BINARY DATA

View online: https://www.construct.net/en/make-games/manuals/construct-3/plugin-reference/binary-data

The **Binary Data plugin** allows raw access to an allocated section of memory (often referred to as a *buffer*). For example it could allocate 16 bytes of memory, and read and write anywhere in that buffer as individual bytes, 32-bit integers, floating point numbers, text, and so on.

A comprehensive description of how binary data/computer memory storage works is out of the scope of this manual. However there is lots of information on the Internet that covers it, and most computer science or computing courses will also cover it. The Binary Data object is also useful even if you do not access its contents: it integrates with other plugins like AJAX and Local Storage, allowing binary data such as images to be stored or transferred in useful ways. Despite the name, Binary Data can also store text, using the *Set from text* action and *GetAllText* expression, which can be useful with other plugins such as Cryptography.

Construct expressions, like JavaScript, only use double-precision floating point numbers (Float64). When reading and writing other data types with Binary Data, they are converted to and from Float64 - no other types are used in expressions. Fortunately Float64 can store all other types losslessly.

The Binary Data object starts empty (with a zero byte buffer). You must set its length or load data from another source before reads and writes can be used.

Unlike unmanaged languages like C, the Binary Data object is implemented in the memory-safe language JavaScript. This means the binary data cannot be used unsafely: out-of-bounds writes are ignored, and out-of-bounds reads return 0.

Scripting

When using JavaScript or TypeScript coding, the features of this object can be accessed via the IBinaryDataInstance script interface. (JavaScript and TypeScript have support for binary data built-in, but this allows for interacting with binary data used in event sheets.)

Binary Data properties

Endian

The default endian to use when reading and writing binary values.

Binary Data conditions

Compare length

Compare the length of the data buffer, in bytes.

Compare value

Read a value of a given data type at an offset in the data buffer, and then compare that value to another number.

Binary Data actions

Copy

Copy data from another (or the same) Binary Data object. This does not change the size of the buffer. The *Start* and *Length* specify the range of data in the source Binary Data object to copy, defaulting to copying the entire buffer. The *Target* is a byte offset to write the copy at.

Fill

Fill a range of the buffer with copies of the same value. For example this can be used to fill an entire range of the buffer with zero bytes. The *Offset* and *Length* can be used to specify a byte range to fill, but they default to covering the entire buffer.

If you fill a value other than Int8 or Uint8, make sure the range size is a multiple of the size of the data type. For example if you fill with a Uint32 value, make sure the range is a multiple of 4 bytes. If the range is not a multiple, the end of the range will not be filled, e.g. filling a 5 byte range with a 4 byte value will only write the first 4 bytes and will not alter the last byte.

Set endian

Change the endian used when reading and writing values.

Set from Binary Data

Set the contents of the Binary Data object to a copy of another Binary Data object. This changes the size of the buffer to be the same as the other object.

Set from string

Set the contents of the Binary Data object from the contents of a string, changing the size of the data buffer to fit it. The string may be provided in three formats:

- **Text:** set the data to the provided text string encoded as UTF-8.

- **Base64:** set the data by decoding the provided base64-encoded string to binary.
- **Hex:** set the data by decoding the provided hexadecimal-encoded string to binary.

Set length

Set the size of the memory buffer in bytes. Note Binary Data initially has a zero sized buffer, so this must be used first before any reads or writes can complete. The existing data is preserved when changing the size; if the new size is smaller, data is truncated, and if the new size is larger, zero bytes are added.

Set value

Write a value of a given data type at a byte offset in the memory buffer. The entire value must be within the bounds of the buffer, otherwise the write is ignored. For example a 4-byte Uint32 value cannot be written anywhere in a 3-byte buffer, because all four bytes must be inside the buffer.

Compress

Decompress

Use a compression algorithm to compress or decompress the data stored in the Binary Data object. The supported compression algorithms are GZIP and DEFLATE. The same algorithm as used for compression must be used for decompression. Compression algorithms work by identifying and eliminating repeating patterns in data and can significantly shrink the amount of data. However the effectiveness depends on the size and type of data. Small amounts of data, or highly unpredictable data, generally does not compress well; large amounts of highly repetitive data generally compresses very well. See the Compression example for a demonstration.

These actions are asynchronous, which means they can take a moment to complete while working in the background. Use the system action Wait for previous actions to complete before using any further actions to work with the resulting data.

Binary Data expressions

ByteLength

Return the current length of the buffer in bytes.

GetBase64

Return the entire contents of the data buffer encoded as a base64 string. This is useful when binary data must be stored in a text-based format like JSON.

> *Base64 data is larger and slower to process than the equivalent binary data. It is more efficient to avoid converting to base64 where possible. For example instead of posting an image to a server as a base64 string, the AJAX object is able to post a Binary Data object directly.*

GetHex

Return the entire contents of the data buffer encoded as a hexadecimal string. This always uses two hexadecimal digits per byte.

> *Base64 is a more efficient text representation of binary data. Hexadecimal is only generally used for relatively small amounts of data such as hashes.*

GetURL

Return a URL that can be used locally to load the binary data. For example if the Binary Data represents an image, this URL can be passed the Sprite object's *Load image from URL* action to load the image from the Binary Data object.

> *The URL is a `blob:` URL referring to data in memory. It can only be used in the same session, in the same browser, on the same device. Sharing the URL or saving it to be re-used in another session later will not work.*

> *The provided URL will be valid until the next time the data stored in the Binary Data object changes. For example while the data stays the same, the GetURL expression continues to return the same URL. However if any part of the Binary Data object's stored data changes, the URL returned by the GetURL expression will change, and the old URL will become invalid and no longer work.*

GetUint8(byteOffset)

GetInt8(byteOffset)

GetUint16(byteOffset)

GetInt16(byteOffset)

GetUint32(byteOffset)

GetInt32(byteOffset)

GetFloat32(byteOffset)

GetFloat64(byteOffset)

Read a value of the corresponding type from the buffer at a byte offset. The entire value must be within the bounds of the buffer, otherwise it returns 0. For example a 4-byte Uint32 value cannot be read from a 3-byte buffer, because all four bytes must be inside the buffer.

> *Construct expressions, like JavaScript, do not use different number types and instead treat all values as Float64. Therefore the read value is always converted to Float64 for use in expressions.*

GetText(byteOffset, length)

Decode text in the UTF-8 encoding in a range of the buffer in bytes, and return the string. If the UTF-8 encoding is invalid, or any part of the range is outside the buffer, an empty string is returned.

GetAllText

Decode all the data in the Binary Data object as text in the UTF-8 encoding. This is the same as using the *GetText* expression with a range that specifies all the data.