[[strawman:
binary_data]]

ES
Wiki

Binary data

Table of Contents

Goals

Provide portable, memory-safe, efficient, and structured access to compact (i.e., contiguously allocated) binary data, as well as an interface for external binary I/O facilities such as XMLHttpRequest, HTML5 File API, and WebGL.

Desiderata:

- expressive and convenient way to create structured binary data
- no new primitive (i.e., non-object) ECMAScript values
- admit architecture-native internal representation while preserving portability:
  - hide struct layout/padding
  - hide endianness
  - prevent multiple interpretations of the same binary data structure at different types
- convenient conversion to native ECMAScript values
- reference semantics without changing ECMAScript evaluation model

The design of this library allows implementations to represent allocated binary data in architecture-specific formats – in particular, using the architecture’s native padding/alignment and endianness – without exposing these details to ECMAScript. This allows for efficient implementation without creating portability hazards.

Examples

```
const Point2D = new StructType([["x", uint32], ["y", uint32]]);
const Color = new StructType([["r", uint8], ["g", uint8], ["b", uint8]]);
const Pixel = new StructType([["point", Point2D], ["color", Color]]);

const Triangle = Pixel.array(3);

let t = new Triangle([{ point: { x:  0, y: 0 }, color: { r: 255, g: 255, b: 255 } },
                            { point: { x:  5, y: 5 }, color: { r: 128, g: 0,   b: 0   } },
                            { point: { x: 10, y: 0 }, color: { r: 0,   g: 0,   b: 128 } }]);
...
```

TODO: more examples

Notation

This spec uses *Italicized* names for datatypes of ECMAScript values, i.e., values that can be stored in variables and object properties and passed to and returned from functions. E.g.:

- *UInt32* - integer values in the range [0, 2^{32})
- *Int32* - integer values in the range [-2^{31}, 2^{31})
- *Any* - any ECMAScript value

This spec uses **bold** names for datatypes of spec-internal data structures. E.g.:

- **string** - internal strings, such as the value of the [[ClassName]] property
- **reference[t]** - a reference into the program store (aka heap) to an internal value of type **t**
- **reference[***Object***]** - a reference into the program store (aka heap) to an ECMAScript value of type *Object*

Binary data blocks

This spec introduces a new, spec-internal **block** datatype, intuitively representing a contiguously allocated block of binary data. Blocks are not ECMAScript values and appear only in the program store (aka heap).

A **block** is one of:

- a **number-block**
- an **array-block[t,** *n***]**
- a **struct-block[t*1*, ..., t*n*]**

A **number-block** is one of:

- an **unsigned-integer**
- a **signed-integer**

a **floating-point**

An **unsigned-integer** is one of **uint8**, **uint16**, **uint32**, or **uint64**. A **signed-integer** is one of **int8**, **int16**, **int32**, or **int64** A **floating-point** is one of **float32** or **float64**

A **uint*k*** is an integer in the range [0, 2^k). An **int*k*** is an integer in the range [-2^{k-1}, 2^{k-1}). A **float*k*** is a floating-point number representable as a *k*-bit IEE754 value.

An **array-block[t**, n**]** is an ordered sequence of *n* blocks of homogeneous block type **t**. Each element of the array is stored at in independently addressable location in the program store, and multiple Block objects may contain references to the element.

A **struct-block[t*1*, ..., t*n*]** is an ordered sequence of *n* blocks of heterogeneous block types **t*1*** to **t*n***, respectively. Each field of the struct is stored at in independently addressable location in the program store, and multiple Block objects may contain references to the field.

The spec also introduces a datatype of *Block* objects, which are ECMAScript values that encapsulate references to **block** data in the program store. Every *Block* object has the following properties:

- [[ClassName]] = "Block"
- [[Value]] : **reference[block]** – a reference to a block in the program store
- [[BlockType]] : **reference[***BlockType***]** – a reference to a *BlockType* describing this object's block data

Block types

Every binary data block is associated with a fixed *BlockType* object, which describes the permanent shape, size, and interpretation of the block, somewhat like a runtime type tag. All references to a given block in the program store are associated with the same *BlockType*. Consequently, implementations can allocate blocks as untagged memory buffers (e.g., raw C data structures) without violating memory safety.

Every block type supports the following properties:

- [[ClassName]] = "BlockType" – the constant class name of all *BlockType* objects
- [[BlockTypeName]] : string – a string determining the variant of *BlockType*, for compatibility between block objects across multiple execution environments (such as multiple windows, module loaders, etc.)
- [[Convert]](val : *Any*) → **reference[block]** – converts an ECMAScript value to a reference to a binary data block
- [[IsSame]](t : *BlockType*) – compare block types for equality
- `bytes` : *UInt32* – the *logical* size of blocks of this type, in bytes

Note that the `bytes` property does not expose information about the *actual* size of a block type, just the *logical* size of its components. This avoids exposing architecture- and implementation-specific details like struct padding.

There is a built-in `BlockType` constructor which serves as a "base class" for block types:

```
function BlockType()
```

The “abstract” base constructor for block types, whose prototype is a descendant of all block type prototypes. Calling the `BlockType` constructor always raises an error; it is provided solely for access to its prototype.

`BlockType`.[[Prototype]] = `Function.prototype`

`BlockType.prototype`.[[Prototype]] = `Function.prototype`

`BlockType.prototype.constructor` = `BlockType`

`BlockType.prototype.array`(*n* : *UInt32* | *Int64* | *UInt64*) → ArrayBlockType

Convenience method that produces the same result as `new ArrayType(this, n)`.

Block objects

There is a built-in `Block` constructor which serves as a “base class” for block types:

```
function Block()
```

The “abstract” base constructor for block objects, whose prototype is a descendant of all block object prototypes. Calling the `Block` constructor always raises an error; it is provided solely for access to its prototype.

`Block`.[[Prototype]] = `BlockType.prototype`

`Block.prototype`.[[Prototype]] = `Object.prototype`

`Block.prototype.constructor` = `Block`

`Block.prototype.update`(*val* : *Any*) → *Void*

```
If !IsObject(this) || this.[[ClassName]] != “Block”
    Throw TypeError
Let R ?= this.[[BlockType]].[[Convert]](val)
Copy Dereference(R) into this.[[Value]]
Return undefined
```

Numeric blocks

There are several pre-defined block type objects describing numeric binary data.

```
var uint8, uint16, uint32, uint64 : BlockType
var int8, int16, int32, int64 : BlockType
var float32, float64 : BlockType
```

Let *t* be one of the above built-in block type objects.

t.[[Prototype]] = `BlockType.prototype`

t.[[BlockTypeName]] = the name of *t* as a string (e.g., “uint8” for `uint8`)

t.[[Convert]](val : *Any*) → **reference[block]**

```
Let R = a reference to a new number-block of the appropriate size for t
If val = true
    R := 1
Else If val = false
    R := 0
Else If val is an ECMAScript number in the domain of this type
```

```
        R := val
    Else If val is a UInt64 or Int64 and val.[[Value]] is in the domain of t
        R := val.[[Value]]
    Else Throw TypeError
    Return R

t.[[IsSame]](u : BlockType)
    Return t.[[BlockTypeName]] = u.[[BlockTypeName]]

t.[[Cast]](val : Any) → number-block
    Let V = t.[[Create]](val)
    If !IsError(V)
        Return Dereference(V.value)
    If val = Infinity or val = NaN
        Return 0
    If val is an ECMAScript number
        Return t.[[CCast]](val)
    If val is a UInt64 or Int64
        Return t.[[CCast]](val.[[Value]])
    If val is a numeric string, possibly prefixed by "0x" or "0X" for uints or /(-)?(0[xX])?/ for ints
        Let V = ParseNumber(val)
        Return uintk.[[CCast]](V)
    Throw TypeError

t.[[CCast]](n : number) → number-block
    TODO: do roughly what C does

t.[[Call]](val : Any) → Number
    Let x ?= t.[[Cast]](x)
    Let R = a reference to a new number block with value x       Return t.[[Reify]](R)

t.[[Reify]](R : reference[number-block])
    Let x = Dereference(R)
    If t.[[BlockTypeName]] = "uint64"
        Return a new UInt64 with [[Value]] x
    If t.[[BlockTypeName]] = "int64"
        Return a new Int64 with [[Value]] x
    Return x
```

For each built-in type object *t* with suffix *k* (e.g., for `uint32`, *k* = 32):

t.`bytes` = *k* / 8

Array blocks

Programmers can create array block-type objects, which describe fixed-length sequences of block data of homogeneous block-type, using the following constructor.

```
function ArrayType(elementType : BlockType, length : UInt32 | Int64 | UInt64) ->
BlockType
```

`ArrayType.prototype`.[[Prototype]] = `BlockType.prototype`

`ArrayType.prototype.constructor` = ArrayType

```
ArrayType.prototype.repeat(val : Any) → ArrayBlock
    If !IsObject(this) || this.[[ClassName]] != "BlockType" || this.[[BlockTypeName]] != "array"
        Throw TypeError
    Let V = this.[[Construct]]()
    For each integer i in [0, this.[[Length]])
        Let R ?= this.[[ElementType]].[[Convert]](val)
        Copy Dereference(R) into the ith element of this.[[Value]]
    Return V
```

Array block-type objects

Let `elementType` be a *BlockType* object and `length` a non-negative integer. Then it is possible to define a new array block-type object *t* using the `ArrayType` constructor:

```
t = new ArrayType(elementType, length)
```

The resulting array block-type object *t* has the following properties:

t.[[ClassName]] = "BlockType"

t.[[Prototype]] = `ArrayType.prototype`

t.[[BlockTypeName]] = "array"

t.[[ElementType]] = `elementType`

t.[[Length]] = `length`

```
t.[[Convert]](val : Any) → reference[block]
    If IsObject(val) && val.[[ClassName]] = "Block"
        If val.[[BlockType]].[[IsSame]](t)
            Return val.[[Value]]
        Throw TypeError
    If !IsObject(val)
        Throw TypeError
    Let u = t.[[ElementType]]
    Let n = t.[[Length]]
    Let L ?= val.[[Get]]("length")
    If L not in UInt32 || L !== n
        Throw TypeError
    Let R = a reference to a new array-block of the appropriate size for n elements of type u
    For each integer i in [0, n)
        Let V ?= val.[[Get]](i)
        Let W ?= u.[[Convert]](V)
        Copy Dereference(W) into the ith member of R
    Return R
```

```
t.[[IsSame]](u : BlockType)
    Return u.[[BlockTypeName]] = "array" &&
      t.[[ElementType]].[[IsSame]](u.[[ElementType]]) &&
      t.[[Length]] = u.[[Length]]
```

```
t.[[Construct]](val : Any) → Block
    (described below)
```

```
t.[[Reify]](R : reference[array-block])
    Let V = a new block object with
        V.[[BlockType]] = t
        V.[[Value]] = R
    Return V
```

t.`prototype`.`constructor` = *t*

```
t.prototype.fill(val : Any) → Void
    If !IsObject(this) || this.[[ClassName]] != "Block" || !this.[[BlockType]].[[IsSame]](t)
        Throw TypeError
    For each integer i in [0, t.[[Length]])            Let R ?= t.[[ElementType]].[[Convert]](val)
        Copy Dereference(R) into the ith element of this.[[Value]]
    Return undefined
```

t.`elementType` = `elementType`

t.`length` = `length`

t.`bytes` = `elementType.bytes` x `length`

Array block objects

Given an array block-type object such as *t* above, it is possible to construct new array blocks:

```
a = new t()
a = new t(val)
```

The resulting array block object *a* has the following properties:

a.[[ClassName]] = "Block"

a.[[Prototype]] = *t*.`prototype`

```
a.[[Value]] = a newly allocated struct-block of the appropriate size for t, initialized to all zeroes
    If val is defined
        Let R ?= t.[[Convert]](val)
        Copy Dereference(R) into a.[[Value]]
```

a.[[BlockType]] = *t*

a.length = *t*.[[Length]]

For each *i* in [0, *t*.[[Length]]):

```
get a.i()
    Let R be a reference to the ith element of a.[[Value]]
    Return t.[[ElementType]].[[Reify]](R)
```

```
set a.i(x)
    Let R ?= a.[[ElementType]].[[Convert]](x)
    Copy Dereference(R) into the ith member of a.[[Value]]
    Return undefined
```

Struct blocks

Programmers can create struct block-type objects, which describe fixed-length sequences of block data of heterogeneous block-types, using the following constructor.

```
function StructType(fields : [ [ string, BlockType ], ... ]) -> BlockType
```

`StructType.prototype`.[[Prototype]] = `BlockType.prototype`

`StructType.prototype.constructor` = `StructType`

Struct block-type objects

Let `fields` be an ECMAScript value. Then it is possible to define a new struct type object *t* using the `StructType` constructor:

```
t = new StructType(fields)
```

```
Semantics
    Let n ?= fields.[[Get]]("length")
    If n is not in UInt32
```

```
        Throw TypeError
    For each i in [0, n)
        Let V ?= fields.[[Get]](i)
        Let L ?= V.[[Get]]("length")
        If L !== 2
            Throw TypeError
        Let s1 ?= V.[[Get]](0)
        Let t1 ?= V.[[Get]](1)
    Let t be a new object with t.[[Fields]] = [ (s0, t0), … ] and the properties below
    Return t
```

t.[[ClassName]] = "BlockType"

t.[[Prototype]] = `StructType.prototype`

t.[[BlockTypeName]] = "struct"

```
t.[[Convert]](val : Any) → reference[block]
    If IsObject(val) && val.[[ClassName]] = "Block"
        If val.[[BlockType]].[[IsSame]](t)
            Return val.[[Value]]
        Throw TypeError
    If !IsObject(val)
        Throw TypeError
    Let names ?= EnumerateOwn(val)
    If names != { X | (X, u) in t.[[Fields]] }
        Throw TypeError
    Let R = a reference to a new struct-block of the appropriate size for t.[[Fields]]
    For each (X, u) in t.[[Fields]]
        Let V ?= val.[[Get]](X)
        Let W ?= u.[[Convert]](V)
        Copy Dereference(W) into field X of R
    Return R
```

```
t.[[IsSame]](u : BlockType)
    Return t === u
```

```
t.[[Construct]](val : Any) → Block
    (described below)
```

```
t.[[Reify]](R : reference[struct-block])
    Let V = a new block object with
        V.[[BlockType]] = t
        V.[[Value]] = R
    Return V
```

t.`prototype.constructor` = *t*

t.`fields` = a frozen array representing *t*.[[Fields]], in the same format as the `fields` constructor argument above

t.`bytes` = *t0*.`bytes` + … + *tn-1*.`bytes`

Struct block objects

Given a struct block-type object such as *t* above, it is possible to construct new array blocks:

```
s = new t()
```

The resulting struct block object *s* has the following properties:

s.[[ClassName]] = "Block"

s.[[Prototype]] = *t*.`prototype`

s.[[Value]] = a newly allocated **struct-block** of the appropriate size for *t*, initialized to all zeroes

s.[[BlockType]] = *t*

s.`constructor` = *t*

For each ("*fi*", ui) in *t*.[[Fields]]:

```
get s.fi()
    Let R be a reference to the ith field of s.[[Value]]
    Return ui.[[Reify]](Dereference(R))

set s.fi(x)
    Let R ?= ui.[[Convert]](x)
    Copy Dereference(R) into the ith field of s.[[Value]]
    Return undefined
```

Prototype hierarchy

To help visualize the prototype inheritance relationship, the following diagram illustrates a case of an array type object `A` and an array block `a` created by `A`.

TODO: this is accurate for js-ctypes but may or may not be accurate for this spec – stay tuned, we're working on it. :)

```
Type Constructors               Type Objects                     Block Objects
-----------------              --------------                   ---------------

        to Function.p                 to Function.p                   to Object.p
      ^                             ^                               ^
      | [[Prototype]]               | [[Prototype]]                 | [[Prototype]]
      |                             |                               |
 +-----------+                 +-------------+                 +---------------+
 |           |----prototype--->|             |----prototype--->|               |
 | BlockType |                 |    Block    |                 |    Block.p    |
 |           |<--constructor---|             |<--constructor---|               |
 +-----------+                 +-------------+                 +---------------+
      ^                             ^                               ^
      | [[Prototype]]               | [[Prototype]]                 | [[Prototype]]
      |                             |                               |
 +-----------+                 +-------------+                 +---------------+
 |           |----prototype--->|             |----prototype--->|               |
 | ArrayType |                 | ArrayType.p |                 | ArrayType.p.p |
 |           |<--constructor---|             |<--constructor---|               |
 +-----------+                 +-------------+                 +---------------+
                                    ^                               ^
                                    | [[Prototype]]                 | [[Prototype]]
                                    |                               |
                               +-------------+                 +---------------+
                               |             |----prototype--->|               |
                               |   A         |                 |    A.p        |
                               |             |<--constructor---|               |
                               +-------------+                 +---------------+
                                                                    ^
                                                                    | [[Prototype]]
                                                                    |
                                                               +---------------+
                                                               |               |
                                                               |      a        |
                                                               |               |
                                                               +---------------+
```

Large integers

Some operations produce or require integers larger than the ECMAScript number type can represent. The following two types are a simple object type that encapsulates 64-bit integers, both unsigned and signed.

Unsigned 64-bit integers

```
new UInt64(n : String | Number | Int64 | UInt64 = 0) -> UInt64
```

Semantics

```
If n is a string
    Let V ?= ParseInt(n)
    If V not in [0, $2^{64}$) Throw TypeError
    Let W = a new UInt64 object with W.[[Value]] = V.
    Return W
If n is a number and n in [0, $2^{64}$)
    Let W = a new UInt64 object with W.[[Value]] = n
    Return W
If n is an Int64 object and n.[[Value]] in [0, $2^{64}$)
    Let W = a new UInt64 object with W.[[Value]] = n.[[Value]]
    Return W
If n is a UInt64 object
    Let W = a new UInt64 object with W.[[Value]] = n.[[Value]]
    Return W
Throw TypeError
```

```
UInt64.lo(n : UInt64) -> UInt32
```

Returns the low-order 32-bit value of n.[[Value]].

```
UInt64.hi(n : UInt64) -> UInt32
```

Returns the high-order 32-bit value of n.[[Value]].

```
UInt64.join(hi : Number | Int64 | UInt64, lo : Number | Int64 | UInt64) -> UInt64
```

Returns a new *UInt64* whose [[Value]] is computed by joining the numeric value of `hi` as the high-order 32 bits and the numeric value of `lo` as the low-order 32 bits.

```
UInt64.compare(a : UInt64, b : UInt64) -> -1 | 0 | 1
```

If a and b are both *UInt64* objects, returns -1 if a.[[Value]] < b.[[Value]], 0 if a.[[Value]] = b.[[Value]], and 1 if a.[[Value]] > b.[[Value]]. Otherwise throws a TypeError.

```
UInt64.prototype.toString(radix : 2 | 10 | 16 = 10) -> String
```

Returns a string representation of **this**.[[Value]] in base `radix`, consisting of one or more lowercase digits of base `radix`.

Signed 64-bit integers

```
new Int64(n : String | Number | Int64 | UInt64 = 0) -> UInt64
```

Semantics

```
If n is a string
    Let V ?= ParseInt(n)
    If V not in [-2^63, 2^63) Throw TypeError
    Let W = a new Int64 object with W.[[Value]] = V.
    Return W
If n is an integer and n in [-2^63, 2^63)
    Let W = a new Int64 object with W.[[Value]] = n
    Return W
If n is a UInt64 object and n.[[Value]] in [-2^63, 2^63)
    Let W = a new Int64 object with W.[[Value]] = n.[[Value]]
    Return W
If n is an Int64 object
    Let W = a new Int64 object with W.[[Value]] = n.[[Value]]
    Return W
Throw TypeError
```

```
Int64.lo(n : Int64) -> UInt32
```

Returns the low-order 32-bit value of n.[[Value]].

```
Int64.hi(n : Int64) -> Int32
```

Returns the high-order 32-bit value of n.[[Value]].

```
Int64.join(hi : Number | Int64 | UInt64, lo : Number | Int64 | UInt64) -> Int64
```

Returns a new *Int64* whose [[Value]] is computed by joining the numeric value of `hi` as the high-order 32 bits and the numeric value of `lo` as the low-order 32 bits.

```
Int64.compare(a : UInt64, b : UInt64) -> -1 | 0 | 1
```

If a and b are both *Int64* objects, returns -1 if a.[[Value]] < b.[[Value]], 0 if a.[[Value]] = b.[[Value]], and 1 if a.[[Value]] > b.[[Value]]. Otherwise throws a TypeError.

```
Int64.prototype.toString(radix : 2 | 10 | 16 = 10) -> String
```

Returns a string representation of **this**.[[Value]] in base `radix`, consisting of a possible leading minus sign followed by one or more lowercase digits of base `radix`.

Rationale

Aliasing

Block objects encapsulate references to blocks of mutable data in the program store. These references can be shared and aliased. In other words, block objects provide a "reference semantics" for binary data.

The member accessors of array and struct blocks allow the creation of new block objects with references to shared data. As a result, it is possible for multiple objects to refer to the same store location. Thus, the same reference may be pointed to by two block objects that are distinguished by the strict-equality (===) operator.

An alternative would be to require memoization of block objects, so that any reference is the root reference of at most one block object. However, this could be difficult to implement (since a reference may always be a nested part of a larger block in the heap), and it does not eliminate aliasing (since, again, struct and array accessors allow sharing references to nested sub-blocks).

Numeric blocks

The numeric types are meant to simulate datatypes with non-reference semantics from languages like C. There are several potential approaches to providing this functionality:

1. provide a reference semantics for numeric types via block objects
2. introduce new non-reference primitive datatypes (aka value types)
3. memoize block objects to simulate value semantics with objects

This spec takes a simpler approach, simply eliminating immediate access to numeric blocks as ECMAScript values.

Consider Python's `struct` library, or js-ctypes. In both, it is possible to construct a first-class numeric block, which is given a separate object identity and heap-allocated cell containing a number. It is not clear that this provides much utility, since the same can easily be achieved with a one-element array type or one-field struct. Moreover, it leads to confusing properties such as:

```
(new uint32(42)) == (new uint32(42)) // false
```

We could attempt to patch around this problem by extending the semantics of == (but not ===, since that must always distinguish two distinct objects), but it seems more consistent simply to avoid simulating new primitive datatypes with reference-typed objects.

Large integers

The *Int64* and *UInt64* types are the simplest realization of 64-bit integers possible, but they are not ideal. It would likely be better to add language support for bignums. In the interest of keeping this spec orthogonal, we have used *Int64* and *UInt64*.

Struct-type constructor API

In the js-ctypes API, struct types take as their argument an array of field descriptors expected to have exactly one own-property:

```
new StructType([{ x: uint32 }, { y: uint32 }])
```

An even more convenient form would be to allow the use of a single object literal, using the (admittedly subtle!) enumeration order of the properties to determine the order of the fields in the struct layout:

```
new StructType({ x: uint32, y: uint32 })
```

Because struct types are meant to be compatible with actual I/O, where the order of struct fields is significant, it must be easy to guarantee the order of fields. For this reason, the explicit order of arrays makes it easier to reason about the order of fields.

It might however, be good to provide a hybrid interface to the `StructType` constructor to allow both the convenient API and the more explicitly-ordered API. It is not entirely clear how to distinguish the two different types of input, however, except possibly to offer two different API's, e.g.:

```
new StructType({ x: uint32, y: uint32 })
StructType.create([ [ "x", uint32 ], [ "y", uint32 ] ])
```

Deviations from js-ctypes

- *ConvertToJS* isn't an appropriate name; renamed to *Reify*
- renamed CData to Block and CType to BlockType
- no numeric CData, to avoid treating "value types" as reference types
- numeric CTypes are only cast functions, not block-constructors
- *ImplicitConvert* is just called *Convert*
- *ExplicitConvert* is called *Cast* and only works on number block types
- compound CTypes are only block-constructors, not cast functions

To do

- API to produce struct descriptor
- alternative construction forms for struct-types (object-literal convenience vs. array of arrays) and structs (positional vs object?)
- move `update` into `BlockType.prototype`?
- chars and string conversion
- note non-configurable and non-writeable attributes throughout

References

Common Lisp:

- http://www.gigamonkeys.com/book/practical-parsing-binary-files.html

Python:

- http://docs.python.org/library/struct.html

Mozilla JavaScript:

- https://developer.mozilla.org/en/javascript_code_modules/ctypes.jsm
- https://wiki.mozilla.org/Jsctypes/api

Racket:

- http://docs.racket-lang.org/foreign/types.html
- http://www.ccs.neu.edu/scheme/pubs/scheme04-bo.pdf

Survey:

- http://autocad.xarch.at/lisp/ffis.html