ASCOM ALPACA

API Reference

Abstract

This document is the technical reference for the ASCOM Alpaca APIs and describes how to use the API. It also explains some of the fundamental behavioural principles that underly the APIs and their effective exploitation.

Peter Simpson, Bob Denny
peter@peterandjill.co.uk, rdenny@dc3.com

Contents

1. Introduction

1.1 Language

When used within this document, these words have the following meanings:

Term	Meaning
Must	This is an absolute, mandated, requirement; no deviation is possible.
Should	This is highly recommended best practice, but is not mandated
Could	This is optional at the implementor's discretion
IP Endpoint	The host / IP address and port number on which the Alpaca device is operating
Application Endpoint	A URL that retrieves information about, or changes the state of, an ASCOM device. E.g. /api/v1/telescope/0/rightascension is an application endpoint because it returns the telescope's RA. E.g. /api/v1/telescope/0 is not an application endpoint because it neither returns information nor changes the state of the telescope.

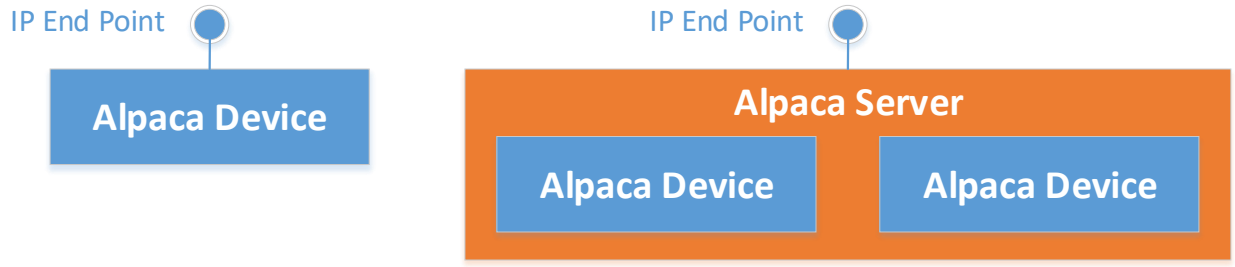
1.2 Terminology

The Alpaca API is designed to be flexible and to provide access to both:

- **Simple implementations** - where only one device is accessible on a particular IP endpoint (host:port).
- **Complex implementations** - where several independent devices, possibly of different ASCOM types, e.g. Telescope and Focuser, are accessible through the same IP endpoint.

In today's ASCOM COM architecture all devices fit in the "simple" model above where one ProgID, the device's well-known address, is associated with just one ASCOM device. To maintain continuity with current naming practice, simple implementations will be known as "**Alpaca Devices**".

Complex implementations will be known as "**Alpaca Servers**" that serve multiple "**Alpaca Devices**".



In the real world, an observatory may well have multiple Alpaca Devices and multiple Alpaca Servers within its overall design.

1.3 Consolidation

The Alpaca API supports consolidation of multiple downstream Alpaca Devices and Alpaca Servers into one virtual Alpaca Server presenting a single aggregated device tree under one IP end point.

There is no requirement that an Alpaca Server must contain Alpaca Devices, which opens the way to creation of physical devices to proxy Alpaca requests and the use of web servers acting as reverse proxies.

1.4 ASCOM Alpaca API Documentation

The ASCOM RESTful APIs are documented using the Swagger toolset and are available through a URL on the ASCOM Standards web site. The ASCOM API is fully documented here:

https://www.ascom-standards.org/api

To start exploring go to the above API URL and click a grey Show/Hide link to expand one of the sets of methods and then click the blue GET or orange PUT methods for detailed information on that API call.

Anyone who is familiar with the ASCOM COM based APIs will feel at home with the functionality available through the Alpaca API.

2. Alpaca API Contract

This section describes how to construct REST calls to devices that expose ASCOM Alpaca interfaces. It assumes a basic knowledge of HTTP, JSON and REST.

2.1 Alpaca API Format

2.1.1 Basic format

Alpaca APIs follow the standard Internet URL format:

```
http(s)://host:port/path?parameters
```

2.1.2 Alpaca API Path

The Alpaca device API path consists of five fixed elements:

```
/api/vversion_number/device_type/device_number/command
```

Fixed elements are black and variable elements are red.

Element Number	Element	Description
1	api	Fixed lower-case text denoting the root of the API path
2	vversion_number	Integer API version number prefixed with a lower-case v
3	device_type	ASCOM device type e.g. camera, telescope, focuser etc.
4	device_number	Integer device number of the required device
5	command	Command to be processed by the device in lower-case

For example, these are valid API calls:

```
http://api.peakobservatory.com/api/v1/telescope/0/atpark
http://api.peakobservatory.com/api/v1/camera/0/imagearray
```

2.1.3 Parameters

Many ASCOM methods require parameter values. All methods that use the **HTTP GET** verb should include parameters as query string name-value pairs.

All methods that use the **HTTP PUT** verb should include name-value parameters in the body using the "application/x-www-form-urlencoded" media type.

For example, these are valid API parameters on an HTTP GET transaction:

```
/api/v1/telescope/0/canslew?clientid=0,clienttransactionid=23
```

2.2 Case Sensitivity

2.2.1 API Path

All five elements of the API path are **case sensitive** and must always be in **lower case**. For example, this is the only valid casing for a call to the Telescope.CanSlew property:

```
/api/v1/telescope/0/canslew
```

These are examples of invalid casing:

~~/API/V1/TELESCOPE/0/CANSLEW~~ ~~/Api/V1/Telescope/0/CanSlew~~

~~/api/v1/telescopE/0/canslew~~ ~~/api/v1/telescope/0/CanSlew~~

2.2.2 Alpaca API Parameters

Alpaca parameters are key-value pairs where:

- The parameter **key** is **case insensitive**
- The parameter **value** can have **any casing** required.

This is the same behaviour as defined for HTTP header keys in RFC7230.

For example, these are all valid API parameters:

```
/api/v1/telescope/0/canslew?clientid=0,clienttransactionid=23

/api/v1/telescope/0/canslew?ClientID=0,ClientTransactionID=23

/api/v1/telescope/0/canslew?CLIENTID=0,CLIENTTRANSACTIONID=23
```

Clients and drivers must expect incoming API parameter keys to have any casing.

2.3 Locale and Culture

The Alpaca API is culture neutral in order to facilitate use between clients and devices running in different locales, e.g. a client running on a UK locale device connecting to a remote device running with a Spanish locale.

This has consequences for data formats in two circumstances:

2.3.1 Encoding Parameter Values That Have Decimal Points

When decimal parameter values are passed into the API, they must use period (0x2E) as the decimal separator. This is so that they can be reliably parsed on receipt

E.g. **23.456** is a valid value to supply when setting the Telescope.TargetRightAscension property, while ~~**23,456**~~ is not a valid value.

2.3.2 JSON Responses

JSON responses must be formatted in accordance with the JavaScript Object Notation (JSON) Data Interchange Format specification RFC 8259. In consequence Alpaca devices must use the period character (0x2E) as the decimal separator when returning decimal values.

Alpaca Clients must expect to receive decimal values in this invariant culture format and to interpret them accordingly.

e.g. Clients must parse the value returned by the Telescope.SiteElevation property using period as the decimal separator, regardless of the locale in which they are running.

2.4 Http Verbs

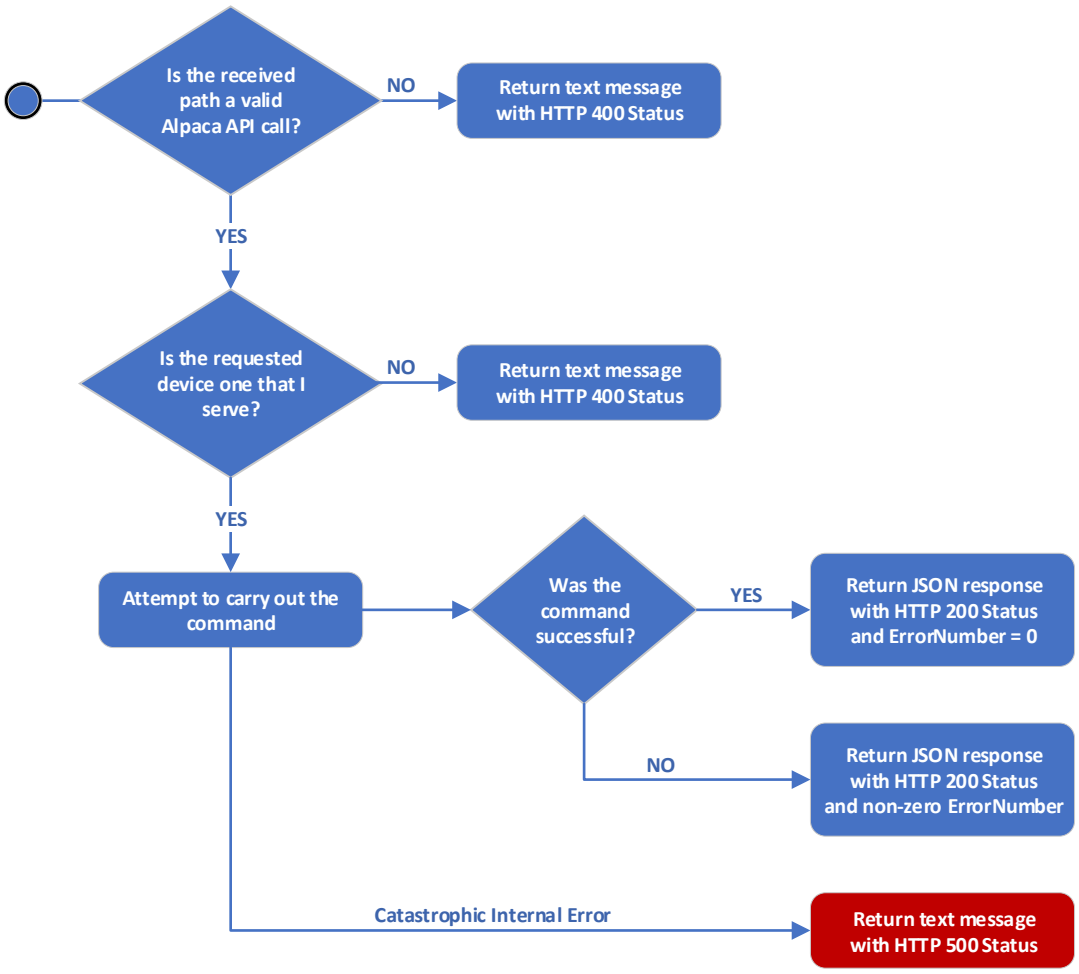
Verb	Description
GET	Used for all information retrieval where the device state is not changed, e.g. most properties and a few functions such as Telescope. AxisRates(Axis).
PUT	Used for all commands which change the state of the device, e.g. Telescope.SideOfPier and Telescope.SlewToCoordinates().

2.5 HTTP Status Codes

Returned HTTP status codes should reflect the device's status as follows:

Code	Interpretation	Extended Interpretation
200	**OK**	The API request was handled successfully and the response is in the expected JSON format for the supplied command.
400	**Bad request**	The API request could not be understood. The response is a text error message and is not in the expected JSON format.
500	**Internal Server Error**	A technical error occurred in the Alpaca device which prevented successful processing of the request. The response is a text error message and is not in the expected JSON format.

The following flow diagram shows how to decide whether to return a 200, 400 or 500 HTTP status.



Please note that, catastrophic errors aside, a 200 status must be returned when the device understands the supplied command regardless of whether or not it can action it in an ASCOM sense.

2.5.1 Status Code Examples - Transactions with Valid Paths

A "200" status code must be returned if the transactions below were received by an Alpaca Telescope device that supports Alpaca interface version 1:

Path	Reason for Rejection with 200 Status
PUT /api/v1/telescope/0/park	Assuming this telescope device does not have park functionality, it would return a NotImplemented Alpaca error code (0x400) and message with an HTTP 200 status.
PUT: /api/v1/telescope/0/siteelevation **New value: -400**	Elevations lower than -300m are invalid so return an InvalidValue Alpaca error code (0x401) and message with an HTTP 200 status.

2.5.2 Status Code Examples - Transactions with Bad Paths

A "400" status code must be returned if the transactions below were received by an Alpaca Camera device that only supports Alpaca interface version 1:

Path	Reason for Rejection with 400 Status
GET: /apii/v1/telescope/0/canslew	Valid Alpaca API requests start with "api" rather than "apii".
GET: /api/v2/telescope/0/canslew	The Alpaca "v2" API is not supported by the device.
GET: /api/v1/telescop/0/canslew	"Telescop" is not one of the valid ASCOM device types.
GET: /api/v1/camera/0/canslew	CanSlew is not a valid Camera command.

2.6 ID Fields

To aid operational management and debugging, three optional ID fields are defined that can be supplied as parameters on read and write transactions:

ID	Maintained by	Description
ClientID	Client	This is a 32-bit unsigned integer that the client can choose to identify itself. It is recommended that values should be within the range 0::65535 so that log files appear orderly and readable!
ClientTransactionID	Client	This is a 32-bit unsigned integer that the client maintains. The value should start at 1 and be incremented by the client on each request to the Alpaca device.
ServerTransactionID	Alpaca Device	This is a 32-bit unsigned integer that the server maintains. The value should start at 1 and be incremented by the Alpaca device on each request.

The client id and transaction numbers should be supplied by the client in the request to uniquely identify the client instance and specific transaction. The Alpaca device must return the client transaction number, or zero if no value was supplied by the client, as part of its response to enable the client to confirm that the response does relate to the request it submitted.

Alpaca devices should record client ids and transaction numbers in their logs so that issues can be tied back to specific transactions and outcomes correlated with client-side logs.

The server transaction id should be returned by the Alpaca device with every response so that issues identified on the client side can easily be correlated with Alpaca device logs.

2.7 JSON Responses

The outcome of the command is returned in JSON encoded form. The following information is always returned in every transaction response that has an HTTP 200 status:

Item	Type	Contents
ClientTransactionID	Unsigned 32-bit integer	Transaction ID supplied by the client in its request
ServerTransactionID	Unsigned 32-bit integer	The server's transaction number.
ErrorNumber	Signed 32-bit integer	ASCOM Alpaca error number, see section 2.8.3.
ErrorMessage	String	If the driver throws an exception, its message appears here, otherwise an empty string is returned.

In addition, the JSON response will include the output from the command (if any) in the "Value" parameter. This example is from the Telescope Simulator SupportedActions property:

```
GET /api/v1/telescope/0/supportedactions?Client=1&ClientTransaction=6
```

{"Value":["AssemblyVersionNumber","SlewToHA","AvailableTimeInThisPointingState","TimeUntilPointingStateCanChange"],"ClientTransactionID":6,"ServerTransactionID":6, "ErrorNumber":0, "ErrorMessage":"" }

This example shows the response for:

```
GET /api/v1/telescope/0/canslewasync?Client=1&ClientTransaction=20
```

{"Value":true,"ClientTransactionID":20,"ServerTransactionID":168, "ErrorNumber":0, "ErrorMessage":"" }

2.8 Reporting Device Errors Through the Alpaca API

2.8.1 Historic COM Approach

ASCOM COM drivers use a range of reserved ASCOM exceptions and unique driver specific exceptions to report issues to COM clients such as "this method is not implemented" or "the supplied parameter is invalid" and these are documented in the Developer Help file at:

https://ascom-standards.org/Help/Developer/html/N_ASCOM.htm

Each exception has an associated HResult code in the range 0x80040400 to 0x80040FFF for historic reasons related to Microsoft's approach to error handling for COM applications. When expressed as signed integers these exception numbers translate into very large and unwieldy negative numbers e.g. 0x80040400 becomes -2,147,220,480 and 0x80040FFF becomes -2,147,217,409.

2.8.2 New Alpaca Approach

Alpaca devices still need to express different error conditions to the client so, for Alpaca, the error number range has been simplified to the range 0x400 (1024) to 0xFFF (4095) by truncating the leftmost 5 digits so that an Alpaca error number of 0x401 would have the same meaning as the original COM error with HResult of 0x80040401.

2.8.3 ASCOM Reserved Error Numbers

The following table relates the new Alpaca error codes for reserved ASCOM error conditions to the corresponding COM HResult numbers, which are in the range 0x80040400 to 0x800404FF.

Condition	Alpaca Error Number	COM Exception Number
Successful transaction	0x0 (0)	N/A
Property or method not implemented	0x400 (1024)	0x800400400
Invalid value	0x401 (1025)	0x800400401
Value not set	0x402 (1026)	0x800400402
Not connected	0x407 (1031)	0x800400407
Invalid while parked	0x408 (1032)	0x800400408
Invalid while slaved	0x409 (1033)	0x800400409
Invalid operation	0x40B (1035)	0x80040040B
Action not implemented	0x40C (1036)	0x80040040C

2.8.4 Driver Specific Error Numbers

The Alpaca error number range for driver specific errors is 0x500 to 0xFFF and their use and meanings are at the discretion of driver / firmware authors.

2.8.5 Error Number Backwards Compatibility

Native Alpaca clients will inspect the ErrorNumber and ErrorMessage fields as returned to determine if something went wrong with the transaction. However, to ensure COM client backward compatibility, ASCOM Remote clients will translate Alpaca error numbers into their equivalent COM exception numbers before throwing the expected ASCOM exceptions to the COM client.

2.8.6 Driver Error Example

The following example shows the expected invalid value JSON response when an attempt is made to set the site elevation to -400, which is below the minimum allowed value of -300.

```
PUT /api/v1/Telescope/0/SiteElevation
```

(parameters for the PUT verb are placed in the form body (not shown here) and do not appear after the URI as they do for the GET verb)

Expected JSON response:

{"ClientTransactionID":23,"ServerTransactionID":55,**"ErrorNumber":1025,**
"ErrorMessage":"SiteElevation set - '-400' is an invalid value. The valid range is: -300 to 10000."}

3. ASCOM APIs - Essential Concepts

Today's world is clearly modular, cross-platform, and distributed. The core aspect of any modular system is its interfaces. If a system is built on top of poorly designed interfaces, it suffers throughout its life with limitations, instabilities, gremlins, and the like. Interface design *and negotiation* is an engineering art, the best practitioners are those that have suffered and learned.

3.1.1 Object Models - Properties and Methods

The ASCOM APIs are built on an object model which provides properties that represent some state of the device, and methods that can change the state of the device. For example, the current positional right ascension of a telescope mount is a property, and a command to slew the mount to a different position is a method. In ASCOM COM, properties are normally accessed by assignment statements in the native syntax of any of twenty languages on Windows. Methods are represented by native syntax function calls, some with parameters. There are exceptions. Some properties require parameters to signify some aspect of state, and thus may be represented by a function call which returns the property value (which need not be a scalar), for example, Telescope.AxisRates(Axis).

3.1.2 ASCOM API Characteristics

The following information applies to the existing COM-based ASCOM APIs as well as the REST-based APIs. The behaviours must be the same to provide transparent interoperation.

- **Routine Operations:** Interface design always involves some negotiations between the parties. Inevitably, a device maker may wish to have included in the interface some clever means to make their device stand out above those of his competitors. On the other hand, client programmers don't want to be writing code to manage an ever-expanding set of these clever functions. It defeats the purpose of the standardized API. The ASCOM API was therefore designed at its outset to cover *routine operations only*.

  For example, a mount really only needs "point to these coordinates" and "track the apparent motion of my object". The more accurately it does these things, the better. As a client program developer, I don't want to be concerned about PEC or encoder resolutions or servo currents.

- **Synchronous vs Asynchronous Methods:** One may think of a method call as one that returns only when the requested operation has completed, which is a synchronous call. But some types of operations can benefit by *starting* the operation and returning immediately. For example, the Rotator.Move() method may return immediately.

  If so, its return means only that the rotation was successfully *started*. Rotators are typically slow, and the system can benefit by overlapping mount and rotator movement, so both provide asynchronous calls. The status properties such as Rotator.IsMoving and Telescope.IsSlewing are used to monitor progress of asynchronous calls.

- **"Can" Properties:** Some ASCOM APIs have "can" properties, which tell the client whether or not a corresponding capability is available. For example, in the Telescope API, the CanSlewAltAz property tells the client whether this specific mount can successfully execute

the SlewToAltAz() method. These "can" properties exist only for methods which can't be directly tested without changing the state of the device.

For example, a client *can* tell that the mount provides its positional azimuth by trying to read the Azimuth property; it will either get an answer or a "not implemented" error. However, a client *cannot* tell whether a mount can slew to alt/az coordinates without calling the method and possibly changing the mount's position. This is why a CanSlewAltAz property is provided for the SlewToAltAz() method.

3.1.3 Behavioural Rules

Heterogeneous distributed systems require both common standardized APIs and a set of behavioural rules that must be obeyed by all modules in the system. The *implementation* of a module is where these rules are effected, they do not appear in the abstract API definitions themselves. These behavioural rules are already implemented by ASCOM COM drivers.

ASCOM's modular rules are:

- **Do it right or report an error:** Fetching or changing a property, or calling a method, must always result in one of two outcomes: The request must complete successfully, or an error must be signalled, preferably with some (human readable) indication of why the request could not be satisfied. An example of violating this rule would be a method call to move a rotator to a given mechanical angle, but the rotator ends up at some other angle and no error is reported to the caller.
- **Retries prohibited:** No module must ever depend on another to provide timeouts or retry logic. If a device needs check-and-retry logic in its routine operation, that logic must be contained within the module itself. If there's a problem and your module's own retry logic can't resolve the issue report the error as required above.
- **Independence of operations:** To the extent possible with the device, each API operation should be independent of the others. For example, don't impose a specific call order such as needing to fetch the positional right ascension of a mount immediately before fetching the declination.
- **Timing Independence:** To the extent possible with the device, modules must not place timing constraints on properties and methods. Implement asynchronous calls wherever possible in order not to lock up clients unnecessarily.
- **Self-Protection – Over Use:** Drivers must protect themselves and the instrument from excessive rates of incoming requests from clients. Of course, clients should minimize the need for calling across the internet to avoid flooding, but responsibility for protecting a device from excessive request rates rests with the device and its driver.
- **Self-Protection – Illegal/hazardous operations:** Drivers and instruments should protect themselves from illegal or hazardous operations. E.g. a dome may be opening but receives a request to close the shutter. If the shutter can be safely reversed while opening, the driver could simply close the shutter and report success. Alternatively, the driver may permit the shutter to fully open and return an illegal operation error response to the close command.
- **No Status Inconsistencies:** In the example above, the driver ShutterStatus property must accurately reflect the physical shutter condition at all times. If it reports ShutterOpen, even for an instant, before the shutter starts to open, the client will assume that the shutter is properly open and move on to its next task, even though the shutter is still opening.