ALPACA IMAGEBYTES

Concept and Implementation

Abstract

This paper describes a new Alpaca specific standard to transfer images efficiently between Alpaca Camera devices and Alpaca clients. The new mechanic supersedes the Base64Handoff mechanic that was introduced in 2019.

Peter Simpson
peter@peterandjill.co.uk

Contents

1. Context

Shortly after introducing Alpaca, it became clear that the Camera.ImageArray JSON mechanic was very slow when transferring large images. This is because conversion of large image datasets to JSON is computationally expensive and results in substantial network traffic.

Following discussion on the ASCOM Developer Forum, the Base64Handoff (ASCOM Remote Description) mechanic was developed, which converts the camera's image array to a byte array and then encodes the byte values using base64 for transfer over the network connection.

The Base64Handoff mechanic provides a substantial performance improvement over JSON, but use has exposed a number of opportunities for improvement:

- Clients must set a proprietary HTTP header to indicate that they implement the handoff mechanic and inspect response headers for the presence of the proprietary header to confirm that the Alpaca device supports the mechanic. Setting and inspecting proprietary headers is not always straightforward for Alpaca clients and devices.
- The handoff mechanic is a two-step process, requiring two network round trips to retrieve every image:
    1. To retrieve the metadata describing the array.
    2. To retrieve the base64 encoded array data.
- While efficient, the base64 algorithms still add a processing overhead to encode the image data on the Alpaca device and to decode it on the client.
- The base64 encoded image data is about 33% larger than the original data, extending retrieval times and consuming greater network bandwidth than an optimal solution.
- Image data is transferred over the network as 32bit values in accordance with the Camera interface specification, although most image data does not exceed a 16bit dynamic range. An opportunity to halve the amount of data being transferred is not taken.

The new ImageBytes mechanic mitigates the issues above.

2. ImageBytes Mechanic

ImageBytes is an Alpaca specific, single step mechanic that transfers image data as a structured binary byte stream and uses standard HTTP headers for discovery and control. This mechanic:

- Is effected directly through the Camera.ImageArray and Camera.ImageArrayVariant Alpaca endpoints and consequently does not require additional REST endpoints.
- Sends image data in binary form rather than base64encoded form avoiding the base64 encode and decode overheads as well as reducing network traffic
- Provides for data of byte and Int16/UInt16 size to be transmitted as one / two-byte values rather than as four-byte Int32 values.
- Sends array metadata together with array data in a single structured byte stream.
- Orders array metadata before array data so that the image array data structure can be created before reading the array data.
- Is Alpaca specific and does not change the ASCOM Camera interface definition.

3. ImageBytes Benefits

Compared to the JSON and Base64Handoff mechanics, the ImageBytes mechanic simplifies implementation, reduces Alpaca device processor requirements and improves image download times by:

- Making use of standard HTTP Accept and Content-Type headers, avoiding the need to create and inspect proprietary headers.
- Eliminating a network round trip to the device by employing a single step that returns both array metadata and image data in a single response.
- Transferring binary data rather than base64 encoded data, which:
    - Reduces network traffic by about 33%
    - Eliminates the base64 encoding overhead on Alpaca devices and the base64 decoding overhead on Alpaca clients.
- Transferring 8bit and 16bit image data over the network as 8 bit / 16bit values rather than as 32bit values, which:
    - Reduces network traffic by a further 50% to 75%, at the expense of some additional processing at the client to change the received data back to the Int32 form required by the interface specification.
    - Improves response times for the user.

In addition:

- There is no impact on the Camera interface definition or version number because there are no new interface members.
- The new mechanic is backward compatible with both the JSON and Base64Handoff mechanics.
    - Current Alpaca and COM clients are not impacted in any way.
    - COM clients can take advantage of the new mechanic by using the Platform's revised Alpaca Dynamic Camera clients.

4. Performance Benefits

Performance benefits depend on device capabilities, network speed and topology. As a benchmark, these timings were obtained over a 650Mbit/sec 802.11ac wireless link by a Windows laptop client when accessing a Camera Simulator hosted by an ASCOM Remote Server running on a desktop PC.

Image Data: 6000 x 4000 monochrome image[1] totalling 24 million elements.

	Image Transfer Time (seconds)		
Data Value Range	**JSON[2]**	**Base64Handoff[3]**	**ImageBytes[4]**
Int32	25.6	3.4	2.3
Int16	16.3	3.3	1.0
UInt16	15.1	3.3	1.1
Byte	12.6	3.5	0.7

[1] Array data were random values within the minimum and maximum values of the data type.
[2] JSON timings are influenced by the average number of text digits (including the negative sign) across the data range.
[3] Base64Handoff timings are not influenced by data value range because elements are always transferred as four-byte Int32 values.
[4] ImageBytes timings are influenced by the number of bytes required to hold the data type

Image Data: 6000 x 4000 x 3 plane colour RGB image[1] totalling 72 million elements.

	Image Transfer Time (seconds)		
Data Value Range	JSON[2]	Base64Handoff[3]	ImageBytes[4]
Int32	95.2	10.2	6.4
Int16	64.9	9.9	3.5
UInt16	56.9	9.6	3.5
Byte	47.4	10.1	1.9

5. ImageBytes Implementation

To retrieve the camera's image data the client makes an Alpaca call to the Camera.ImageArray property as described in the Alpaca specification.

5.1 Client Initiation

If a client supports the ImageBytes protocol, it indicates this to the Alpaca device by including the "application/imagebytes" mime type in the HTTP "Accept" header that it sends to the Alpaca device's ImageArray endpoint.

5.2 Device Response

If the Alpaca device doesn't support the ImageBytes mechanic, it will return the image data as a JSON string and indicate this by setting the "Content-Type" header in its response to "application/json". However, if the client supports the ImageBytes mechanic, it will return the image as a binary byte stream, in the format described below, and indicate this by setting the "Content-Type" header to "application/imagebytes".

5.3 Client Response Handling

On receiving the response from the Alpaca device, the client will inspect the "Content-Type" header to determine whether the Alpaca device has sent a JSON string or an ImageBytes binary stream. If a JSON string has been returned the client will deserialise it as for any other Alpaca API call. However, if an ImageBytes response is received, the client will:

- Inspect the first four bytes of the binary data to determine the metadata version
- Extract the metadata from the byte stream
- Use the metadata to determine whether the operation was successful or resulted in an error condition
- If successful:
    - Use the metadata to construct an image array of appropriate type and dimensions.
    - Read the data from the byte stream and populate the image array.
    - Return the array to the application for processing.
- If unsuccessful:
    - Extract the error number from the metadata
    - Extract the UTF encoded error message bytes
    - Decode the UTF bytes back to a string
    - Return an error to the application containing the error number and message.

Tools to facilitate these operations will be provided as part of an ASCOM Cross-Platform Library.

6. ImageBytes Binary Data Format

The returned binary data always comprises two parts:

1. A standard metadata structure.
2. The returned data.

The nature of the data depends on whether the operation succeeded or failed:

6.1.1 Operation Succeeded

1. **Metadata:** Information describing the image data, with the error number field set to zero.
2. **Data:** Image data (2 or 3-dimension array of element values as a serialised byte stream)

6.1.2 Operation Failed

1. **Metadata:** As for "operation succeeded" but with a non-zero error number.
2. **Data:** Error message as a UTF8 encoded byte stream.

7. Metadata

7.1 Metadata Structure

The following structure describes the returned metadata:

```
int MetadataVersion; // Bytes 0..3 - Metadata version = 1
int ErrorNumber; // Bytes 4..7 - Alpaca error number or zero for success
uint ClientTransactionID; // Bytes 8..11 - Client's transaction ID
uint ServerTransactionID; // Bytes 12..15 - Device's transaction ID
int DataStart; // Bytes 16..19 - Offset of the start of the data bytes
int ImageElementType; // Bytes 20..23 - Element type of the source image array
int TransmissionElementType; // Bytes 24..27 - Element type as sent over the network
int Rank; // Bytes 28..31 - Image array rank (2 or 3)
int Dimension1; // Bytes 32..35 - Length of image array first dimension
int Dimension2; // Bytes 36..39 - Length of image array second dimension
int Dimension3; // Bytes 40..43 - Length of image array third dimension (0 for 2D array)
```

7.2 Image and Transmission Array Element Types

The following image element data types are supported:

```
public enum ImageArrayElementTypes
{
    Unknown = 0, // 0 to 3 are values already used in the Alpaca standard
    Int16 = 1,
    Int32 = 2,
    Double = 3,
    Single = 4, // 4 to 9 are an extension to include other numeric types
    UInt64 = 5
    Byte = 6,
    Int64 = 7,
    UInt16 = 8,
    UInt32 = 9
}
```

8. Serialised Array Formatting

8.1 Element Ordering

The ImageArray and ImageArrayVariant property definitions specify that returned arrays must be:

- Monochrome and Bayer Matrix images (2D array): **`Array[NumX, NumY]`**
- Colour images (3D array): **`Array[NumX, NumY, ColourPlane]`**

Where NumX indicates the image width, NumY indicates the image height and ColourPlane uses the values 0, 1 and 2 to represent the red, green and blue colour planes.

The C, C++, C# and VB.NET languages use row-major ordering[5] to store element values in memory where the rightmost array index changes most quickly. This array serialisation order is used by ImageBytes to maximise serialisation and deserialization performance. For image arrays this results in:

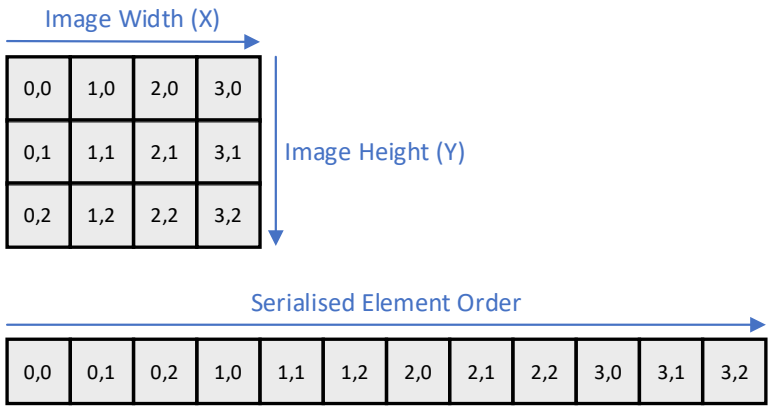
- **2D array element ordering**: the rightmost, **height** dimension changes most quickly in the serialised byte stream followed by the **width** dimension
- **3D array ordering:** the rightmost **colourplane** dimension changes most quickly in the serialised byte stream, followed by the **height** dimension followed by the **width** dimension

Note: Counterintuitively this approach results in an element order where element values appear ordered by *image height* rather than by *image width*. This happens because the ASCOM image array specification defines the image height dimension as being to the right of the image width dimension, which means that the height dimension changes most quickly when row-major serialised.

8.1.1 Two-dimensional Array Order

The following pseudo code and illustration show how elements are ordered in a 2D array:

```
for (int i = 0; i < NumX; i++)
{
    for (int j = 0; j < NumY; j++)
    {
        SerialiseElement(imageArray2D[i, j]);
    }
}
```

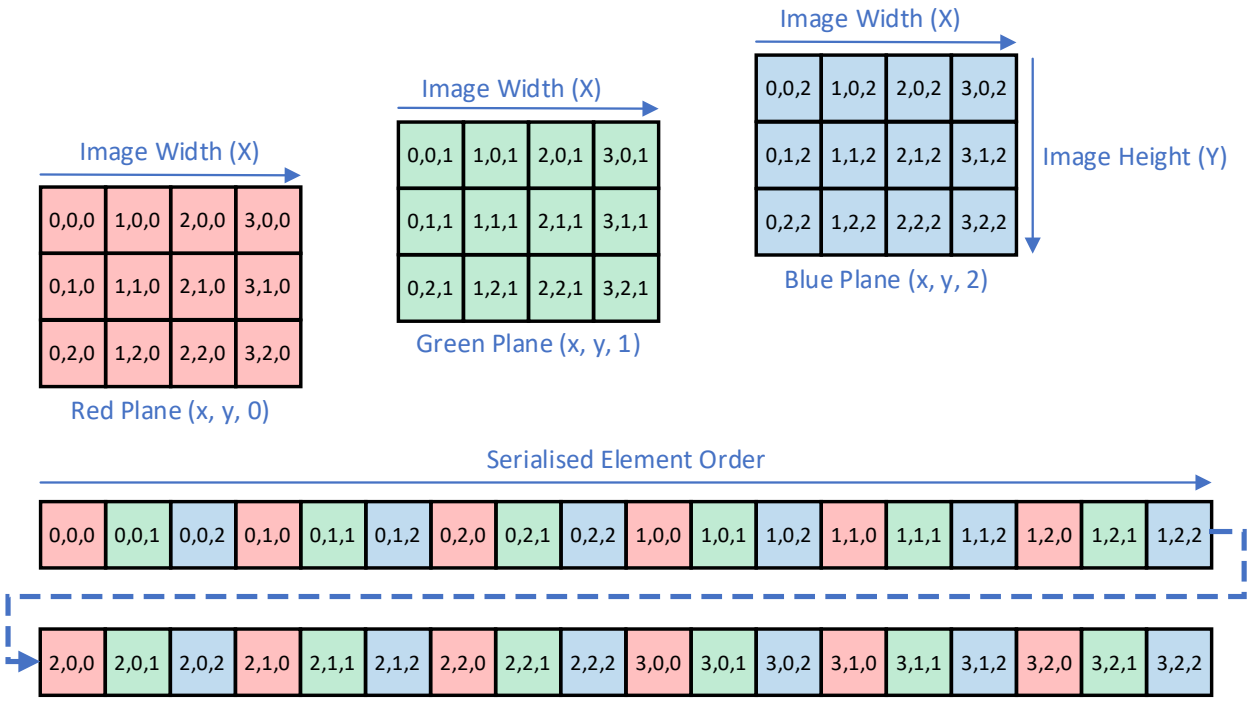


[5] Wikipedia - Row and Column Major Order.

8.1.2 Three-dimensional Array Order

The following pseudo code and illustration show how elements are ordered in a 3D array:

```
for (int i = 0; i < NumX; i++)
{
    for (int j = 0; j < NumY; j++)
    {
        for (int k = 0; k < 3; k++)
        {
            SerialiseElement(imageArray3D[i, j, k]);
        }
    }
}
```



8.2 Integer Byte ordering

For maximum compatibility, the "little endian" format will be used for integer values. This is compatible with:

- MacOS on Intel processors
- MacOS on Apple silicon processors
- Most modern Linux distros
- Modern ARM chips that use the chip default endianness
- Raspberry Pi
- Arduino
- Windows on Intel and AMD

For example, the Int32 integer `2,135,263,542` (`0x7F458936`) will be serialised as the byte stream: [36] [89] [45] [7F].

9. Error Handling

To signal an error condition to the client, the Alpaca device must:

- Set the ErrorNumber field to a non-zero Alpaca error number to indicate the type of issue.
- Return an error message by UTF8 encoding the message string and putting the resultant bytes at the offset indicated by the DataStart field, in place of the image array data bytes.
    - Note: The first 128 UTF8 characters are identical to the first 128 ASCII characters.
    - **Alpaca Devices**: Error messages that only use ASCII characters do not require further UTF8 encoding.
    - **Alpaca Devices**: A string terminator is not required because the end of the returned byte array signals the end of the message string.
    - **Clients**: Must implement UTF8 decoding and not assume that all devices will send ASCII characters.

10. Support Routines

Library routines will be provided to assist developers building Alpaca clients and devices using .NET Framework, .NET Core and .NET 5/6. These will:

- Create a byte array, including metadata, from a supplied image array and, if possible, reducing transmission size by converting element values to types that occupy less space. E.g. **Int32** values will be converted to:
    - **Int16** if all image array element values are in the range: -32768 to 32767.
    - **UInt16** if all image array element values are in the range: 0 to 65535.
    - **Byte** if all image array element values are in the range: 0 to 255.
- Create an **Int32** array of the required dimensions from a supplied byte array containing a metadata structure and the image array data.
- Return the metadata version of a supplied byte array.
- Validate and extract all metadata values in a supplied byte array as a struct for easy manipulation. Validation failures will result in exceptions being thrown.
- Return an error message string from a supplied byte array that has a non-zero error number.
- Provide the appropriate ASCOM COM exception, given an error message and an Alpaca error number.

11. Document History

Date	Version	Change
9th January 2022	1.0	• Initial release