control byte identifying the start of the packet
FLAGS	Packet Flags	Bit-flags that modify the behavior of the packet
TID	Target ID	Address of the target, expressed as a port ID (upper nibble) and a node ID (lower nibble). (Optional)
SID	Source ID	Address of the source, expressed as a port ID (upper nibble) and a node ID (lower nibble). (Optional)
DID	Device ID	The command group ("virtual device") of the command being sent
CID	Command ID	The command to execute
SEQ	Sequence Number	The token used to link commands with responses
ERR	Error Code	Command error code of the response packet (optional)
DATA...	Message Data	Zero or more bytes of message data
CHK	Checksum	The sum of all bytes (excluding SOP & EOP) mod 256, bit-inverted
EOP	End of Packet	Control byte identifying the end of the packet

Packet Encoding

To avoid misinterpretations of the packet structure, the SOP and EOP bytes are never allowed in the rest of the packet. If these byte values are necessary in the payload, they are encoded before transmission and decoded by the receiving parser. The encoding method is an extension of SLIP encoding, using a two-byte escape sequence to replace special characters. This method necessitates a third special character, “escape” (ESC). The three special characters are encoded by prepending the ESC character and changing the original values to corresponding “escaped” values. These six values are shown in the table below.

Abbreviation	Description	Value
ESC	Escape	0xAB
SOP	Start of Packet	0x8D
EOP	End of Packet	0xD8
ESC_ESC	Escaped "Escape"	Ox23
ESC_SOP	Escaped "Start of Packet"	0x05
ESC_EOP	Escaped "End of Packet"	0x50

When the ESC, SOP, or EOP bytes are needed in the payload, they are encoded into (or decoded from) two-byte escape sequences as follows:

Encoding	Decoding
ESC → ESC, ESC_ESC SOP → ESC, ESC_SOP EOP → ESC, ESC_EOP	ESC, ESC_ESC → ESC ESC, ESC_SOP → SOP ESC, ESC_EOP → EOP

The values for ESC, SOP, and EOP were specifically selected to minimize the cost of encoding. Additionally, these values were selected to provide a simple relationship between control values and their corresponding escaped value (unencoded value = escaped value | 0x88).

Flags

API flags are used to modify the behavior of the API system. The available API flags are listed below. Bits are listed 0 to 7, where 0 is the least significant bit.

Bit	Flag	Bit = 1	Bit = 0
0	Packet is a Response	Packet is a response. This implies that the packet has the error code byte in the header.	Packet is a command.
1	Request Response	Request response to a command (only valid if the packet is a command).	Do not request any response.
2	Request Only Error Response	Request response only if command results in an error (only valid if packet is a command and "Request Response" flag is set).	Do not request only error responses.
3	Packet is Activity	This packet counts as activity. Reset receiver's inactivity timeout.	Do not reset receiver's inactivity timeout.
4	Packet has Target ID	Packet has Target ID byte in header.	Packet has no specified target.
5	Packet has Source ID	Packet has Source ID byte in header.	Packet has no specified source.
6	Currently Unused	n/a	n/a
7	Extended Flags	The next header byte is extended flags.	This is the last flags byte.

Error Codes

Success	0x00	Command executed successfully
Bad Device ID	0x01	Device ID is invalid (or is invisible with current permissions)
Bad Command ID	0x02	Command ID is invalid (or is invisible with current permissions)
Not Yet Implemented	0x03	Command is not yet implemented or has a null handler
Command is Restricted	0x04	Command cannot be executed in the current state or mode
Bad Data Length	0x05	Payload data length is invalid
Command Failed	0x06	Command failed to execute for a command-specific reason
Bad Parameter Value	0x07	At least one data parameter is invalid
Busy	0x08	The operation is already in progress or the module is busy
Bad Target ID	0x09	Target does not exist
Target Unavailable	0x0A	Target exists but is unavailable (e.g., it is asleep or disconnected)
Currently Unused	0x0B – 0xFF	Currently unused

Payload Data

Supported Types

Type	Size(s) [bits]	Encoding
Unsigned Integer	8, 16, 32	
Signed Integer	8, 16, 32	two's compliment
Floating Point	32	IEEE 754
Bool	8	zero = false, non-zero = true
Byte Array	8 * n	
String	8 * n	generally null-terminated

Endianness

In the Sphero API, all multi-byte words are big-endian. Byte arrays and strings are sent in index-order (lowest indexed item first).

Standard (Preferred) Units

We aspire towards using the standard units below to make it easier on both you and our own team to be able to develop with confidence and consistency across our commands. We'll do our best not to let you down on this front and to adhere to these units for all of the API commands we all have the pleasure of utilizing.

- Distance: Meters
- Time: Seconds
- Speed: Meters / Second
- Angle: Degrees
- Velocity: Degrees / Second
- Acceleration: G
- Temperature: C
- RSSI: dB
- Quaternion: normalized
- Audio Volume: (TBD)

Packet Addressing and Routing

This API utilizes in-protocol packet routing, allowing peers to directly target each other's API nodes. In addition to more clearly specifying communication paths, this feature lets processors host identical commands (e.g., "Get Main App Version" or "Echo").

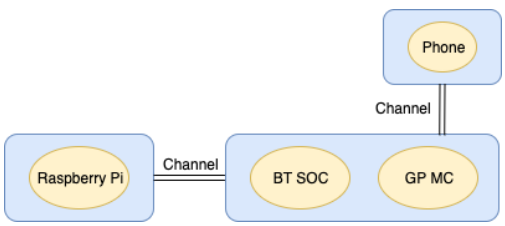
The Key Elements

The Sphero API is built on a system of nodes, which send and receive information. Each node is contained in a system and a system can contain more than one node. As an example, the RVR is a system that contains two nodes, (1) the Bluetooth SOC, which also handles the power system, indications, the color sensor, and the ambient light sensor and (2) a general-purpose microcontroller which handles the control system, IR communication, USB, and the IMU.

Connections Between Key Elements

Nodes (yellow ovals below) can communicate with other nodes within their own system (blue rounded rectangle below) OR with nodes in systems directly connected to their own, however, you cannot talk to a system (blue rounded rectangle below) through another system (only systems with direct connections ("channels") can communicate).

As an example, you can mount a board, such as a Raspberry Pi onto the top of RVR and connect the Raspberry Pi to the RVR via the RVR's UART port (creating a direct connection/channel between the two). You can also use the Sphero Edu App on your phone to control your RVR. Both the Raspberry Pi and the phone are connected to and can send information back and forth with the RVR, but the phone and the Raspberry Pi cannot communicate with one another (at least not through the RVR).



Sending Information via Channels

When information is sent between nodes (yellow ovals), there is a “source” node which *sends* the command and a “target” node which *receives* the command. In the Sphero API, when the target node (yellow oval) receives a command, it sends a response back to the original source node (yellow oval); the sending of this response packet causes the target and source to switch places (as the original target is now the sender (source) and the original source is now the receiver (target)).

The addressing system for each node (yellow oval) uses a two-part address, such that the first part of the address is the id of the port (red diamonds below) the packet leaves the system (blue rounded rectangle) through in order to get to the target node *from the source node's perspective* and the second part of the address is the id of the node (yellow oval) that is the source or target (depending on whether we are reporting the address of the source or the address of the target). A first (port) value of “0” means we are staying within the system (blue rounded rectangle).

Port ID (upper nibble)	**Node ID (lower nibble)**
Bits 7-4	Bits 3-0

Just like port “0”, the internal port, is reserved to address nodes on the same device, node “0”, the wildcard node, is reserved for commands that do not target a particular node on a device. In this case, the target system (system containing the target node) decides which node will service the command, if any (when a node receives a packet with an empty target node id, if it can handle it, it does; otherwise, it passes the packet down the line and each node goes through the same decision-making process) .

For the address of the target node (yellow oval), the port (red diamond) through which the packet leaves the source system (blue rounded rectangle) to get to the target node is the first part of the address, and the id of the node where the packet is being sent is the second part of the address. For the address of the source, (again, from the source's perspective) the source does not have to leave its system (blue rounded rectangle) to get to itself, so the value of the port is always “0” and the node value of the address is just the id of the source node (yellow oval).

To continue on with our example, below is a diagram of the two systems of the Raspberry Pi and the RVR. The Raspberry Pi system (blue rounded rectangle) contains just one node (yellow oval) for its processor and the RVR system (blue rounded rectangle) contains two nodes (yellow ovals), one each for the Bluetooth SOC and the general-purpose microcontroller mentioned above. The Raspberry Pi has a node id of “a” and the port (red diamond) to the Raspberry Pi system has an id of “1”. The Bluetooth SOC has a node id of “b”, the general-purpose microcontroller has a node id of “c” and the port (red diamond) to the RVR system has an id of “2”. (**Note**: these ids are arbitrary and may or may not happen to coincide with the ids used in your individual systems.)

If a command is sent from the Raspberry Pi node (yellow oval) to the general-purpose microcontroller node (yellow oval), the addresses would be as such:

- **Source: 0a** (as the source doesn't have to leave the system to find itself (port id = 0) and the id of the source node (Raspberry Pi) is “a”).
- **Target: 1c** (as, in order for the source node (Raspberry Pi) to find the target node (general-purpose microcontroller), it has to go through its port with an id of “1” and the id of the target node (general-purpose microcontroller) is “c”).

Now for the fun part: the response. As we noted above, the response, while it contains the same information (to allow us to check that everything arrived correctly), is a separate packet transmission, so (in our example) the source of the response is the general-purpose microcontroller node (yellow oval) and the target of the response is the Raspberry Pi node (yellow oval). This means that the addresses are now from the perspective of the general-purpose microcontroller. As such, the addresses for the response are:

- **Source: 0c** (as the source doesn't have to leave the system (blue rounded rectangle) to find itself and the id of the source node (general-purpose microcontroller) is “c”)
- **Target: 2a** (as, in order for the general-purpose microcontroller node to find the Raspberry Pi node, it must leave the RVR system (blue rounded rectangle) via port “2” and the id of the Raspberry Pi node is “a”)

Behind the Scenes

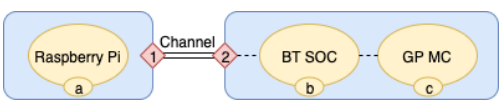
Note: The information (address system) above is all you truly need to understand in order to successfully use the API; this section is purely to dive a little deeper into how all of this is happening, for those who are curious.

In reality, a node only really knows about what is going on in its own system (so the Raspberry Pi node doesn't know about RVR system's port 2 and the general-purpose microcontroller doesn't know about the Raspberry Pi system's port 1) and, furthermore, each node only knows the first step toward getting to another node (inside or outside of its own system (blue rounded rectangle)). Let's dive into each of those concepts individually….

Nodes (yellow ovals) don't know about other systems' ports (red diamonds):

As was noted earlier, the addresses used to send the packets back and forth are sent from the source node for any given packet transfer exactly as we wrote them above. With the nodes only being familiar with their own systems, however, the addresses in a packet in transit must be translated before arriving at the target node, so that the target node knows where, from its own perspective, to send the response, once it is the source.

Using our same example as before:



As the command packet is crossing the channel between the two systems the address of the source is translated from **0a** to **2a** (so the port portion of the address changes to reflect which of the RVR system's ports the packet came through, but the node (yellow oval) it came from is still the same) and the address of the target is translated from **1c** to **0c** (so the port portion of the address changes to reflect that the target node (yellow oval) would not have to leave its own system (blue rounded rectangle) to find itself, but the id of the node (yellow oval) the packet is going to is still the same).

When a response is sent back, it, too, is translated so that the new target (the original source, Raspberry Pi) can understand the packet it has received and from where it received it. The response's source address is translated from **0c** to **1c** (again, just the port portion of the address changes to one that the receiving node (yellow oval) will understand) and the response's target address is translated from **2a** to **0a** (again, just the port portion of the address changes to one that the receiving node (yellow oval) will understand).

Again, you do not need to be an expert on this translation concept to successfully utilize the Sphero API; this concept is just a fun deep-dive for curious minds.

Nodes only know the first step towards their target:

To keep things simple and keep any node from having to do too much heavy lifting, each one has a "map", of sorts, of its own system (blue rounded rectangle), which tells it which pathway to follow out of itself to get to each of the nodes (yellow oval) and ports (red diamonds) in its system. This "map" is known as a "table map" and is literally just a table that lists a corresponding direction to head in for each of the given targets a node has access to. Using our same example:

If, as in the cross-system examples above, node "c" wants to send information through port "2" (now do you see why we use the port (red diamond) through which the packet exits the system in the addresses we plug into the API?), it knows it must go through node "b", and only that it must go through node "b"; there could be 7 more steps after node "b", but node "c" knows only that, if it wants to get information to port "2", it must pass that information to node "b" (per the dotted line in the above diagram). Node "b" has its own table that tells it what it needs to do in order to get information in its possession to the designated target. If node "b" receives a packet that is not addressed to node "b", it checks its table to determine where that packet must go next. The above system is fairly simple, but this process really comes in handy for more complex systems, so that each node (yellow oval) does not have to store entire pathways for each of the targets it can access.

Optimizing Transmission

Nodes are required to respect two optimizing rules:

- If a node receives a packet with no target ID, it should interpret it as "internal port, wildcard node."
- If a node receives a packet with no source ID, it should assume that the source is the node from which it received the packet. When forwarding the packet, the node should add the missing source ID. (You can read more about how nodes pass information above)

These rules allow senders to omit addressing in the following cases:

- Nodes may always omit the target ID if the intended target is "internal port, wildcard node."
- Nodes directly connected to port P, may omit the target ID if the intended target is "port P, wildcard node."
- The original sender of a packet may always omit the source ID.

Together, these optimizations mean that the follow configurations never need to transmit source or target IDs:

- Single-node peers
- Single-node devices using the wildcard ID to target a peer

Packet Routing

Packet routing may be implemented in any way that is convenient for a platform as long as commands are guaranteed to reach their targets, and targets produce valid responses. Specifically, if a target is invalid or unavailable, the sending node must receive the "Bad Target ID" or "Target Unavailable" from some node in the system.

Example Routing Strategy

Every node has a table that lists all the ports and nodes on its device. Each entry in the table maps to the channel that a packet should go through to reach that port or node.

When a packet is received, every node follows these rules:

- If the received packet targets the current node, handle the packet as appropriate.
- If the received packet targets a different node, look up the target's port or node in the table:
  - For targets on external ports, look up the port (ignoring the target node ID).
  - For targets on the internal port, look up the node.
- If the target port/node is found, forward the packet or respond with "Target Unavailable" as appropriate.
- If the target port/node is not found, respond with the "Bad Target ID" error. In this case, the target/source should be swapped as normal in the response packet so that it contains the bad target as the source.

Glossary

- **Channel** - a communication link between nodes
- **Client** - a node that sends API commands (typically, nodes are both clients and servers)
- **Device** - a system with one or more nodes or a group of commands on a device (as in “device ID”)
- **Internal Port** - a special port ID used to address nodes on the same device
- **Node** - any message processing point in an API network. Each node on a device has a permanent, unique ID, which should be the same on all similar devices.
- **Peer** - a device that is directly connected to another device through a port
- **Port** - a channel that links a node on a peer device (as opposed to a node on the same device) - you name the port you are going *from* (source) and you specify the node you are going *to* (target) - between systems along a chanel. As with nodes, each port on a device has a unique ID, which should be the same on all similar devices. If you are staying in a system (ST to Nordic or vice versa) you have a port of 0 (self) for both the target and the source.
- **Server** - a node that services API commands (typically, nodes are both clients and servers)
- **Source** - the node that originally sent a packet
- **Target** - a packet’s intended destination node
- **Wildcard Node** - a special node ID that matches any node supporting a packet’s command (0)