Formal Verification of the *SharedArrayBuffer* ECMAScript specification: Memory Model

Cristian Mattarei `mattarei@stanford.edu`

November 29, 2016

Preliminaries

This document integrates the specification of the JavaScript Memory Model with a set of formal interpretations used to perform a rigorous validation. The content of this document is automatically extracted from the HTML specification [SAB16], thus it also contains some parts that are not relevant to the FM.JS project. In fact, we refer the reader to the Memory Model Section 6.3.1.

Contents

List of Tables

List of Theorems

Todo list

Stage 2 Draft / November 15, 2016ECMAScript Shared Memory and AtomicsSome algorithms and semantics in this proposal are presented as modifications to existing ES262 algorithms and semantics. The base revision of ES262 for the modifications (here denoted "ES7") is currently f35248729043089b47ccf29e4c47559386f5f6fd (2016-06-29). At the moment, ES7 (ES2016) section numbering differs from ES6 (ES2015) only for TypedArrays (section 22.2).

Introduction

This proposal adds a simple form of shared memory to ECMAScript.Shared memory is being exposed in the form of a new SharedArrayBuffertype; the existing TypedArray and DataView types are adapted in such a waythat they can be used to create views on shared memory. The newglobal Atomics object provides atomic operations on shared memorylocations, including operations that can be used to create blockingsynchronization primitives. Though not a part of this specification,we intend for the role of "threads" to be played by Web Workers in webbrowser environments.

At this time the proposal adds only low-level primitives toECMAScript; we expect the developer community to put togetherabstractions that are useful in specific domains.

The work has been driven by the following use cases:

- Support for threaded code in programs written in other languages that are translated to asm.js or plain JS or a combination of the two, notably C and C++ but also other, safe, languages.
- Support for hand-written JS or JS+asm.js that makes use of multiprocessing facilities for select tasks, such as image processing, asset management, or game AI.

The proposal makes only very basic assumptions about the requiredhardware and should be broadly implementable with good performance.Prototype implementations of the API are available in Firefox and inGoogle Chrome.

This specification constitutes a restatement and formalization ofan earlierwork, thespec document for which also contains additional rationale,background information, discussions, implementation notes, andcomments.

Changelog:

- 2016-11-15 – Fix bug 149: TypedArray(typedArray) constructor handling of shared memory
- 2016-11-14 – Pull the coherence condition out of *reads-bytes-from* §6.3.1.8 and into its own constraint.
- 2016-11-14 – Refine all partial orders into strict partial orders.
- 2016-11-14 – Provide spec draft notes for the intuition that a valid execution always exists.
- 2016-11-02 – Expand compiler and codegen guidelines.
- 2016-10-28 – Add an informal preface to the *memory model* §6.3 section.
- 2016-10-28 – Make candidate executions more precise, formally define relations and orders.
- 2016-10-25 – Fix Tear Free Aligned Reads to distinguish TypedArray vs DataView provenance instead of just alignment.
- 2016-10-25 – Model candidate executions explicitly with evaluation semantics that can read all possible values; model events as Records.
- 2016-10-24 – Fix RMW events to correctly return the read value instead of the modified value.
- 2016-10-24 – Add Tear Free Aligned Reads requirement to valid executions; fixed Access Atomicity to impart that the *memory model* §6.3 only solves ordering of events and that access atomicity is an orthogonal property.

- 2016-10-24 – Rewrote *memory model* §6.3, again, to be axiomatic.
- 2016-07-22 – Editorial fixes.
- 2016-07-21 – Replace [[SharedArrayBufferData]] and [[SharedArrayBufferByteLength]] with [[ArrayBufferData]] and [[ArrayBufferByteLength]]; remove specialization that was based on the removed properties and instead use type checks with *IsSharedArrayBuffer* §6.2.1.2 in ArrayBuffer, SharedArrayBuffer, and Atomics methods; spec that Shared Data Blocks are distinguishable (by unspecified means) from regular Data Blocks and use this to defined *IsSharedArrayBuffer* §6.2.1.2.
- 2016-07-20 – Editorial fix: clarify the operator passed to AtomicFetchOp.
- 2016-07-13 – Editorial fixes.
- 2016-07-11 – Rewrote the *memory model* §6.3 section, which introduced separate access functions for shared memory, and caused many other changes.
- 2016-07-05 – Rewrote (and shrunk) the informative Web Browser Embedding section so that it only addresses the requirements of this spec, not how HTML ought to evolve.
- 2016-07-05 – Rephrased the wait/wake mutual exclusion in terms of critical sections named by (G,i), as the old specification was too strong.
- 2016-06-30 – Algorithms that are modified relative to their ES262 forms include more context and are more clearly marked, and there's a defined ES262 base revision for those modifications. Updated section references to reference that revision.
- (Older changelog removed)

Chapter 1

Overview (ES7 4)

1.1 ECMAScript Overview (ES7 4.2)

In the third paragraph, include SharedArrayBuffer after ArrayBuffer.

Chapter 2

ECMAScript Data Types and Values (ES7 6)

2.1 ECMAScript Language Types (ES7 6.1)

2.1.1 The Number Type (ES7 6.1.6)

In the NOTE, include SharedArrayBuffer along with ArrayBuffer.

2.1.2 The Object Type (ES7 6.1.7)

2.1.2.1 Well-Known Intrinsic Objects (ES7 6.1.7.4)

In table 7, include rows for *Atomics* §6.3, *SharedArrayBuffer*, and *SharedArrayBufferPrototype* in the manner of the row for ArrayBuffer.

2.2 ECMAScript Specification Types (ES7 6.2)

2.2.1 Data blocks (ES7 6.2.6)

Insert the following paragraphs after the third:

A data block that resides in memory that can be referenced from multiple agents concurrently is designated a Shared Data Block. A Shared Data Block has an identity (for the purposes of equality testing Shared Data Block values) that is address-free: it is tied not to the virtual addresses the block is mapped to in any process, but to the set of locations in memory that the block represents. Shared Data Blocks can also be distinguished from Data Blocks.

The semantics of Shared Data Blocks is defined using Shared Data Block events by the Atomics *memory model* §6.3. Abstract operations below act as the interface between evaluation semantics and the event semantics of the *memory model* §6.3. The operations thus introduce ReadSharedMemory, WriteSharedMemory, and ReadModifyWriteSharedMemory events. Because of the axiomatic nature of the *memory model* §6.3, the values of these events are not known until the set of all events is known. The operations thus also nondeterministically read any possible Number value of a particular type during evaluation. The events introduced during an evaluation form a graph. The event graph together with the nondeterministically chosen values and other relations defined in the *memory model* §6.3 form a *candidate execution* §6.3.1.5. The *memory model* §6.3 does not admit all candidate executions; the allowed candidate executions are called valid executions, or simply executions. There is at least one valid execution for a given evaluation.

A *Shared Data Block event* §6.3.1.2 are modeled by Records, defined in the Atomics *memory model* §6.3. All agents in an agent cluster share the same *candidate execution* §6.3.1.5 record in its *Agent Record* §3.3's [[CandidateExecution]] field, which is initialized to an empty *candidate execution* §6.3.1.5 *Record*.

> **Note:** One may think of the evaluation semantics of ECMAScript as a generating function for candidate executions. The *memory model* §6.3 then prunes invalid executions from the set of candidate executions. A single-agent ECMAScript program that does not use SharedArrayBuffer (and, for simplicity, ignoring other sources of nondeterminism such as **Math.random()**) always has exactly one *candidate execution* §6.3.1.5, which is also a valid execution. A multi-agent ECMAScript program that uses SharedArrayBuffer has many candidate executions and at least one valid execution.

2.2.1.1 CopyDataBlockBytes(toBlock, toIndex, fromBlock, fromIndex, count) (ES7 6.2.6.2)

This algorithm is modified as follows:

When the abstract operation CopyDataBlockBytes is called, the following steps are taken:

1. Assert: `fromBlock` and `toBlock` are distinct *Data Block* or *Shared Data Block* §2.2.1 values.
2. Assert: `fromIndex`, `toIndex`, and `count` are integer values ≥ 0.
3. Let `fromSize` be the number of bytes in `fromBlock`.
4. Assert: `fromIndex`+`count` ≤ `fromSize`.
5. Let `toSize` be the number of bytes in `toBlock`.
6. Assert: `toIndex`+`count` ≤ `toSize`.
7. Repeat, while `count`>0
    a. ~~Set `toBlock`[`toIndex`] to the value of `fromBlock`[`fromIndex`].~~If `fromBlock` is a *Shared Data Block* §2.2.1 then
        i. Let `execution` be the [[CandidateExecution]] field of the surrounding agent's *Agent Record* §3.3.
        ii. Let `eventList` be the [[EventList]] field of the element in `execution`.[[EventLists]] whose [[AgentSignifier]] is *AgentSignifier* §6.3.2.3().
        iii. Let `bytes` be a *List* of length `1` that contains the nondeterministically chosen byte value.
        iv. NOTE: In implementations, `bytes` is the result of a non-atomic read instruction on the underlying hardware. The nondeterminism is a semantic prescription of the *memory model* §6.3 to describe observable behavior of hardware with weak consistency.
        v. Let `readEvent` be ReadSharedMemory{ [[Order]]: **"Unordered"**, [[NoTear]]: `true`, [[Block]]: `fromBlock`, [[ByteIndex]]: `fromIndex`, [[ElementSize]]: `1` }.
        vi. Append `readEvent` to `eventList`.
        vii. Append WriteSharedMemory{ [[Order]]: **"Unordered"**, [[NoTear]]: `true`, [[Block]]: `toBlock`, [[ByteIndex]]: `toIndex`, [[ElementSize]]: `1`, [[Payload]]: `bytes` } to the `eventList`.
        viii. Append { [[Event]]: `readEvent`, [[ChosenValue]]: `bytes` } to `execution`.[[ChosenValues]].
    b. Otherwise, set `toBlock`[`toIndex`] to `fromBlock`[`fromIndex`].
    c. Increment `toIndex` and `fromIndex` each by 1.
    d. Decrement `count` by 1.
8. Return *NormalCompletion*(empty).

2.2.1.2 CreateSharedByteDataBlock(size)

When the abstract operation CreateSharedByteDataBlock is called with integer argument size, the following steps are taken:

1. Assert: `size` >= 0.
2. Let `db` be a new *Shared Data Block* §2.2.1 value consisting of `size` bytes. If it is impossible to create such a *Shared Data Block* §2.2.1, throw a `RangeError` exception.
3. Let `execution` be the [[CandidateExecution]] field of the surrounding agent's *Agent Record* §3.3.
4. Let `eventList` be the [[EventList]] field of the element in `execution`.[[EventLists]] whose [[AgentSignifier]] is *AgentSignifier* §6.3.2.3().
5. Let `zero` be a *List* of length 1 that contains `0`.
6. For each index `i` of `db`:
    a. Append WriteSharedMemory{ [[Order]]: **"Init"**, [[NoTear]]: `true`, [[Block]]: `db`, [[ByteIndex]]: `i`, [[ElementSize]]: `1`, [[Payload]]: `zero` } to `eventList`.
7. Return `db`.

Chapter 3

Executable Code and Execution Contexts (ES7 8)

3.1 Jobs and Job Queues (ES7 8.4)

3.1.1 Forward Progress Guarantees

The forward progress guarantee is provided by PR 522 on ES262.

3.2 Agents (AMENDMENTS)

Add the following properties to the Agent Record (which is provided by PR 522 on ES262):

Table 1: Agent Record Fields

Field name	Value	Meaning
[[Signifier]]	A value that admits equality testing	Uniquely identifies the agent within its agent cluster.
[[LittleEndian]]	A Boolean	`true` if the underlying machine is little-endian, `false` otherwise.
[[IsLockFree1]]	A Boolean	`true` if atomic operations on one-byte values are lock-free, `false` otherwise.
[[IsLockFree2]]	Boolean	`true` if atomic operations on two-byte values are lock-free, `false` otherwise.
[[CandidateExecution]]	A *candidate execution* §6.3.1.5 *Record*	See the Atomics *memory model* §6.3.

Once the values of [[Signifier]], [[IsLockFree1]], and [[IsLockFree2]] have been observed by any agent in the agent cluster they cannot change.

> **Note:** The values of [[IsLockFree1]] and [[IsLockFree2]] are not necessarily determined by the hardware, but may also reflect implementation choices that can vary over time and between ECMAScript

implementations.

There is no [[IsLockFree4]] property: 4-byte atomic operations are always lock-free.

Formally, atomic operations are lock-free if, infinitely often, some atomic operation finishes in a finite number of program steps. In practice, if an atomic operation is implemented with any type of lock the operation is not lock-free. Lock-free does not imply wait-free: there is no upper bound on how many machine steps may be required to complete a lock-free atomic operation.

That an atomic access of size n is lock-free does not imply anything about the (perceived) atomicity of non-atomic accesses of size n, specifically, non-atomic accesses may still be performed as a sequence of several separate memory accesses. See ReadSharedMemory and WriteSharedMemory for details.

3.3 Agent Clusters (NEW)

An agent cluster is a maximal set of agents that can communicate by operating on shared memory.

Note 1: Programs within different agents may share memory by unspecified means. At a minimum, the backing memory for SharedArrayBuffer objects can be shared among the agents in the cluster.

There may be agents that can communicate by message passing that cannot share memory; they are never in the same cluster.

Every agent belongs to exactly one agent cluster.

Note 2: The agents in a cluster need not all be alive at some particular point in time. If agent A creates another agent B, after which A terminates and B creates agent C, the three agents are in the same cluster if A could share some memory with B and B could share some memory with C.

All agents within a cluster must have the same value for the [[LittleEndian]] property in their respective Agent Records.

Note 3: If different agents within an agent cluster have different values of [[LittleEndian]] it becomes hard to use shared memory for multi-byte data.

All agents within a cluster must have the same values for the [[IsLockFree1]] property in their respective Agent Records; similarly for the [[IsLockFree2]] property.

All agents within a cluster must have the same value for the [[CandidateExecution]] in their respective Agent Records.

All agents within a cluster must have different values for the [[Signifier]] property in their respective Agent Records.

An agent cluster is a specification mechanism and need not correspond to any particular artefact of an ECMAScript implementation.

An embedding may deactivate (stop forward progress) or activate (resume forward progress) an agent without the agent's knowledge or cooperation. If the embedding does so, it must not leave some agents in the cluster active while other agents in the cluster are deactivated indefinitely.

Note 4: The purpose of the preceding restriction is to avoid a situation where an agent deadlocks or starves because another agent has been suspended. For example, if a DOM SharedWorker shares memory with a regular worker, and the regular worker is suspended while it holds a lock (because the web page the regular worker is in is pushed into the window history), and the SharedWorker tries to acquire the lock, then the SharedWorker will be blocked until the regular worker wakes up again, if ever. Meanwhile other workers trying to access the SharedWorker from other web pages will starve.

The implication of the restriction is that it will not be possible to share memory between agents that don't belong to the same suspend/wake collective within the embedding.

An embedding may terminate an agent without any of the agent's cluster's other agents' prior knowledge or cooperation. If an agent is terminated not by programmatic action of its own or of another agent in the cluster but by forces external to the cluster, then the embedding must choose one of two strategies: Either terminate all the agents in the cluster, or provide reliable APIs that allow the agents in the cluster to coordinate so that at least one remaining member of the cluster will be able to detect the termination, with the termination data containing enough information to identify the agent that was terminated.

Note 5: Examples of that type of termination are: operating systems or users terminating agents that are running in separate processes; the embedding itself terminating an agent that is running in-process with the other agents when per-agent resource accounting indicates that the agent is runaway.

Note 6: This proposal additionally suggests (see later text) that if termination is signaled then the signal creates a *synchronizes-with* §6.3.1.11 edge in the memory ordering.

Chapter 4

The Global Object (ES7 18)

4.1 Constructor Properties of the Global Object (ES7 18.3)

4.1.1 SharedArrayBuffer

Add a new subsection for SharedArrayBuffer, pointing to the appropriate new section (below).

4.1.2 Atomics

Add a new subsection for Atomics, pointing to the appropriate new section (below).

Chapter 5

Indexed Collections (ES7 22)

5.1 TypedArray Objects (ES7 22.2)

5.1.1 Properties of the *TypedArrayPrototype* object (ES7 22.2.3)

5.1.1.1 *TypedArray*.prototype.set(overloaded [, offset]) (ES7 22.2.3.23)

5.1.1.1.1 *TypedArray*.prototype.set(array [, offset]) (ES7 22.2.3.23.1)

This algorithm is modified as follows:

In the call to *SetValueInBuffer* §6.1.1.5, pass `true` as the fourth argument to indicate that the operation is performed on a TypedArray.

5.1.1.1.2 *TypedArray*.prototype.set(typedArray [, offset]) (ES7 22.2.3.23.2)

This algorithm is modified as follows:

Sets multiple values in this TypedArray, reading the values from the typedArray argument object. The optional offset value indicates the first element index in this TypedArray where values are written. If omitted, it is assumed to be 0.

```
1. Assert: typedArray has a [[TypedArrayName]] internal slot. If it does not, the definition in 22.2.3.23.1 applies.
2. Let targetOffset be the this value.
3. If Type(target) is not Object, throw a TypeError exception.
4. If target does not have a [[TypedArrayName]] internal slot, throw a TypeError exception.
5. Assert: target has a [[ViewedArrayBuffer]] internal slot.
6. Let targetOffset be ? ToInteger(offset).
7. If targetOffset $< 0$, throw a RangeError exception.
8. Let targetBuffer be target.[[ViewedArrayBuffer]].
9. If IsDetachedBuffer(targetBuffer) is true, throw a TypeError exception.
```

```
10. Let targetLength be target.[[ArrayLength]].
11. Let srcBuffer be typedArray.[[ViewedArrayBuffer]].
12. If IsDetachedBuffer(srcBuffer) is true, throw a TypeError exception.
13. Let targetName be the String value of target.[[TypedArrayName]].
14. Let targetType be the String value of the Element Type value in Table 50 for targetName.
15. Let targetElementSize be the Number value of the Element Size value specified in Table 50 for targetName.
16. Let targetByteOffset be target.[[ByteOffset]].
17. Let srcName be the String value of typedArray.[[TypedArrayName]].
18. Let srcType be the String value of the Element Type value in Table 50 for srcName.
19. Let srcElementSize be the Number value of the Element Size value specified in Table 50 for srcName.
20. Let srcLength be typedArray.[[ArrayLength]].
21. Let srcByteOffset be typedArray.[[ByteOffset]].
22. If srcLength + targetOffset > targetLength, throw a RangeError exception.
23. If both IsSharedArrayBuffer §6.2.1.2(srcBuffer) and IsSharedArrayBuffer §6.2.1.2(targetBuffer) are true, then let same be true if srcBuffer.[[ArrayBufferData]] equals targetBuffer.[[ArrayBufferData]]; otherwise let same be SameValue(srcBuffer, targetBuffer).
24. ~~If SameValue(srcBuffer, targetBuffer)~~ same is true, then
    a. Let srcBuffer be ? CloneArrayBuffer(srcBuffer, srcByteOffset, srcLength, ArrayBuffer).
    b. NOTE: ArrayBuffer is used to clone srcBuffer because is it known to not have any observable side-effects.
    c. Let srcByteIndex be 0.
25. Else, let srcByteIndex be srcByteOffset.
26. Let targetByteIndex be targetOffset × targetElementSize + targetByteOffset.
27. Let limit be targetByteIndex + targetElementSize × srcLength.
28. If SameValue(srcType, targetType) is true, then
    a. NOTE: If srcType and targetType are the same, the transfer must be performed in a manner that preserves the bit-level encoding of the source data.
    b. Repeat, while targetByteIndex < limit
       i. Let value be GetValueFromBuffer §6.1.1.3(srcBuffer, srcByteIndex, "Uint8", true).
       ii. Perform SetValueInBuffer §6.1.1.5(targetBuffer, targetByteIndex, "Uint8", true, value).
       iii. ...
29. Else,
    a. Repeat, while targetByteIndex < limit
       i. Let value be GetValueFromBuffer §6.1.1.3(srcBuffer, srcByteIndex, srcType, true).
       ii. Perform SetValueInBuffer §6.1.1.5(targetBuffer, targetByteIndex, targetType, true, value).
       iii. ...
30. Return undefined.
```

5.1.1.1.3 *TypedArray*.prototype.slice(start, end) (ES7 22.2.3.24)

This algorithm is modified as follows:

In the calls to *GetValueFromBuffer* §6.1.1.3 and *SetValueInBuffer* §6.1.1.5, pass `true` as the fourth argument to indicate that the operations are performed on a TypedArray.

5.1.2 The *TypedArray* constructors (ES7 22.2.4)

5.1.2.1 TypedArray (typedArray) (ES7 22.2.4.3)

This algorithm is modified as follows:

In the calls to *GetValueFromBuffer* §6.1.1.3 and *SetValueInBuffer* §6.1.1.5, pass `true` as the fourth argument to indicate that the operations are performed on a TypedArray.

Modify steps 17 and 18 to take shared memory into account, as follows:

```
1. If SameValue(elementType, srcType) is true, then
    a. Let srcLength be typedArray.[[ByteLength]].
    b. If IsSharedArrayBuffer §6.2.1.2(srcData) is false, then
        i. Let data be ? CloneArrayBuffer(srcData, srcByteOffset, srcLength).
    c. Else,
        i. Let data be ? CloneArrayBuffer(srcData, srcByteOffset, srcLength, ArrayBuffer).
2. Else,
    a. If IsSharedArrayBuffer §6.2.1.2(srcData) is false, then
        i. Let bufferConstructor be ? SpeciesConstructor(srcData, ArrayBuffer).
    b. Else,
        i. Let bufferConstructor be ArrayBuffer.
    c. Let data be ? AllocateArrayBuffer(bufferConstructor, byteLength).
```

Chapter 6

Structured Data (ES7 24)

6.1 ArrayBuffer Objects (ES7 24.1)

6.1.1 Abstract Operations for ArrayBuffer (ES7 24.1.1)

6.1.1.1 DetachArrayBuffer (arrayBuffer) (ES7 24.1.1.3)

This algorithm is modified as follows:

The abstract operation DetachArrayBuffer with argument arrayBuffer performs the following steps:

```
1. Assert: Type(arrayBuffer) is Object and it has [[ArrayBufferData]] and [[ArrayBufferByteLength]] internal slots.
2. Assert: IsSharedArrayBuffer §6.2.1.2(arrayBuffer) is false.
3. ...
```

6.1.1.2 RawBytesToNumber(type, rawBytes, isLittleEndian)

This is a new abstract operation.

The abstract operation RawBytesToNumber takes three parameters, a String type, a *List* rawBytes, and a Boolean isLittleEndian.

```
1. If isLittleEndian is false, reverse the order of the elements of rawBytes.
2. If type is "Float32", then
    a. Let value be the byte elements of rawBytes concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary32 value.
    b. If value is an IEEE 754-2008 binary32 NaN value, return the NaN Number value.
    c. Return the Number value that corresponds to value.
3. If type is "Float64", then
```

   a. Let `value` be the byte elements of `rawBytes` concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary64 value.
   b. If `value` is an IEEE 754-2008 binary64 NaN value, return the `NaN` Number value.
   c. Return the Number value that corresponds to `value`.

4. If the first code unit of `type` is **"U"**, then
   a. Let `intValue` be the byte elements of `rawBytes` concatenated and interpreted as a bit string encoding of an unsigned little-endian binary number.
5. Else,
   a. Let `intValue` be the byte elements of `rawBytes` concatenated and interpreted as a bit string encoding of a binary little-endian 2's complement number of bit length `elementSize` × 8.
6. Return the Number value that corresponds to `intValue`.

6.1.1.3 GetValueFromBuffer(arrayBuffer, byteIndex, type, isTypedArray, order [, isLittleEndian]) (ES7 24.1.1.5)

This algorithm is modified as follows:

The abstract operation GetValueFromBuffer takes ~~four~~six parameters, an ArrayBuffer or SharedArrayBuffer arrayBuffer, an integer byteIndex, a String type, a Boolean isTypedArray, a String order, and optionally a Boolean isLittleEndian. This operation performs the following steps:

1. Assert: *IsDetachedBuffer*(`arrayBuffer`) is `false`.
2. Assert: There are sufficient bytes in `arrayBuffer` starting at `byteIndex` to represent a value of `type`.
3. Assert: `byteIndex` is an integer value ≥ 0.
4. Let `block` be `arrayBuffer`.[[ArrayBufferData]].
5. Let `elementSize` be the Number value of the Element Size value specified in *Table 50* for Element Type `type`.
6. ~~Let `rawValue` be a *List* of `elementSize` containing, in order, the `elementSize` sequence of bytes starting with `block`[`byteIndex`].~~If *IsSharedArrayBuffer* §6.2.1.2(`arrayBuffer`) is `true` then
   a. Let `execution` be the [[CandidateExecution]] field of the surrounding agent's *Agent Record* §3.3.
   b. Let `eventList` be the [[EventList]] field of the element in `execution`.[[EventLists]] whose [[AgentSignifier]] is *AgentSignifier* §6.3.2.3().
   c. Let `noTear` be `true` if `isTypedArray` is `true` and `type` is **"Int8"**, **"Uint8"**, **"Int16"**, **"Uint16"**, **"Int32"**, or **"Uint32"**, otherwise `false`.
   d. Let `rawValue` be a *List* of length `elementSize` of nondeterministically chosen byte values.
   e. NOTE: In implementations, `rawValue` is the result of a non-atomic or atomic read instruction on the underlying hardware. The nondeterminism is a semantic prescription of the *memory model* §6.3 to describe observable behavior of hardware with weak consistency.
   f. Let `readEvent` be ReadSharedMemory{ [[Order]]: `order`, [[NoTear]]: `noTear`, [[Block]]: `block`, [[ByteIndex]]: `byteIndex`, [[ElementSize]]: `elementSize` }.
   g. Append `readEvent` to `eventList`.
   h. Append { [[Event]]: `readEvent`, [[ChosenValue]]: `rawValue` } to `execution`.[[ChosenValues]].
7. Else, let `rawValue` be a *List* containing, in order, the `elementSize` sequence of bytes starting with `block`[`byteIndex`].

8. If isLittleEndian is not present, set isLittleEndian to ~~either true or false. The choice is implementation dependent and should be the alternative that is most efficient for the implementation. An implementation must use the same value each time this step is executed and the same value must be used for the corresponding step in the *SetValueInBuffer* §6.1.1.5 abstract operation~~ the value of the [[LittleEndian]] internal slot of the surrounding agent's *Agent Record* §3.3.

9. ~~If isLittleEndian is false, reverse the order of the elements of rawValue.~~Let value be *RawBytesToNumber* §6.1.1.2(type, rawValue, isLittleEndian).

10. ~~If type is **"Float32"**, then~~

    a. ~~Let value be the byte elements of rawValue concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary32 value.~~

    b. ~~If value is an IEEE 754-2008 binary32 NaN value, return the NaN Number value.~~

    c. ~~Return the Number value that corresponds to value.~~

11. ~~If type is **"Float64"**, then~~

    a. ~~Let value be the byte elements of rawValue concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary64 value.~~

    b. ~~If value is an IEEE 754-2008 binary64 NaN value, return the NaN Number value.~~

    c. ~~Return the Number value that corresponds to value.~~

12. ~~If the first code unit of type is **"U"**, then~~

    a. ~~Let intValue be the byte elements of rawValue concatenated and interpreted as a bit string encoding of an unsigned little-endian binary number.~~

13. ~~Else,~~

    a. ~~Let intValue be the byte elements of rawValue concatenated and interpreted as a bit string encoding of a binary little-endian 2's complement number of bit length elementSize × 8.~~

14. ~~Return the Number value that corresponds to intValue.~~ Return value.

All existing uses of the form GetValueFromBuffer(arrayBuffer, byteIndex, type, isLittleEndian) in TypedArray methods and abstract operations are changed to GetValueFromBuffer(arrayBuffer, byteIndex, type, true, **"Unordered"**, isLittleEndian).

All existing uses of the form GetValueFromBuffer(arrayBuffer, byteIndex, type, isLittleEndian) in non-TypedArray methods and abstract operations are changed to GetValueFromBuffer(arrayBuffer, byteIndex, type, false, **"Unordered"**, isLittleEndian).

All existing uses of the form GetValueFromBuffer(arrayBuffer, byteIndex, type) in TypedArray methods and abstract operations are changed to GetValueFromBuffer(arrayBuffer, byteIndex, type, true, **"Unordered"**).

All existing uses of the form GetValueFromBuffer(arrayBuffer, byteIndex, type) in non-TypedArray methods and abstract operations are changed to GetValueFromBuffer(arrayBuffer, byteIndex, type, false, **"Unordered"**).

6.1.1.4 NumberToRawBytes(type, value, isLittleEndian)

The abstract operation NumberToRawBytes takes three parameters, a String type, a Number value, and a Boolean isLittleEndian.

1. If type is **"Float32"**, then

   a. Set rawBytes to a *List* containing the 4 bytes that are the result of converting value to IEEE 754-2008 binary32 format using Round to nearest, ties to even rounding mode. If isLittleEndian is **false**, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If value is NaN, rawValue may be set to any implementation chosen IEEE 754-2008 binary32 format Not-a-Number encoding. An implementation must always choose the same encoding for each implementation distinguishable NaN value.

2. Else if type is **"Float64"**, then
   a. Set rawBytes to a *List* containing the 8 bytes that are the IEEE 754-2008 binary64 format encoding of value. If isLittleEndian is **false**, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If value is NaN, rawValue may be set to any implementation chosen IEEE 754-2008 binary64 format Not-a-Number encoding. An implementation must always choose the same encoding for each implementation distinguishable NaN value.
3. Else,
   a. Let n be the Number value of the Element Size specified in *Table 50* for Element Type type.
   b. Let convOp be the abstract operation named in the Conversion Operation column in *Table 50* for Element Type type.
   c. Let intValue be convOp(value).
   d. If intValue ≥ 0, then
      i. Let rawBytes be a *List* containing the n-byte binary encoding of intValue. If isLittleEndian is **false**, the bytes are ordered in big endian order. Otherwise, the bytes are ordered in little endian order.
   e. Else,
      i. Let rawBytes be a *List* containing the n-byte binary 2's complement encoding of intValue. If isLittleEndian is **false**, the bytes are ordered in big endian order. Otherwise, the bytes are ordered in little endian order.
4. Return rawBytes.

6.1.1.5 SetValueInBuffer(arrayBuffer, byteIndex, type, value, isTypedArray, order [, isLittleEndian]) (ES7 24.1.1.6)

This algorithm is modified as follows:

The abstract operation SetValueInBuffer takes ~~five~~seven parameters, an ArrayBuffer or SharedArrayBuffer arrayBuffer, an integer byteIndex, a String type, a Number value, a Boolean isTypedArray, a String order, and optionally a Boolean isLittleEndian. This operation performs the following steps:

1. Assert: *IsDetachedBuffer*(arrayBuffer) is false.
2. Assert: There are sufficient bytes in arrayBuffer starting at byteIndex to represent a value of type.
3. Assert: byteIndex is an integer value ≥ 0.
4. Assert: *Type*(value) is Number.
5. Let block be arrayBuffer.[[ArrayBufferData]].
6. Assert: block is not undefined.
7. If isLittleEndian is not present, set isLittleEndian to ~~either true or false. The choice is implementation dependent and should be the alternative that is most efficient for the implementation. An implementation must use the same value each time this step is executed and the same value must be used for the corresponding step in the *GetValueFromBuffer* §6.1.1.3 abstract operation~~ the value of the [[LittleEndian]] internal slot of the surrounding agent's *Agent Record* §3.3.

8. ~~If type is **"Float32"**, then~~Let rawBytes be *NumberToRawBytes* §6.1.1.4(type, value, isLittleEndian).
    a. ~~Set rawBytes to a *List* containing the 4 bytes that are the result of converting value to IEEE 754-2008 binary32 format using Round to nearest, ties to even rounding mode. If isLittleEndian is false, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If value is NaN, rawValue may be set to any implementation chosen IEEE 754-2008 binary32 format Not-a-Number encoding. An implementation must always choose the same encoding for each implementation distinguishable NaN value.~~
9. ~~Else if type is **"Float64"**, then~~
    a. ~~Set rawBytes to a *List* containing the 8 bytes that are the IEEE 754-2008 binary64 format encoding of value. If isLittleEndian is false, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If value is NaN, rawValue may be set to any implementation chosen IEEE 754-2008 binary64 format Not-a-Number encoding. An implementation must always choose the same encoding for each implementation distinguishable NaN value.~~
10. ~~Else,~~
    a. ~~Let n be the Number value of the Element Size specified in *Table 50* for Element Type type.~~
    b. ~~Let convOp be the abstract operation named in the Conversion Operation column in *Table 50* for Element Type type.~~
    c. ~~Let intValue be convOp(value).~~
    d. ~~If intValue ≥ 0, then~~
        i. ~~Let rawBytes be a *List* containing the n-byte binary encoding of intValue. If isLittleEndian is false, the bytes are ordered in big endian order. Otherwise, the bytes are ordered in little endian order.~~
    e. ~~Else,~~
        i. ~~Let rawBytes be a *List* containing the n-byte binary 2's complement encoding of intValue. If isLittleEndian is false, the bytes are ordered in big endian order. Otherwise, the bytes are ordered in little endian order.~~
11. ~~Store the individual bytes of rawBytes into block, in order, starting at block[byteIndex].~~If *IsSharedArrayBuffer* §6.2.1.2(arrayBuffer) is true, then
    a. Let execution be the [[CandidateExecution]] field of the surrounding agent's *Agent Record* §3.3.
    b. Let eventList be the [[EventList]] field of the element in execution.[[EventLists]] whose [[AgentSignifier]] is *AgentSignifier* §6.3.2.3().
    c. Let noTear be true if isTypedArray is true and type is **"Int8"**, **"Uint8"**, **"Int16"**, **"Uint16"**, **"Int32"**, or **"Uint32"**, otherwise false.
    d. Append WriteSharedMemory{ [[Order]]: order, [[NoTear]]: noTear, [[Block]]: block, [[ByteIndex]]: byteIndex, [[ElementSize]]: elementSize, [[Payload]]: rawBytes } to eventList.
12. Else, store the individual bytes of rawBytes into block, in order, starting at block[byteIndex].
13. Return *NormalCompletion*(undefined).

All existing uses of the form SetValueInBuffer(arrayBuffer, byteIndex, type, value, isLittleEndian) in TypedArray methods and abstract operations are changed to SetValueInBuffer(arrayBuffer, byteIndex, type, value, true, **"Unordered"**, isLittleEndian).

All existing uses of the form SetValueInBuffer(arrayBuffer, byteIndex, type, value, isLittleEndian) in non-TypedArray methods and abstract operations are changed to SetValueInBuffer(arrayBuffer, byteIndex, type, value, false, **"Unordered"**, isLittleEndian).

All existing uses of the form SetValueInBuffer(arrayBuffer, byteIndex, type, value) in TypedArray methods and abstract operations are changed to SetValueInBuffer(arrayBuffer, byteIndex, type, value, true, **"Unordered"**).

All existing uses of the form SetValueInBuffer(arrayBuffer, byteIndex, type, value) in non-TypedArray methods and abstract operations are changed to SetValueInBuffer(arrayBuffer, byteIndex, type, value, false, **"Unordered"**).

6.1.1.6 GetModifySetValueInBuffer(arrayBuffer, byteIndex, type, value, op [, isLittleEndian])

This is a new abstract operation.

The abstract operation GetModifySetValueInBuffer takes six parameters, An ArrayBuffer or SharedArrayBuffer arrayBuffer, a nonnegative integer byteIndex, a String type, a semantic function op, a Number value, and optionally a Boolean isLittleEndian. This operation performs the following steps given a valid execution execution (see the Atomics *memory model* §6.3):

1. Assert: *IsDetachedBuffer*(`arrayBuffer`) is `false`.
2. Assert: There are sufficient bytes in `arrayBuffer` starting at `byteIndex` to represent a value of `type`.
3. Assert: `byteIndex` is an integer value ≥ 0.
4. Assert: *Type*(`value`) is Number.
5. Let `block` be `arrayBuffer`.[[ArrayBufferData]].
6. Let `elementSize` be the Number value of the Element Size value specified in *Table 50* for Element Type `type`.
7. If `isLittleEndian` is not present, set `isLittleEndian` to the value of the [[LittleEndian]] internal slot of the surrounding agent's *Agent Record* §3.3.
8. Let `rawBytes` be *NumberToRawBytes* §6.1.1.4(`type`, `value`, `isLittleEndian`).
9. Assert: *IsSharedArrayBuffer* §6.2.1.2(`arrayBuffer`) is `true`.
10. Let `execution` be the [[CandidateExecution]] field of the surrounding agent's *Agent Record* §3.3.
11. Let `eventList` be the [[EventList]] field of the element in `execution`.[[AgentOrders]] whose [[AgentSignifier]] is *AgentSignifier* §6.3.2.3().
12. Let `rawBytesRead` be the a *List* of length `elementSize` of nondeterministically chosen byte values.
13. NOTE: In implementations, `rawBytesRead` is the result of a load-link or of an operand of a read-modify-write instruction on the underlying hardware. The nondeterminism is a semantic prescription of the *memory model* §6.3 to describe observable behavior of hardware with weak consistency.
14. Let `rmwEvent` be ReadModifyWriteSharedMemory{ [[Order]]: **"SeqCst"**, [[NoTear]]: `true`, [[Block]]: `block`, [[ByteIndex]]: `byteIndex`, [[ElementSize]]: `elementSize`, [[Payload]]: `rawBytes`, [[ModifyOp]]: `op` }.
15. Append `rmwEvent` to `eventList`.
16. Append { [[Event]]: `rmwEvent`, [[ChosenValue]]: `rawBytesRead` } to `execution`.[[ChosenValues]].
17. Return *RawBytesToNumber* §6.1.1.2(`type`, `rawBytesRead`, `isLittleEndian`).

6.1.2 Properties of the ArrayBuffer Prototype Object (ES7 24.1.4)

6.1.2.1 get ArrayBuffer.prototype.byteLength (ES7 24.1.4.1)

This algorithm is modified as follows:

ArrayBuffer.prototype.byteLength is an accessor property whose set accessor function is `undefined`. Its get accessor function performs the following steps:

1. Let `O` be the `this` value.

2. If *Type*(O) is not Object, throw a TypeError exception.
3. If O does not have an [[ArrayBufferData]] internal slot, throw a TypeError exception.
4. If *IsSharedArrayBuffer* §6.2.1.2(O) is true, throw a TypeError exception.
5. ...

6.1.2.2 ArrayBuffer.prototype.slice (start, end) (ES7 24.1.4.3)

This algorithm is modified as follows:

The following steps are taken:

1. Let O be the this value.
2. If *Type*(O) is not Object, throw a TypeError exception.
3. If O does not have an [[ArrayBufferData]] internal slot, throw a TypeError exception.
4. If *IsSharedArrayBuffer* §6.2.1.2(O) is true, throw a TypeError exception.
5. ...

6.2 SharedArrayBuffer Objects

6.2.1 Abstract Operations for SharedArrayBuffer

6.2.1.1 AllocateSharedArrayBuffer(constructor, byteLength)

The abstract operation AllocateSharedArrayBuffer with arguments constructor and byteLength is used to create a SharedArrayBuffer object. It performs the following steps:

1. Let obj be ? *OrdinaryCreateFromConstructor*(constructor, "$SharedArrayBufferPrototype$", [[ArrayBufferData]], [[ArrayBufferByteLength]]).
2. Assert: byteLength is a nonnegative integer.
3. Let block be ? *CreateSharedByteDataBlock* §2.2.1.2(byteLength).
4. Set obj.[[ArrayBufferData]] to block.
5. Set obj.[[ArrayBufferByteLength]] to byteLength.
6. Return obj.

6.2.1.2 IsSharedArrayBuffer(obj)

IsSharedArrayBuffer tests whether an object that is an ArrayBuffer, a SharedArrayBuffer, or a subtype of either is a SharedArrayBuffer or a subtype of it. It performs the following steps:

1. Assert: *Type*(obj) is Object and it has an [[ArrayBufferData]] internal slot.

2. Let `bufferData` be `obj`.[[ArrayBufferData]].
3. If `bufferData` is `null` or `undefined` then return `false`.
4. If `bufferData` is a *Data Block* then return `false`.
5. Assert: `bufferData` is a *Shared Data Block* §2.2.1.
6. Return `true`.

6.2.1.3 The SharedArrayBuffer Constructor

The SharedArrayBuffer constructor is the *SharedArrayBuffer* intrinsic object and the initial value of the **SharedArrayBuffer** property of the *global object*. When called as a constructor it creates and initializes a new SharedArrayBuffer object. The SharedArrayBuffer constructor is not intended to be called as a function and will throw an exception when called in that manner.

The SharedArrayBuffer constructor is designed to be subclassable. It may be used as the value of an **extends** clause of a class definition. Subclass constructors that intend to inherit the specified SharedArrayBuffer behaviour must include a **super** call to the SharedArrayBuffer constructor to create and initialize subclass instances with the internal state necessary to support the **SharedArrayBuffer.prototype** built-in methods.

> **Note:** Unlike an ArrayBuffer, a SharedArrayBuffer cannot become detached, and its internal [[ArrayBufferData]] slot is never `null` or `undefined`.

6.2.1.3.1 SharedArrayBuffer(length)

SharedArrayBuffer called with argument length performs the following steps:

1. If NewTarget is `undefined`, throw a `TypeError` exception.
2. Let `numberLength` be ? *ToNumber*(`length`).
3. Let `byteLength` be *ToLength*(`numberLength`).
4. If *SameValueZero*(`numberLength`, `byteLength`) is `false`, throw a `RangeError` exception.
5. Return AllocateSharedArrayBuffer(NewTarget, `byteLength`).

6.2.1.4 Properties of the SharedArrayBuffer constructor

The value of the [[Prototype]] internal slot of the SharedArrayBuffer constructor is the intrinsic object *FunctionPrototype* (q.v.).

Besides its **length** property (whose value is 1), the SharedArrayBuffer constructor has the following properties:

6.2.1.4.1 SharedArrayBuffer.prototype

The initial value of SharedArrayBuffer.prototype is the intrinsic object *SharedArrayBufferPrototype* (q.v.).

This property has the attributes { [[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `false` }.

6.2.1.4.2 get SharedArrayBuffer [@@species]

SharedArrayBuffer[@@species] is an accessor property whose set accessor function is `undefined`. Its get accessor function performs the following steps:

1. Return the `this` value.

The value of the name property of this function is " **get [Symbol.species]**".

6.2.1.5 Properties of the SharedArrayBuffer prototype object

The SharedArrayBuffer prototype object is the intrinsic object *SharedArrayBufferPrototype*. The value of the [[Prototype]] internal slot of the SharedArrayBuffer prototype object is the intrinsic object *ObjectPrototype* (19.1.3). The SharedArrayBuffer prototype object is an ordinary object. It does not have an [[ArrayBufferData]] or [[ArrayBufferByteLength]] internal slot.

6.2.1.5.1 get SharedArrayBuffer.prototype.byteLength

SharedArrayBuffer.prototype.byteLength is an accessor property whose set accessor function is `undefined`. Its get accessor function performs the following steps:

1. Let `O` be the `this` value.
2. If *Type*(`O`) is not Object, throw a `TypeError` exception.
3. If `O` does not have a [[ArrayBufferData]] internal slot, throw a `TypeError` exception.
4. If *IsSharedArrayBuffer* §6.2.1.2(`O`) is `false`, throw a `TypeError` exception.
5. Let `length` be `O`.[[ArrayBufferByteLength]].
6. Return `length`.

6.2.1.5.2 SharedArrayBuffer.prototype.constructor

The initial value of **SharedArrayBuffer.prototype.constructor** is the intrinsic object *SharedArrayBuffer*.

6.2.1.5.3 SharedArrayBuffer.prototype.slice(start, end)

The following steps are taken:

1. Let `O` be the `this` value.
2. If *Type*(`O`) is not Object, throw a `TypeError` exception.
3. If `O` does not have an [[ArrayBufferData]] internal slot, throw a `TypeError` exception.
4. If *IsSharedArrayBuffer* §6.2.1.2(`O`) is `false`, throw a `TypeError` exception.
5. Let `len` be `O`.[[ArrayBufferByteLength]].

```
6. Let relativeStart be ? ToInteger(start).
7. If relativeStart < 0, let first be max((len + relativeStart), 0); else let first be min(relativeStart, len).
8. If end is undefined, let relativeEnd be len; else let relativeEnd be ? ToInteger(end).
9. If relativeEnd < 0, let final be max((len + relativeEnd), 0); else let final be min(relativeEnd, len).
10. Let newLen be max(final - first, 0).
11. Let ctor be ? SpeciesConstructor(O, SharedArrayBuffer).
12. Let new be ? Construct(ctor, newLen).
13. If new does not have an [[ArrayBufferData]] internal slot, throw a TypeError exception.
14. If SameValue(new, O) is true, throw a TypeError exception.
15. If new.[[ArrayBufferByteLength]] < newLen, throw a TypeError exception.
16. Let fromBuf be O.[[ArrayBufferData]].
17. Let toBuf be new.[[ArrayBufferData]].
18. Perform CopyDataBlockBytes §2.2.1.1(toBuf, 0, fromBuf, first, newLen).
19. Return new.
```

6.2.1.5.4 SharedArrayBuffer.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the String value **”SharedArrayBuffer”**.

This property has the attributes { [[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `true` }.

6.2.1.6 Properties of the SharedArrayBuffer instances

SharedArrayBuffer instances inherit properties from the SharedArrayBuffer prototype object. SharedArrayBuffer instances each have an [[ArrayBufferData]] internal slot and a [[ArrayBufferByteLength]] internal slot.

> **Note:** SharedArrayBuffer instances, unlike ArrayBuffer instances, are never detached.

6.3 The Atomics Object

The Atomics object is the *Atomics* intrinsic object and the initial value of the **Atomics** property of the *global object*. The Atomics object is a single ordinary object.

The Atomics object provides functions that operate indivisibly (atomically) on shared memory array cells as well as functions that let agents wait for and dispatch primitive events. When used with discipline, the Atomics functions allow multi-agent programs that communicate through shared memory to execute in a well-understood order even on parallel CPUs. The rules that govern shared-memory communication are provided by the memory model, defined below.

The value of the [[Prototype]] internal slot of the Atomics object is the intrinsic object *ObjectPrototype*.

The Atomics object is not a function object. It does not have a [[Construct]] internal method; it is not possible to use the Atomics object as a constructor with the **new** operator. The Atomics object also does not have a [[Call]] internal method; it is not possible to invoke the Atomics object as a function.

6.3.1 Memory Model

Note 1: (Spec draft notes)

What We Talk About When We Talk About Atomicity

Atomicity means two things. One, it means that an action is indivisible. For example, writing a value to memory should write the whole value all at once. This is called access atomicity. Two, it means the timing with which different threads can observe a memory access, i.e., the ordering of memory events. For example, on x86, a memory write becomes visible to all cores other than the writing core at exactly the same time. On ARM, a memory write may become visible to other cores at different times. This is called copy atomicity.

Access atomicity is the easier problem. Hardware provides access atomic instructions that provide the guarantee. Copy atomicity is more difficult. Memory models concern themselves with prescribing some flavor of copy atomicity by ordering memory events. Access atomicity may be thought of as an orthogonal property that is applied on top of the *memory model* §6.3.

Intuitions for the Memory Model

The *memory model* §6.3 proper is a set of relational constraints on a set of memory events. The memory events correspond to read, write, and read-modify-write operations. The usual, intuitive semantic model of programs is operational: a program's meaning is the steps it takes and what each step does, in algorithmic fashion. ECMA262, excluding this section, is specified operationally. The *memory model* §6.3, in contrast, corresponds is a set of axiomatic constraints that must be satisfied: a program's meaning is mathematical objects that satisfy those constraints. Instead of constructing an algorithm that builds up the possible memory values, the model describes the set of all possible allowed memory values of a program directly via relational constraints.

Thought of another way, the operational model of ECMA262 constructs an execution that is correct by construction. The *memory model* §6.3 instead is a filter on all executions. Some of those executions may not be correct, and the correct ones are exactly the ones that are allowed by the *memory model* §6.3.

The model is best understood in two parts: the axiomatic part and the interfacing with the rest of ECMA262.

The axiomatic part is a weak consistency model. It needs to be flexible enough to allow a set of astonishing observable behaviors that are exhibited on weak hardware like ARM and Power and result from compiler transformations. For example, in certain cases, causality is allowed to be violated. At the same time, the *memory model* §6.3 needs to enable programmers to reason about programs. These two needs run counter to each other. The compromise in the community is a guarantee called Sequential Consistency for Data Race Free Programs, or SC-DRF. In informal terms, SC-DRF says that if a program does not have concurrent, non-atomic memory accesses on the same memory locations (no data races), the program semantics is as if it were an interleaving of steps from each individual thread (sequentially consistent). The first goal of this model is to enable SC-DRF reasoning. For non-*data race free* §6.3.1.25 programs, the *memory model* §6.3 aims to give unambiguous meaning to those data races. The second goal of this model is to fully specify the semantics of data races.

To understand the axiomatic model, we start with the input to the model, candidate executions. Assume that there is a set of shared memory events (reads and writes) that correspond to some evaluation of a program. The evaluation gives an initial partial ordering of these events called *agent-order* §6.3.1.7: if an event E was evaluated before D in some thread, then E must come before D in *agent-order* §6.3.1.7. Along with the set of events and *agent-order* §6.3.1.7 is a map from read events to write events called *reads-from* §6.3.1.9. This map says which reads read from which writes during the evaluation. These three things make up a *candidate execution* §6.3.1.5, which is the input to the model. The model's job is to further order those events in such a way that candidate executions that can be observed on actual hardware are deemed valid, i.e., allowed, and executions that cannot be observed are deemed invalid, i.e. disallowed.

The partial order *agent-order* §6.3.1.7 only orders events from the same thread. From this order we will build a new partial order called *happens-before* §6.3.1.12, which additionally relates pairs of correctly

> synchronized events across threads. To build *happens-before* §6.3.1.12, we start by relating atomic reads that *reads-from* §6.3.1.9 atomic writes on the same memory location. These pairs are considered correctly synchronized. The *happens-before* §6.3.1.12 partial order then includes all pairs from *agent-order* §6.3.1.7 and *synchronizes-with* §6.3.1.11, as well as pairs of events that are transitively related. Data races, then, are precisely the events that are not relatable by *happens-before* §6.3.1.12, where one of them is a write, and access the same location.
>
> The SharedArrayBuffer API raises the difficulty that it allows aliased access on the same memory location via different sizes. That is, a read event can *reads-from* §6.3.1.9 multiple write events, resulting in reading a composite value of bytes written by multiple events. Problematically, hardware often cannot guarantee the atomicity of concurrent accesses that happen on overlapping locations. To this end the model only guarantees atomic reads and writes on exactly the same memory *range* §6.3.1.2 to be synchronized. Atomic reads and writes on overlapping locations may behave as if they were non-atomic. The *memory model* §6.3 fully specifies the possible composite values that may be read. Together with the definition of data races, we have a fully defined semantics for data races.
>
> In addition to *happens-before* §6.3.1.12, we need one last constraint on synchronized atomics to enable SC-DRF reasoning. Recall that sequential consistency is an interleaving. Mathematically, this is equivalent to putting events in a *total order* §6.3.1.1 (i.e., a sequence). Thus the model requires that valid executions have a total ordering of correctly synchronized atomic events called *memory-order* §6.3.1.20.
>
> These relations and orders mutually constrain each other so that they are coherent. The coherence constraints roughly correspond to the intuition that a read event cannot read from a write event that's ”too old”, in the sense that there's another write event on the memory location that happens-after the write event that was read from.
>
> Putting all of these together nets us a decision procedure that can decide if a *candidate execution* §6.3.1.5 is valid or not.
>
> The decision procedure alone is not yet enough. How does a set of memory events arise in the first place? How do we connect this decision procedure, which acts on all possible executions, with the evaluation semantics of ECMA262?
>
> To connect the axiomatic semantics to the operational semantics, the operational semantics are made nondeterministic. Instead of constructing a single, valid execution, the nondeterministic operational semantics constructs all candidate executions. This is captured by *GetValueFromBuffer* §6.1.1.3 returning any possible byte value when operating on SharedArrayBuffers. It bears stressing that this is the stuff of semantics and not actual implementations. Implementations will simply return the value that the corresponding hardware load instruction returns. But the hardware, the compiler, or both, may act astonishingly due to the sophistication of their implementation and seemingly read values from e.g., the future, or a branch not taken. One may think of this nondeterminism as treating the reads on shared memory as black boxes (e.g., an opaque cache interface on a CPU). Up front, it reads something, the exact value of which is unknown until later, when we apply the *memory model* §6.3 decision procedure.
>
> With this nondeterminism the evaluation semantics can accumulate events as a side effect of performing operations on SharedArrayBuffers. Events and nondeterministically chosen values are accumulated in the [[EventLists]] and [[ChosenValues]] fields in a *candidate execution* §6.3.1.5 record during *SetValueInBuffer* §6.1.1.5. The model then validates that the chosen values were not completely bogus in addition to all the constraints described above. It bears restressing that this reified *candidate execution* §6.3.1.5 record and validation are purely formal mechanisms. Implementations will simply return what is read by hardware.
>
> So what we talk about when we talk about the meaning of a program that uses SharedArrayBuffer is the set of valid executions. To wit, the evaluation semantics first generates the set of all possible candidate executions, then the *memory model* §6.3 filters out the invalid ones. The remaining executions is the meaning of the program. A program always has at least one valid execution.

The memory consistency model, or *memory model* §6.3, specifies the possible orderings of *Shared Data Block* §2.2.1 events, arising via accessing TypedArray instances backed by a SharedArrayBuffer and via methods on the Atomics object. When the program has no data races (defined below), the ordering of events appears as sequentially consistent, i.e., as an interleaving of actions from each agent. When the program has data races, shared memory operations may appear sequentially inconsistent. For example, programs may exhibit causality-violating behavior and other astonishments. These astonishments arise from compiler transforms and the design of CPUs (e.g., out-of-order execution and speculation). The *memory model* §6.3 defines both the precise conditions under which a program exhibits sequentially consistent behavior as well as the possible values read from data races. To wit, there is no undefined behavior.

The *memory model* §6.3 is defined as relational constraints on events introduced by abstract operations on SharedArrayBuffer or by methods on the Atomics object during an evaluation.

> **Note 2:** This section provides an axiomatic model on events introduced by the abstract operations on SharedArrayBuffers. It bears stressing that the model is not expressible algorithmically, unlike the rest of this specification. The nondeterministic introduction of events by abstract operations is the interface between the operational semantics of ECMAScript evaluation and the axiomatic semantics of the *memory model* §6.3. The semantics of these events is defined by considering graphs of all events in an evaluation. These are neither Static Semantics nor Runtime Semantics. There is no demonstrated algorithmic implementation, but instead a set of constraints that determine if a particular event graph is allowed or disallowed.

6.3.1.1 The Set and Relation Specification Types

The Set type is used to explain a collection of unordered elements. Values of the Set type are simple collections of elements, where no element appears more than once. Elements may be added to and removed from Sets. Sets may be unioned, intersected, or subtracted from each other.

The Relation type is used to explain constraints on Sets. Values of the Relation type are Sets of ordered pairs of values from its value domain. For example, a Relation on events is a set of ordered pairs of events. For a Relation R and two values a and b in the value domain of R, a R b is shorthand for saying the ordered pair (a, b) is a member of R. A Relation is least with respect to some conditions when it is the smallest Relation that satisfies those conditions.

A strict partial order is a Relation value R that satisfies the following conditions.

1. For all `a`, `b`, and `c` in `R`'s domain:
   a. It is not the case `a R a`, and
   b. If `a R b` and `b R c` then `a R c`.

> **Note 1:** The two properties above are called, in order, irreflexivity and transitivity.

A total order is a Relation value R that satisfies the following conditions.

1. For all `a`, `b`, and `c` in `R`'s domain:
   a. `a R b` or `b R a`, and
   b. If `a R b` and `b R a` then `a` is `b`, and
   c. If `a R b` and `b R c` then `a R c`.

> **Note 2:** The three properties above are called, in order, totality, antisymmetry, and transitivity.

Formal Integration

Definition 6.3.1 (Sets Relation). *Given two sets X and Y, a binary relation R over X and Y is defined as*

$$R \subseteq X \times Y$$

Moreover, given $x \in X$ and $y \in Y$, the predicate $R(x, y)$ holds iff $(x, y) \in R$.

□

Definition 6.3.2 (Reflexive Relation). *Given a set X, and a binary relation R over X is reflexive when*

$$\forall x, x' \in X.\ R(x, x')$$

holds.

□

Definition 6.3.3 (Transitive Relation). *Given a set X, and a binary relation R over X is transitive when*

$$\forall x, x', x'' \in X.\ (R(x, x') \wedge R(x', x'')) \rightarrow R(x, x'')$$

holds.

□

Definition 6.3.4 (Antisymmetric Relation). *Given a set X, and a binary relation R over X is antisymmetric when*

$$\forall x, x' \in X.\ (R(x, x') \wedge R(x', x)) \rightarrow (x = x')$$

holds.

□

Definition 6.3.5 (Total Relation). *Given a set X, and a binary relation R over X is total when*

$$\forall x, x' \in X.\ R(x, x') \vee R(x', x)$$

holds.

□

Definition 6.3.6 (Partial Order). *Given a binary relation R over X, the partial order property $O^P(R)$ holds when R is Reflexive, Transitive, and Antisymmetric.*

□

Definition 6.3.7 (Total Order). *Given a binary relation R over X, the total order property $O^T(R)$ holds when R is Transitive, Antisymmetric, and Total.*

□

6.3.1.2 Fundamentals

Shared memory accesses (reads and writes) are divided into two groups, atomic accesses and data accesses, defined below. Atomic accesses are sequentially consistent, i.e., there is a total ordering of events agreed upon by all agents in an agent cluster. Non-atomic accesses do not have a total ordering agreed upon by all agents, i.e., unordered.

> **Note 1:** No orderings weaker than sequentially consistent and stronger than unordered, such as release-acquire, are supported.

A Shared Data Block event (or simply event) is either a ReadSharedMemory, WriteSharedMemory, or ReadModify-WriteSharedMemory *Record*.

Table 2: ReadSharedMemory Event Fields

Field	Value	Meaning
[[Order]]	**"SeqCst"**, **"Unordered"**, or **"Init"**.	The weakest ordering guaranteed by the *memory model* §6.3 for the event.
[[NoTear]]	A Boolean	Whether this event is allowed to read from multiple write events on equal range as this event.
[[Block]]	A *Shared Data Block* §2.2.1	The block the event operates on.
[[ByteIndex]]	A nonnegative integer	The byte address of the read in [[Block]].
[[ElementSize]]	A nonnegative integer	The size of the read.

Table 3: WriteSharedMemory Event Fields

Field	Value	Meaning
[[Order]]	**"SeqCst"**, **"Unordered"**, or **"Init"**.	The weakest ordering guaranteed by the *memory model* §6.3 for the event.
[[NoTear]]	A Boolean	Whether this event is allowed to be read from multiple read events with equal range as this event.
[[Block]]	A *Shared Data Block* §2.2.1	The block the event operates on.
[[ByteIndex]]	A nonnegative integer	The byte address of the write in [[Block]].
[[ElementSize]]	A nonnegative integer	The size of the write.
[[Payload]]	A *List*	The *List* of byte values to be read by other events.

Table 4: ReadModifyWriteSharedMemory Event Fields

Field	Value	Meaning
[[Order]]	**"SeqCst"**	Read-modify-write events are always sequentially consistent.
[[NoTear]]	`true`	Read-modify-write events cannot tear.
[[Block]]	A *Shared Data Block* §2.2.1	The block the event operates on.
[[ByteIndex]]	A nonnegative integer	The byte address of the read-modify-write in [[Block]].
[[ElementSize]]	A nonnegative integer	The size of the read-modify-write.
[[Payload]]	A *List*	The *List* of byte values to be passed to [[ModifyOp]].
[[ModifyOp]]	A semantic function	A pure semantic function that returns a modified *List* of byte values from a read *List* of byte values and [[Payload]].

These events are introduced by abstract operations or by methods on the Atomics object.

Additionally, *Shared Data Block* §2.2.1 events include host-specific events.

Let the range of a ReadSharedMemory, WriteSharedMemory, or ReadModifyWriteSharedMemory event be the Set of contiguous integers from its byteIndex to byteIndex+elementSize-1. Two events' ranges are equal when the events have the same block, and the ranges are element-wise equal. Two events' ranges are overlapping when the events have the same block, the ranges are not equal and their intersection is non-empty. Two events' ranges are disjoint when the events do not have the same block or their ranges are neither equal nor overlapping.

> **Note 2:** Examples of host-specific synchronizing events that should be accounted for are: sending a SharedArrayBuffer from one agent to another (e.g., by **postMessage** in a browser), starting and stopping agents, and communicating within the agent cluster via channels other than shared memory. It is assumed those events are appended to *agent-order* §6.3.1.7 during evaluation like the other SharedArrayBuffer events.

Shared Data Block §2.2.1 events are ordered within candidate executions by the relations defined mutually recursively below.

Formal Integration

A *Memory Operation*, also called *Shared Data Block Event*, identify the set of operations that can be performed on shared memory.

Definition 6.3.8 (Memory Operation). *A Memory Operation is a tuple* $\langle O, A, R, B, M, D, S\rangle$ *where:*

- $O \in \{Read\ (R), Write\ (W), ReadModifyWrite\ (M)\}$ *is the operation;*
- $A \in \{Atomic\ (At), Non\text{-}Atomic\ (Na)\}$ *is the atomicity attribute;*
- $R \in \{Init(I), SeqCst(SC), Unordered(U)\}$ *is the order attribute;*
- B *is the Shared Data Block;*
- $M \in \mathbb{N}^n \ : \ \forall_{m \in \{min(\rho)..max(\rho)\}} m \in \rho$ *is the set of integers representing the memory addresses of the operation in* B*;*
- D *is the read value;*
- S *is the write value.*

A Memory Operation $o := \langle O, A, R, \{i, \ldots, j\}, D, S\rangle$ *can be also represented as* $O_A[i, j](D, S)$*, where* D *or* S *can be omitted, respectively, when* $O = R$ *and* $O = W$*.*

□

The allowed memory operations are represented as follows:

- $R_N b[i, j](v)$ non-atomic read-only operations: R m[i..j] = v;
- $W_N b[i, j](w)$ non-atomic write-only operations: W m[i..j] = w;
- $R_A b[i, j](v)$ atomic read-only operations: atomic R m[i..j] = v;
- $W_A b[i, j](w)$ atomic write-only operations: atomic W m[i..j] = w; and
- $M_A b[i, j](v, w)$ atomic read-modify-write operations: atomic RMW m[i..j] = v/w.

6.3.1.3 Agent Events Records

An Agent Events Record is a *Record* with the following fields.

Table 5: Agent Events Record Fields

Field	Value	Meaning
[[AgentSignifier]]	A value that admits equality testing	The agent whose evaluation resulted in this ordering.
[[EventList]]	A *List* of *Shared Data Block* §2.2.1 events	Events are appended to the list during evaluation.

Formal Integration

A *Thread Execution* is defined as a set of ordered memory events that perform a specific memory operation.

Definition 6.3.9 (Thread Execution). *A thread execution is a tuple* $\langle E, L, PO, DD\rangle$ *where:*

- E *is a set of memory events;*
- $L : (E \rightarrow A)$ *is a labeling that relates each event to a Memory Operation;*
- $PO \subseteq E \times E$ *is the program order total order relation;*
- $DD \subseteq E \times E$ *where* $DD \subseteq PO$ *is the data dependency relation.*

Specifically,

- $PO(e, e')$ *holds when* e *precedes* e'*;*
- $DD(e, e')$ *holds when* e *depends from* e'*;*

□

Definition 6.3.10 (Event Range). *Given a Thread Execution* $t := \langle E, L, PO, DD\rangle$*, a Memory Event* $e \in E$*, and a Memory Operation* $o := \langle O, A, M, D, S\rangle$*, such that* $L(e) = a$*. The Range of* e *is defined as*

$$Range(e) := M$$

□

Definition 6.3.11 (Atomic Event). *Given a Thread Execution* $t := \langle E, L, PO, DD\rangle$*, a Memory Event* $e \in E$*, and a Memory Operation* $o := \langle O, A, M, D, S\rangle$*, such that* $L(e) = a$*. The Atomic predicate over* e *is defined as*

$$Atomic(e) := (A = At)$$

□

Definition 6.3.12 (Read Event). *Given a Thread Execution* $t := \langle E, L, PO, DD\rangle$*, a Memory Event* $e \in E$*, and a Memory Operation* $o := \langle O, A, M, D, S\rangle$*, such that* $L(e) = a$*. The Read predicate over* e *is defined as*

$$Read(e) := (O = R)$$

□

Definition 6.3.13 (Write Event). *Given a Thread Execution* $t := \langle E, L, PO, DD\rangle$*, a Memory Event* $e \in E$*, and a Memory Operation* $o := \langle O, A, M, D, S\rangle$*, such that* $L(e) = o$*. The Write predicate over* e *is defined as*

$$Write(e) := (O = W)$$

□

Definition 6.3.14 (ReadModifyWrite Event). *Given a Thread Execution* $t := \langle E, L, PO, DD \rangle$*, a Memory Event* $e \in E$*, and a Memory Operation* $o := \langle O, A, M, D, S \rangle$*, such that* $L(e) = o$*. The ReadModifyWrite predicate over e is defined as*

$$\mathit{Modify}(e) := (O = M)$$

□

Definition 6.3.15 (Read or ReadModifyWrite Event). *The Read or ReadModifyWrite predicate over an event e is defined as*

$$\mathit{RoM}(e) := \mathit{Read}(e) \vee \mathit{Modify}(e)$$

□

Definition 6.3.16 (Write or ReadModifyWrite Event). *The Write or ReadModifyWrite predicate over an event e is defined as*

$$\mathit{WoM}(e) := \mathit{Write}(e) \vee \mathit{Modify}(e)$$

□

Definition 6.3.17 (Init Event). *Given a Thread Execution* $t := \langle E, L, PO, DD \rangle$*, a Memory Event* $e \in E$*, and a Memory Operation* $o := \langle O, A, R, M, D, S \rangle$*, such that* $L(e) = o$*. The Init predicate over e is defined as*

$$\mathit{Init}(e) := (R = I)$$

□

Definition 6.3.18 (Unordered Event). *Given a Thread Execution* $t := \langle E, L, PO, DD \rangle$*, a Memory Event* $e \in E$*, and a Memory Operation* $o := \langle O, A, R, M, D, S \rangle$*, such that* $L(e) = o$*. The Unordered predicate over e is defined as*

$$\mathit{SeqCst}(e) := (R = U)$$

□

6.3.1.4 Chosen Value Records

A Chosen Value Record is a *Record* with the following fields.

Table 6: Chosen Value Record Fields

Field	Value	Meaning
[[Event]]	A *Shared Data Block event* §6.3.1.2	The ReadSharedMemory or ReadModifyWriteSharedMemory event that was introduced for this chosen value.
[[ChosenValue]]	A *List* of byte values	The bytes that were nondeterministically chosen during evaluation.

6.3.1.5 Candidate Executions

A candidate execution of the evaluation of an agent cluster is a *Record* with the following fields.

An empty candidate execution execution is a *Record* whose fields are empty Lists and Relations defined below, and whose [[EventLists]] field is further initialized by the following steps.

Table 7: Candidate Execution *Record* Fields

Field	**Value**	**Meaning**
[[EventLists]]	A *List* of Agent Events Records.	Maps an agent to Lists of events appended during the evaluation.
[[ChosenValues]]	A *List* of Chosen Value Records.	Maps ReadSharedMemory or ReadModifyWriteSharedMemory events to the *List* of byte values chosen during the evaluation.
[[AgentOrder]]	An *agent-order* §6.3.1.7 *Relation* §6.3.1.1.	Defined below.
[[ReadsBytesFrom]]	A *reads-bytes-from* §6.3.1.8 *Relation* §6.3.1.1.	Defined below.
[[ReadsFrom]]	A *reads-from* §6.3.1.9 *Relation* §6.3.1.1.	Defined below.
[[HostSynchronizesWith]]	A *host-synchronizes-with* §6.3.1.10 *Relation* §6.3.1.1.	Defined below.
[[SynchronizesWith]]	A *synchronizes-with* §6.3.1.11 *Relation* §6.3.1.1.	Defined below.
[[HappensBefore]]	A *happens-before* §6.3.1.12 *Relation* §6.3.1.1.	Defined below.
[[DependsOn]]	A *depends-on* §6.3.1.13 *Relation* §6.3.1.1.	Defined below.

```
1. For each agent agent in the agent cluster:
    a. Let AR be the Agent Record §3.3 of agent.
    b. Let emptyList be an empty List.
    c. Append { [[AgentSignifier]]: AR.[[Signifier]], [[EventList]]: emptyList } to execution.[[EventLists]].
```

Formal Integration

Definition 6.3.19 (Union of Threads). *Given the threads $t_1, t_2, \ldots, t_n$, their union $T := t_1 \cup t_2 \cup \ldots \cup t_n$ is defined as*

$$T := \langle E, L, PO, DD \rangle$$

where

- $E := E_1 \cup E_2 \cup \ldots E_n$;
- $L := (E \to A) \; : \; L(e) = a \leftrightarrow \exists i.\ e \in E_i \wedge L_i(e) = a$;
- $PO := PO_1 \cup PO_2 \cup \ldots \cup PO_n$;
- $DD := DD_1 \cup DD_2 \cup \ldots \cup DD_n$.

□

Definition 6.3.20 (Candidate Execution). *Given a threads union $T := \langle E, L, PO, DD \rangle$, a Candidate Execution over T is defined as*

$$CE(T) := \langle AO, HSW, SW, HB, RBF, DO \rangle$$

where

- $AO \subseteq E \times E$ *is the agent order relation;*
- $HSW \subseteq E \times E$ *is the host synchronizes with relation;*
- $SW \subseteq E \times E$ *is the synchronizes with relation;*
- $HB \subseteq E \times E$ *is the happens before relation;*
- $RBF \subseteq E \times E$ *is the reads bytes from relation;*
- $DO \subseteq E \times E$ *is the depends on relation.*

□

6.3.1.6 EventSet(execution)

The abstract operation EventSet takes one argument, a *candidate execution* §6.3.1.5 execution, and performs the following steps.

```
1. Let events be an empty Set.
2. For each element eventList in execution.[[EventLists]]:
    a. For each event E in eventList:
        i. Add E to events.
3. Return events.
```

6.3.1.7 agent-order

For a *candidate execution* §6.3.1.5 execution, a *Relation* §6.3.1.1 that satisfies the following conditions.

1. For each pair of events E and D in *EventSet* §6.3.1.6(execution):
   a. For each element eventList in execution.[[EventLists]]:
      i. If E and D are in eventList and E is before D in *List* order of eventList then E is *agent-order* §6.3.1.7 before D.

> **Note:** Each agent introduces events in a per-agent *total order* §6.3.1.1 during the evaluation. This is the union of those total orders.

Formal Integration

Definition 6.3.21 (Agent Order). *Given a threads union* $T := \langle E, L, PO, DD \rangle$, *and a Candidate Execution* $CE(T) := \langle AO,\ HSW,\ SW,\ HB,\ RBF,\ DO \rangle$, *the Agent Order relation* AO *is defined as*

$$AO := PO$$

□

6.3.1.8 reads-bytes-from

For a *candidate execution* §6.3.1.5 execution, execution.[[ReadsBytesFrom]] is a *Relation* §6.3.1.1 that satisfies the following conditions.

1. Let *reads-bytes-from* §6.3.1.8 be execution.[[ReadsBytesFrom]].
2. For each ReadSharedMemory or ReadModifyWriteSharedMemory event R in *EventSet* §6.3.1.6(execution):
   a. There is a *List* of length equal to R.[[ElementSize]] of WriteSharedMemory or ReadModifyWriteSharedMemory events Ws such that R *reads-bytes-from* §6.3.1.8 Ws.
   b. Let byteLocation be R.[[ByteIndex]].
   c. For each element W of Ws in *List* order:
      i. W has byteLocation in its *range* §6.3.1.2.
      ii. Increment byteLocation by 1.

Formal Integration

Definition 6.3.22 (Reads Bytes From). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the reads bytes from relation RBF is defined as*

$$\begin{aligned} RBF \subseteq E \times E \;:\; & \forall r, w \in E.\ RoM(r) \wedge WoM(w) \wedge 3' \\ & 3' := (Range(w) \subseteq Range(r)) \rightarrow RBF(r, w) \end{aligned}$$

□

Definition 6.3.23 (Reads Bytes From (Abstraction)).

$$RBF := \sigma_{RBF}(ER, HB)$$

□

6.3.1.9 reads-from

For a *candidate execution* §6.3.1.5 execution, execution.[[ReadsFrom]] is a *Relation* §6.3.1.1 that satisfies the following conditions.

1. Let *reads-from* §6.3.1.9 be `execution`.[[ReadsFrom]].
2. Let *reads-bytes-from* §6.3.1.8 be `execution`.[[ReadsBytesFrom]].
3. For each pair of events `R` and `W` in *EventSet* §6.3.1.6(`execution`):
   a. If `R` *reads-bytes-from* §6.3.1.8 a *List* of WriteSharedMemory events `Ws` that contains `W` then `R` *reads-from* §6.3.1.9 `W`.

Formal Integration

Definition 6.3.24 (Reads From). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the Happens Before relation RF is defined as*

$$RF \subseteq E \times E \;:\; \forall r, w \in E.\ RBF(r, w) \rightarrow RF(r, w) \qquad (3.)$$

□

Definition 6.3.25 (Reads From (Abstraction)).

$$RF := RBF$$

□

6.3.1.10 host-synchronizes-with

For a *candidate execution* §6.3.1.5 execution, a host-provided *strict partial order* §6.3.1.1 on host-specific events in *EventSet* §6.3.1.6(execution).

> **Note:** The host-synchronizes-with relation allows the host to provide additional synchronization mechanisms, such as **postMessage** between HTML workers.

Formal Integration

6.3.1.11 synchronizes-with

For a *candidate execution* §6.3.1.5 execution, execution.[[SynchronizesWith]] is a *Relation* §6.3.1.1 that satisfies the following conditions.

```
1. Let synchronizes-with §6.3.1.11 be execution.[[SynchronizesWith]].
2. Let reads-from §6.3.1.9 be execution.[[ReadsFrom]].
3. Let host-synchronizes-with §6.3.1.10 be execution.[[HostSynchronizesWith]].
4. For each pair of events R and W in EventSet §6.3.1.6(execution) such that R.[[Order]] is "SeqCst" and R reads-from §6.3.1.9 W:
   a. Assert: R is a ReadSharedMemory or ReadModifyWriteSharedMemory event.
   b. Assert: W is a WriteSharedMemory or ReadModifyWriteSharedMemory event.
   c. If W.[[Order]] is "SeqCst" and R and W have equal ranges then W synchronizes-with §6.3.1.11 R.
   d. Otherwise, if W has order "Init" then
      i. Let allInitReads be true.
      ii. For each event V such that R reads-from §6.3.1.9 V:
         A. If V.[[Order]] is not "Init" then set allInitReads to false.
      iii. If allInitReads is true then W synchronizes-with §6.3.1.11 R.
5. For each pair of events E and D in EventSet §6.3.1.6(execution):
   a. If E host-synchronizes-with §6.3.1.10 D then E synchronizes-with §6.3.1.11 D.
```

> **Note 1:** Owing to convention, write events synchronizes-with read events, instead of read events synchronizes-with write events.

> **Note 2:** Not all **"SeqCst"** events related by *reads-from* §6.3.1.9 are related by synchronizes-with. Only events that also have equal ranges are related by synchronizes-with.

> **Note 3:** For an event R and a event W such W synchronizes-with R, R may *reads-from* §6.3.1.9 other writes than W.

Formal Integration

Definition 6.3.26 (Synchronizes With). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO$, HSW, SW, HB, RBF, $DO\rangle$, the synchronizes with relation SW is defined as*

$$SW \subseteq E \times E \;:\; 4 \wedge 5$$

$$4 := \forall r, w \in E.\ 4' \rightarrow (4_c \wedge 4_d)) \tag{4.}$$

$$4' := SeqCst(r) \wedge RF(r, w) \tag{4.}$$

$$4_{c'} := SeqCst(w) \wedge (Range(r) = Range(w)) \tag{4c.}$$

$$4_c := 4_{c'} \rightarrow SW(w, r) \tag{4c.}$$

$$4_d := (\neg 4_{c'} \wedge Init(w)) \rightarrow 4_{di-iii} \tag{4d.}$$

$$4_{di-iii} := (\forall v \in E.\ RF(r, v) \leftrightarrow Init(v)) \rightarrow SW(w, r) \tag{4di-iii.}$$

$$5 := \forall e, d \in E.\ (HSW(e, d) \rightarrow SW(e, d)) \tag{5.}$$

Revised:

$$SW \subseteq E \times E \;:\; 4 \wedge 5$$

$$4 := \forall r, w \in E.\ 4' \rightarrow (4_c \wedge 4_d)) \tag{4.}$$

$$4' := SeqCst(r) \wedge RF(r, w) \tag{4.}$$

$$4_{c'} := SeqCst(w) \wedge (Range(r) = Range(w)) \tag{4c.}$$

$$4_c := 4_{c'} \rightarrow SW(w, r) \tag{4c.}$$

$$4_d := (\neg 4_{c'} \wedge Init(w)) \rightarrow 4_{di-iii} \tag{4d.}$$

$$4_{di-iii} := (\forall v \in E.\ RF(r, v) \leftrightarrow Init(v)) \rightarrow SW(w, r) \tag{4di-iii.}$$

$$5 := \forall e, d \in E.\ (HSW(e, d) \rightarrow SW(e, d)) \tag{5.}$$

Property:

$$P_{SW} := \forall r, w \in E.\ 4' \rightarrow 4_{a-b}$$

$$4' := SeqCst(r) \wedge RF(r, w) \tag{4.}$$

$$4_{a-b} := RoM(r) \wedge WoM(w) \tag{4a-b.}$$

□

Definition 6.3.27 (Synchronizes With (Abstraction)).

$$SW := \sigma_{SW}(ER, RF) \cup HSW$$

□

Definition 6.3.28 (Synchronizes With (from [FSAB16])). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO$, HSW, SW, HB, RBF, $DO\rangle$, the synchronizes with relation SW is defined as*

$$SW \subseteq RF \;:\; \forall e, e' \in E.\ SW(e, e') \leftrightarrow (Atomic(e) \wedge Atomic(e')) \wedge (Range(e) = Range(e'))$$

□

6.3.1.12 happens-before

For a *candidate execution* §6.3.1.5 execution, execution.[[HappensBefore]] is the least *strict partial order* §6.3.1.1 that satisfies the following conditions.

1. Let *happens-before* §6.3.1.12 be `execution`.[[HappensBefore]].
2. Let *agent-order* §6.3.1.7 be `execution`.[[AgentOrder]].
3. Let *synchronizes-with* §6.3.1.11 be `execution`.[[SynchronizesWith]].
4. For each pair of events `E` and `D` in *EventSet* §6.3.1.6(`execution`):
   a. If `E` is *agent-order* §6.3.1.7 before `D` then `E` *happens-before* §6.3.1.12 `D`.
   b. If `E` *synchronizes-with* §6.3.1.11 `D` then `E` *happens-before* §6.3.1.12 `D`.
   c. If `E`.[[Order]] is **"Init"** and `E` and `D` have overlapping ranges then:
      i. Assert: `D`.[[Order]] is not **"Init"**.
      ii. `E` *happens-before* §6.3.1.12 `D`.
   d. If there is an event `F` such that `E` *happens-before* §6.3.1.12 `F` and `F` *happens-before* §6.3.1.12 `D` then `E` *happens-before* §6.3.1.12 `D`.

> **Note:** Because happens-before is a superset of *agent-order* §6.3.1.7, candidate executions are consistent with the single-thread evaluation semantics of ECMA262.

Formal Integration

Definition 6.3.29 (Happens Before). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the Happens Before relation HB is defined as*

$$
\begin{aligned}
HB \subseteq E \times E \ &: \ \forall e, d \in E.(4_a \wedge 4_b \wedge 4_c \wedge 4_d) && (4.)\\
4_a &:= PO(e,d) \rightarrow HB(e,d) && (4a.)\\
4_b &:= SW(e,d) \rightarrow HB(e,d) && (4b.)\\
4_c &:= (Init(e) \wedge (Range(e) \cap Range(d) \neq \emptyset)) \rightarrow 4_{ci-ii} && (4c.)\\
4_{ci-ii} &:= \neg Init(d) \rightarrow HB(e,d) && (4ci\text{-}ii.)\\
4_d &:= \forall f \in E.\ (HB(e,f) \wedge HB(f,d)) \rightarrow HB(e,d) && (4d.)
\end{aligned}
$$

Revised interpretation:

$$
\begin{aligned}
HB \subseteq E \times E \ &: \ HB'* && (4c.)\\
HB' \subseteq E \times E \ &: \ \forall e, d \in E.\ HB'(e,d) \leftrightarrow (4_a \vee 4_b \vee 4_c) && (4.)\\
4_a &:= PO(e,d) && (4a.)\\
4_b &:= SW(e,d) && (4b.)\\
4_c &:= Init(e) \wedge (Range(e) \cap Range(d) \neq \emptyset) && (4c.)
\end{aligned}
$$

Property:

$$P_{HB} := \forall e, d \in E.\ 4_c \rightarrow 4_{ci} \tag{4ci.}$$

$$4_c := \mathit{Init}(e) \wedge (\mathit{Range}(e) \cap \mathit{Range}(d) \neq \emptyset) \tag{4c.}$$

$$4_{ci} := \neg \mathit{Init}(d) \tag{4ci.}$$

□

Definition 6.3.30 (Happens Before (Abstraction)).

$$HB := AO \cup SW \cup \sigma_{HB}(ER)$$

□

Definition 6.3.31 (Happens Before (from [FSAB16])). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO,\ HSW,\ SW,\ HB,\ RBF,\ DO \rangle$, the Happens Before relation HB is defined as*

$$HP := (PO \cup SW)*$$

□

6.3.1.13 depends-on

For a *candidate execution* §6.3.1.5 execution, a host-provided *strict partial order* §6.3.1.1 on events in *EventSet* §6.3.1.6(execution) that captures semantic data and control dependencies between events.

> **Note:** The dependence relation may differ between implementations of ECMAScript. For instance, an implementation may provide a syntactic dependence relation that overapproximates semantic dependences.

Formal Integration

Definition 6.3.32 (Data Dependency in a Thread Execution (from [FSAB16])). *Given a Thread Execution $t := \langle E, L, PO, DD \rangle$, the data dependency relation is defined as*

$$\begin{aligned} DD \subseteq E \times E.\ \forall e, e' \in E.\ & (\mathit{Read}(e) \wedge \mathit{Atomic}(e) \wedge PO(e, e') \rightarrow DD(e', e)) \wedge \\ & (\mathit{Write}(e') \wedge \mathit{Atomic}(e') \wedge PO(e, e') \rightarrow DD(e', e)) \end{aligned}$$

□

6.3.1.14 ComposeWriteEventBytes(execution, block, byteIndex, Ws)

The abstract operation ComposeWriteEventBytes takes four arguments, a *candidate execution* §6.3.1.5 execution, a *Shared Data Block* §2.2.1 block, a nonnegative integer byteIndex, and a *List* Ws, and performs the following steps.

```
1. Let byteLocation be byteIndex.
2. Let bytesRead be an empty List.
3. For each element W of Ws in List order:
   a. Assert: W has byteLocation in its range §6.3.1.2.
   b. If W is a WriteSharedMemory event then
      i. Let byte be W.[[Payload]][byteLocation].
   c. Else,
      i. Assert: W is a ReadModifyWriteSharedMemory event.
      ii. Let bytesRead be ValueOfReadEvent §6.3.1.15(execution, W).
      iii. Let bytesModified be W.[[ModifyOp]](bytesRead, W.[[Payload]]).
      iv. Let byte be bytesModified[byteLocation].
   d. Append byte to bytesRead.
   e. Increment byteLocation by 1.
4. Return bytesRead.
```

> **Note 1:** The semantic function [[ModifyOp]] is given by the function properties on the Atomics object that introduce ReadModifyWriteSharedMemory events.

> **Note 2:** This subroutine composes a *List* of write events into a *List* of byte values. It is used in the event semantics of ReadSharedMemory and ReadModifyWriteSharedMemory events.

Formal Integration

Definition 6.3.33 (Compose Write Event Bytes). □

6.3.1.15 ValueOfReadEvent(execution, R)

The abstract operation ValueOfReadEvent takes two arguments, a *candidate execution* §6.3.1.5 execution and a *Shared Data Block event* §6.3.1.2 R, and performs the following steps

```
1. Assert: R is a ReadSharedMemory or ReadModifyWriteSharedMemory event.
2. Let reads-bytes-from §6.3.1.8 be execution.[[ReadsBytesFrom]].
```

3. Let Ws be the *List* of events such that R *reads-bytes-from* §6.3.1.8 Ws.
4. Assert: Ws has length equal to R.[[ElementSize]].
5. Return *ComposeWriteEventBytes* §6.3.1.14(execution, R.[[Block]], R.[[ByteIndex]], Ws).

Formal Integration

Definition 6.3.34 (Value of Read Event). *Given a threads union T, a Candidate Execution $CE(T) := \langle AO$, HSW, SW, HB, RBF, $DO\rangle$, and an event $r \in E$, the Value of Read Event function VRE over $CE(T), r$ is defined as*

$$VRE(CE(T), r, W_s) := CWEB(CE(T), r, W_s := \{w \in E \mid RBF(r, w)\})$$

□

6.3.1.16 Valid Chosen Reads

A *candidate execution* §6.3.1.5 execution has valid chosen reads if the following conditions hold.

1. For each ReadSharedMemory and ReadModifyWriteSharedMemory event R in *EventSet* §6.3.1.6(execution):
    a. Let chosenValue be the element of execution.[[ChosenValues]] whose [[Event]] field is R.
    b. chosenValue is equal to *ValueOfReadEvent* §6.3.1.15(execution, R).

Formal Integration

Definition 6.3.35 (Valid Chosen Reads). □

6.3.1.17 Coherent Reads

A *candidate execution* §6.3.1.5 execution has coherent reads if the following conditions hold.

1. Let *happens-before* §6.3.1.12 be execution.[[HappensBefore]].
2. For each ReadSharedMemory or ReadModifyWriteSharedMemory event R in *EventSet* §6.3.1.6(execution):

a. Let Ws be the *List* of events such that R *reads-bytes-from* §6.3.1.8 Ws.
b. Let byteLocation be R.[[ByteIndex]].
c. For each element W of Ws in *List* order:
   i. It is not the case that R *happens-before* §6.3.1.12 W, and
   ii. There is no WriteSharedMemory or ReadModifyWriteSharedMemory event V that has byteLocation in its *range* §6.3.1.2 such that W *happens-before* §6.3.1.12 V and V *happens-before* §6.3.1.12 R.
   iii. Increment byteLocation by 1.

Formal Integration

Definition 6.3.36 (Coherent Reads). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the coherent reads CR property over $CE(T)$ is defined as*

$$CR(CE(T)) := \forall r, w \in RBF(r, w).\ \neg HB(r, w) \wedge 2_{cii}$$
$$2_{cii} := \nexists v \in E.\ WoM(v) \wedge HB(w, v) \wedge HB(v, r)$$

□

6.3.1.18 No Out of Thin Air Reads

A *candidate execution* §6.3.1.5 has no out of thin air reads if there is no cycle in the union of the Relations execution.[[ReadsFrom]] and execution.[[DependsOn]].

> **Note 1:** Out of thin air reads is an artifact of memory models rather than implementation reality. For other languages with similar memory models, thin air reads have not been observed on any known hardware nor due to any known compiler transform.

> **Note 2:** (Spec draft notes)
>
> This is intentionally underspecified. Precisely capturing and forbidding OOTA is currently an open problem.

Formal Integration

Definition 6.3.37 (No Out of Thin Air). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the no out of thin air OTA property over $CE(T)$ is defined as*

$$OTA(CE(T)) := O^T((DD \cup RF)*)$$

□

6.3.1.19 Tear Free Reads

A *candidate execution* §6.3.1.5 execution has tear free reads if the following conditions hold.

1. Let *reads-from* §6.3.1.9 be `execution`.[[ReadsFrom]].
2. For each ReadSharedMemory or ReadModifyWriteSharedMemory event `R` in *EventSet* §6.3.1.6(`execution`):

   a. If `R`.[[NoTear]] is `true` then

      i. Assert: the remainder of `R`.[[ByteIndex]] / `R`.[[ElementSize]] is `0`.

      ii. For each event `W` such that `R` *reads-from* §6.3.1.9 `W` and `W`.[[NoTear]] is `true`:

         A. If `R` and `W` have equal ranges then there is no `V` such that `V` and `W` have equal *range* §6.3.1.2, `V`.[[NoTear]] is `true`, `W` is not `V`, and `R` *reads-from* §6.3.1.9 `V`.

Note: An event's [[NoTear]] field is `true` when that event was introduced via accessing a TypedArray, and `false` when introduced via accessing a DataView.

Intuitively, this requirement says when a memory *range* §6.3.1.2 is accessed in an aligned fashion via a TypedArray, a single write event on that *range* §6.3.1.2 must "win" when in a *data race* §6.3.1.24 with other write events with equal ranges. More precisely, this requirement says an aligned read event cannot read a value composed of bytes from multiple, different write events all with equal ranges. It is possible, however, for an aligned read event to read from multiple write events with overlapping ranges. See Access Atomicity below.

Formal Integration

Definition 6.3.38 (Tear Free Reads). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the tear free reads TFR property over $CE(T)$ is defined as*

$$TFR(CE(T)) := \forall r \in E.\ RoM(r) \wedge 2_a \tag{2.}$$

$$2_a := NoTear(r) \rightarrow 2_{aii} \tag{2a.}$$

$$2_{aii} := \forall w \in E.\ (RF(r,w) \wedge NoTear(w)) \rightarrow 2_{aiiA} \tag{2aii.}$$

$$2_{aiiA} := (Range(r) = Range(w)) \rightarrow 2_{aiiA'} \tag{2aiiA.}$$

$$2_{aiiA'} := \nexists v \in E.\ (Range(v) = Range(w)) \wedge NoTear(v) \wedge (v \neq w) \wedge RF(r,v) \tag{2aiiA.}$$

Property:

$$P_{TFR} := \forall r \in E.\ RoM(r) \wedge 2_a \rightarrow 2_{ai} \tag{2.}$$

$$2_a := NoTear(r) \tag{2a.}$$

$$2_{ai} := \exists k \in \mathbb{N}.\ ElementSize(r) * k = ByteIndex(r) \tag{2ai.}$$

□

6.3.1.20 memory-order

For a *candidate execution* §6.3.1.5 execution, memory-order is a *total order* §6.3.1.1 that satisfies the following conditions.

1. Let *happens-before* §6.3.1.12 be `execution`.[[HappensBefore]].
2. Let *synchronizes-with* §6.3.1.11 be `execution`.[[SynchronizesWith]].
3. For each pair of events `E` and `D` in *EventSet* §6.3.1.6(`execution`):
   a. If `E` *happens-before* §6.3.1.12 `D` then `E` is *memory-order* §6.3.1.20 before `D`.
   b. If `E` *synchronizes-with* §6.3.1.11 `D` then
      i. Assert: `D` has `order` **"SeqCst"**.
      ii. There is no WriteSharedMemory or ReadModifyWriteSharedMemory event `W` in *EventSet* §6.3.1.6(`execution`) with equal *range* §6.3.1.2 as `D` such that `E` is *memory-order* §6.3.1.20 before `W` and `W` is *memory-order* §6.3.1.20 before `D`.
      iii. NOTE: This clause constrains **"SeqCst"** events on equal ranges, not all **"SeqCst"** events.

Formal Integration

Definition 6.3.39 (Memory Order). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the Memory Order relation MO is defined as*

$$MO \subseteq E \times E \;:\; (O^T(MO') \to (MO := MO')) \wedge (\neg O^T(MO') \to (MO := \emptyset))$$

where

$$\begin{aligned} MO' \subseteq E \times E \;&:\; \forall e, d \in E. && (3.)\\ 3_a &:= HB(e,d) \to MO'(e,d) && (3a.)\\ 3_b &:= SW(e,d) \to 3_{bii} && (3b.)\\ 3_{bii} &:= \nexists w \in E.\; WoM(w) \wedge (Range(w) = Range(d)) \wedge MO'(e,w) \wedge MO'(w,d) && (3bii.) \end{aligned}$$

Revised:

$$\begin{aligned} MO' \subseteq E \times E \;&:\; \forall e, d \in E.\; MO'(e,d) \leftrightarrow (3_a \wedge 3_b) && (3.)\\ 3_a &:= HB(e,d) && (3a.)\\ 3_b &:= SW(e,d) \to 3_{bii} && (3b.)\\ 3_{bii} &:= \neg(\exists w \in E.\; WoM(w) \wedge (Range(w) = Range(d)) \wedge 3_{bii'}) && (3bii.)\\ 3_{bii'} &:= (HB(e,w) \vee SW(e,w)) \wedge (HB(w,d) \vee SW(w,d)) && (3bii.) \end{aligned}$$

Property:

$$\begin{aligned} P_{MO} &:= \forall e, d \in E.\; (3_b \to 3_{bi}) && (3.)\\ 3_b &:= SW(e,d) && (3b.)\\ 3_{bi} &:= SeqCst(d) && (3bi.) \end{aligned}$$

□

6.3.1.21 Sequentially Consistent Atomics

A *candidate execution* §6.3.1.5 has sequentially consistent atomics if a *memory-order* §6.3.1.20 exists.

Formal Integration

Definition 6.3.40 (Sequentially Consistent Atomics). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the Sequentially Consistent Atomics SCA property over $CE(T)$ is defined as*

$$SCA(CE(T)) := MO(CE(T)) \neq \emptyset$$

□

6.3.1.22 Valid Executions

A *candidate execution* §6.3.1.5 execution is a valid execution (or simply an execution) if the following conditions hold.

1. The host provides a *host-synchronizes-with* §6.3.1.10 *Relation* §6.3.1.1 for `execution`.[[HostSynchronizesWith]], and
2. The host provides a *depends-on* §6.3.1.13 *Relation* §6.3.1.1 for `execution`.[[DependsOn]], and
3. `execution` has valid chosen reads, and
4. `execution` has coherent reads, and
5. `execution` has no out of thin air reads, and
6. `execution` has tear free reads, and
7. `execution` has sequentially consistent atomics.

All programs have at least one valid execution.

> **Note:** (Spec draft notes)
>
> The intuition that the set of valid executions for any ECMAScript program is nonempty owes to that a sequentially consistent (i.e., interleaving) execution may always be demonstrated. Clearly, this valid execution is too strong and insufficient to account for all observable behaviors in implementations. Instead, it serves as an existence proof that the *memory model* §6.3 is well defined for ECMAScript. Below is an informal sketch of how to construct a sequentially consistent execution. This sketch spells out the naive intuition programmers often apply to multithreaded programs by imagining them sequentially consistent.
>
> The idea is to build an interleaving of events, a *total order* §6.3.1.1 called cluster-order, that is consistent with, but strictly stronger than, *agent-order* §6.3.1.7. This cluster-order is usable as an *agent-order* §6.3.1.7, and from it the additional relations required for a valid execution are constructible. These additional relations are also stronger than what the model requires by construction. Moreover, constraints such as sequentially consistent atomics also hold by construction. This then demonstrates that an interleaving of events will always result in a valid execution.

The inputs to the *memory model* §6.3 from the evaluation semantics are per-agent agent-orders of events in a *candidate execution* §6.3.1.5's [[EventLists]] and the nondeterministically chosen values in [[ChosenValues]]. Given each agent's own total orders in [[EventLists]], let cluster-order be a *total order* §6.3.1.1 that is consistent with the total orders of all agents. This cluster-order may be constructed by "zipping" events from each agent's order or simply by appending (i.e. thread inlining). One caveat here is that all **"Init"** events introduced by *CreateSharedByteDataBlock* §2.2.1.2 must be made to precede all non-**"Init"** events in cluster-order. For example, if Agent 1 has events [A, B] and Agent 2 has events [C, D], cluster-order can be [A, C, B, D], or [C, A, D, B], or [A, B, C, D], and so forth. It, however, cannot be [D, C, A, B], as that would be inconsistent with Agent 2's *total order* §6.3.1.1 of its own events.

With this cluster-order, a valid *reads-bytes-from* §6.3.1.8 relation is constructible as follows. For each read event R, for each of its byte locations b, choose the most recent, i.e., supremum, write event W that writes to b in cluster-order for R to *reads-bytes-from* §6.3.1.8 W for the byte at location b. Since all byte locations have at least one write event for it, the **"Init"**, such a W always exists. This *reads-bytes-from* §6.3.1.8 relation then induces *reads-from* §6.3.1.9 pointwise.

With this *reads-from* §6.3.1.9 relation, *synchronizes-with* §6.3.1.11 is constructible using the constraints given in the *memory model* §6.3. **"SeqCst"** reads that happen to read from **"SeqCst"** writes on equal ranges are *synchronizes-with* §6.3.1.11. For host-specific events disjoint from the shared memory events, the *memory model* §6.3 stipulates that the host provides a *strict partial order* §6.3.1.1 *host-synchronizes-with* §6.3.1.10 that relates those events. Thus, our constructed *synchronizes-with* §6.3.1.11 relation is a *strict partial order* §6.3.1.1.

Next, *happens-before* §6.3.1.12 is constructed by taking the transitive closure of the union of *synchronizes-with* §6.3.1.11 and *happens-before* §6.3.1.12. Since the constructed *reads-from* §6.3.1.9 and *synchronizes-with* §6.3.1.11 relations are strict partial orders, *happens-before* §6.3.1.12 is also a *strict partial order* §6.3.1.1.

Thus far, we have shown how to construct the needed relations. They must now be checked to satisfy the constraints the model sets out for them: valid chosen reads, coherent reads, no out of thin air reads, tear free reads, and sequentially consistent atomics.

Valid chosen reads may be assumed to be satisfied, as it is a formal mechanism to interface the axiomatic *memory model* §6.3 to evaluation semantics. Put another way, the evaluation semantics introduces nondeterministic read events and both deterministic and nondeterministic write events. Some writes into SharedArrayBuffers, for example, depends solely on intra-agent evaluation and are deterministic. Since the evaluation semantics were given the affordance to read any possible byte value from SharedArrayBuffers, we may simply construct the *candidate execution* §6.3.1.5's [[ChosenReads]] map to be the "right" values for the constructed cluster-order.

Coherent reads is satisfied by the construction of *reads-bytes-from* §6.3.1.8. Since reads always choose the most recent writes for each byte in cluster-order, which is a *total order* §6.3.1.1, this is trivially satisfied.

No out of thin air reads is assumed to be satisfied. The constructed *reads-from* §6.3.1.9 is acylic, and the host is assumed to have supplied a *depends-on* §6.3.1.13 *strict partial order* §6.3.1.1 that resulted in no cycles in the union of itself with *reads-from* §6.3.1.9.

Tear free reads is satisfied by construction of *reads-bytes-from* §6.3.1.8. Since reads always choose the most recent writes for each byte in cluster-order, it is impossible for a [[NoTear]] read to *reads-from* §6.3.1.9 two different write events with equal *range* §6.3.1.2 for different bytes. Since cluster-order is total, only one of the events would have been chosen during the construction of *reads-bytes-from* §6.3.1.8.

This execution has sequentially consistent atomics because cluster-order is a *total order* §6.3.1.1. All memory events are sequentially consistent, so the **"SeqCst"** events must also be sequentially consistent. In other words, cluster-order is the *memory-order* §6.3.1.20.

This concludes the sketch: the constructed execution is valid.

Formal Integration

Definition 6.3.41 (Valid Execution). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, $CE(T)$ is a valid execution, namely $V(CE(T))$, iff*

$$\begin{align}
& PO \neq \emptyset \tag{3.}\\
\wedge\ & SW \neq \emptyset \tag{4.}\\
\wedge\ & HB \neq \emptyset \tag{5.}\\
\wedge\ & RBF \neq \emptyset \tag{6.}\\
\wedge\ & RF \neq \emptyset \tag{7.}\\
\wedge\ & VCR(CE(T)) \tag{8.}\\
\wedge\ & CR(CE(T)) \tag*{(8.)}\\
\wedge\ & OTA(CE(T)) \tag{9.}\\
\wedge\ & TFR(CE(T)) \tag{10.}\\
\wedge\ & SCA(CE(T)) \tag{11.}
\end{align}$$

□

Definition 6.3.42 (Validity). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$ the formula $V(CE(T))$ is satisfiable.* □

6.3.1.23 Races

For an execution execution, two events E and D in *EventSet* §6.3.1.6(execution) are in a race if the following conditions hold.

1. Let *happens-before* §6.3.1.12 be `execution`.[[HappensBefore]].
2. Let *reads-from* §6.3.1.9 be `execution`.[[ReadsFrom]].
3. It is not the case that `E` *happens-before* §6.3.1.12 `D` or `D` *happens-before* §6.3.1.12 `E`, and
4. If `E` and `D` are both WriteSharedMemory or ReadModifyWriteSharedMemory events then
   a. `E` and `D` do not have disjoint ranges.
5. Otherwise:
   a. `E` *reads-from* §6.3.1.9 `D` or `D` *reads-from* §6.3.1.9 `E`.

Formal Integration

Definition 6.3.43 (Race Condition). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO\rangle$, the Race Condition RC over $CE(T)$, and two events $e, d \in E$ is defined as*

$$
\begin{aligned}
RC(CE(T), e, d) &:= \neg(HB(e,d) \vee HB(d,e)) \wedge 4 && (3.)\\
4 &:= (4' \rightarrow 4_a) \wedge (\neg 4' \rightarrow 5) && (4.)\\
4' &:= \mathit{WoM}(e) \wedge \mathit{WoM}(d) && (4.)\\
4_a &:= \mathit{Range}(e) \cap \mathit{Range}(d) \neq \emptyset && (4a.)\\
5 &:= RF(e,d) \vee RF(d,e) && (5.)
\end{aligned}
$$

□

6.3.1.24 Data Races

For an execution execution, two events E and D in *EventSet* §6.3.1.6(execution) are in a data race if the following conditions hold.

1. E and D are in a *race* §6.3.1.23 in execution, and
2. At least one of E or D does not have [[Order]] **"SeqCst"** or E and D have overlapping ranges.

Formal Integration

Definition 6.3.44 (Data Race Condition). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO$, HSW, SW, HB, RBF, $DO\rangle$, the Data Race Condition RC over $CE(T)$, and two events $e, d \in E$ is defined as*

$$
\begin{aligned}
DRC(CE(T), e, d) &:= DR(CE(T), e, d) \wedge 2 && (1.)\\
2 &:= \neg(\mathit{SeqCst}(e) \wedge \mathit{SeqCst}(d)) \vee (\mathit{Range}(d) \cap \mathit{Range}(e) \neq \emptyset) && (2.)
\end{aligned}
$$

□

6.3.1.25 Data Race Freedom

An execution execution is data race free if there are no two events in *EventSet* §6.3.1.6(execution) that are in a *data race* §6.3.1.24.

A program is data race free if all its executions are data race free.

Formal Integration

Definition 6.3.45 (Data Race Freedom). *Given a threads union T, and a Candidate Execution $CE(T) := \langle AO, HSW, SW, HB, RBF, DO \rangle$, the Data Race Freedom DRF property over $CE(T)$ is defined as*

$$DRF(CE(T)) := \neg \exists e, d \in E.\ DRC(CE(T), e, d)$$

□

6.3.1.26 Access Atomicity (informative)

In an execution, a ReadSharedMemory or ReadModifyWriteSharedMemory event is access atomic when it *reads-from* §6.3.1.9 a single WriteSharedMemory event or ReadModifyWriteSharedMemory event. In the *memory model* §6.3, write events have no force beyond being read from by read events and thus have no notion of being access atomic. This differs from the colloquial notion of access atomicity, which usually means if a write or read was performed by hardware atomically.

The Tear Free Reads requirement does not guarantee access atomicity for non-atomic events. Consider the following program where **U8** and **U16** are aliased 1-byte and 2-byte views on the same *Shared Data Block* §2.2.1 and **a** and **b** represent the same memory address (scaled appropriately for the data type).

```
W1: U16[a] = 1; W2: U8[b] = 2;|| W3: U16[a] = 3;|| R: observe(U16[a]);
```

A *candidate execution* §6.3.1.5 where all the following hold is a valid execution.

- **R** *reads-from* §6.3.1.9 **W1**
- **R** *reads-from* §6.3.1.9 **W2**

R reads a value composed of bytes from **W1** and **W2**, and is thus not access atomic. At the same time, it does not read-from both **W2** and **W3**, and thus is tear free.

Similarly, neither the *synchronizes-with* §6.3.1.11 relation, the Sequentially Consistent Atomics requirement guarantees access atomicity for atomic events. Both are concerned with ordering of atomic operations on equal ranges, but it is possible to have sequentially consistent equal-ranged atomics without access atomicity. For example, consider the following program where **U8** and **U16** are aliased 1-byte and 2-byte views on the same *Shared Data Block* §2.2.1.

```
W1: Atomics.store(U16, a, 1); W2: Atomics.store(U8, b, 2);|| W3: Atomics.store(U16, a, 3);|| R: observe(
    Atomics.load(U16, a));
```

A *candidate execution* §6.3.1.5 where all of the following hold is a valid execution.

- **R** *synchronizes-with* §6.3.1.11 **W1**
- **R** *reads-from* §6.3.1.9 **W1**
- **R** *reads-from* §6.3.1.9 **W2**
- **W1** is *memory-order* §6.3.1.20 before **W2** and **W2** is *memory-order* §6.3.1.20 before **R** and **R** is *memory-order* §6.3.1.20 before **W3**

W2 is allowed to come between **W1** and **R** in *memory-order* §6.3.1.20 because it and **R** do not have equal ranges. Thus, **R** may read a value composed of bytes written by both **W1** and **W2**.

Finally, *data race* §6.3.1.24 freedom and sequential consistency do not imply access atomicity. Consider the following program with a single agent.

```
W1: Atomics.store(U16, a, 1); W2: Atomics.store(U8, b, 2); R: observe(Atomics.load(U16, a));
```

The only valid execution is as follows.

- **R** *synchronizes-with* §6.3.1.11 **W1**
- **R** *reads-from* §6.3.1.9 **W1**
- **R** *reads-from* §6.3.1.9 **W2**
- **W1** is *memory-order* §6.3.1.20 before **W2** and **W2** is *memory-order* §6.3.1.20 before **R**

All valid executions of the program are *data race free* §6.3.1.25, since there is a single thread. But **R** is not access atomic since it *reads-from* §6.3.1.9 multiple events.

The central problem the *memory model* §6.3 solves is the ordering of memory events. The SharedArrayBuffer API's lack of type discipline and allowance of overlapping accesses makes access atomicity an orthogonal property from event ordering.

A program wishing to avoid reasoning about access atomicity or wishing to guarantee all its aligned and atomic accesses are access atomic should avoid accesses on overlapping ranges. When a program does not use address ranges in an overlapping fashion, aligned and atomic accesses are access atomic.

6.3.1.27 Sequential Consistency (informative)

The *memory model* §6.3 guarantees sequential consistency of all events for *data race free* §6.3.1.25 programs.

> **Note:** A formal proof of the property requires the host to provide a *depends-on* §6.3.1.13 relation that prohibits out of thin air reads.

6.3.2 Methods

6.3.2.1 ValidateSharedIntegerTypedArray(typedArray [, onlyInt32])

```
1. If Type(typedArray) is not Object, throw a TypeError exception.
2. If typedArray does not have a [[TypedArrayName]] internal slot, throw a TypeError exception.
3. Let typeName be typedArray.[[TypedArrayName]].
4. If onlyInt32 is true then
    a. If typeName is not "Int32Array" then throw a TypeError exception.
5. Else,
    a. If typeName is not "Int8Array", "Uint8Array", "Int16Array", "Uint16Array", "Int32Array", or "Uint32Array" then throw a TypeError exception.
```

6. If typedArray does not have a [[ViewedArrayBuffer]] internal slot, throw a TypeError exception.
7. Let buffer be typedArray.[[ViewedArrayBuffer]].
8. If *Type*(buffer) is not Object, throw a TypeError exception.
9. If buffer does not have an [[ArrayBufferData]] internal slot, throw a TypeError exception.
10. If *IsSharedArrayBuffer* §6.2.1.2(buffer) is false, throw a TypeError exception.
11. Return buffer.

6.3.2.2 ValidateAtomicAccess(typedArray, requestIndex)

Perform the following steps:

1. Assert: typedArray is an Object that has a [[ViewedArrayBuffer]] internal slot.
2. Let numberIndex be ? *ToNumber*(requestIndex).
3. Let accessIndex be *ToInteger*(numberIndex).
4. If numberIndex accessIndex, throw a RangeError exception.
5. Let length be typedArray.[[ArrayLength]].
6. If accessIndex < 0 or accessIndex ≥ length, throw a RangeError exception.
7. Return accessIndex.

6.3.2.3 AgentSignifier( )

When AgentSignifier is called without arguments then the following steps are taken.

1. Let AR be the *Agent Record* §3.3 of the surrounding agent.
2. Let W be AR.[[Signifier]].
3. Return W.

6.3.2.4 AgentCanSuspend( )

When AgentCanSuspend is called without arguments then the following steps are taken.

1. Let AR be the *Agent Record* §3.3 of the surrounding agent.
2. Let B be AR.[[CanBlock]].
3. Return B.

> **Note:** In some environments it may not be reasonable for a given agent to suspend. For example, in a web browser environment, it may be reasonable to disallow suspending a document's main event handling thread, while still allowing workers' event handling threads to suspend.

6.3.2.5 GetWaiterList(block, i)

A WaiterList is a semantic object that contains an ordered list of those agents that are waiting on a location (block, i) in shared memory; block is a *Shared Data Block* §2.2.1 and i a byte offset into the memory of block.

The agent cluster has a store of WaiterList objects; the store is indexed by (block, i). WaiterLists are agent-independent: a lookup in the store of WaiterLists by (block, i) will result in the same WaiterList object in any agent in the agent cluster.

Operations on a WaiterList – adding and removing waiting agents, traversing the list of agents, suspending and waking agents on the list – may only be performed by agents that have entered the WaiterList's critical section.

When GetWaiterList is called with *Shared Data Block* §2.2.1 block and nonnegative integer i, then the following steps are taken.

1. Assert: `block` is a *Shared Data Block* §2.2.1.
2. Assert: `i` and `i`+3 are valid byte offsets within the memory of `block`.
3. Assert: `i` is divisible by 4.
4. Return the *WaiterList* §6.3.2.5 that is referenced by the pair (`block`, `i`).

6.3.2.6 EnterCriticalSection(WL)

When EnterCriticalSection is called with *WaiterList* §6.3.2.5 WL, then the following steps are taken.

1. Assert: The calling agent is not in the critical section for any *WaiterList* §6.3.2.5.
2. Wait until no agent is in the critical section for `WL`, then enter the critical section for `WL` (without allowing any other agent to enter).

6.3.2.7 LeaveCriticalSection(WL)

When LeaveCriticalSection is called with *WaiterList* §6.3.2.5 WL, then the following steps are taken.

1. Assert: The calling agent is in the critical section for `WL`.
2. Leave the critical section for `WL`.

6.3.2.8 AddWaiter(WL, W)

When AddWaiter is called with *WaiterList* §6.3.2.5 WL and agent signifier W, then the following steps are taken.

1. Assert: The calling agent is in the critical section for `WL`.
2. Assert: `W` is not on the list of waiters in any *WaiterList* §6.3.2.5.
3. Add `W` to the end of the list of waiters in `WL`.

6.3.2.9 RemoveWaiter(WL, W)

When RemoveWaiter is called with *WaiterList* §6.3.2.5 WL and agent signifier W, then the following steps are taken.

```
1. Assert: The calling agent is in the critical section for WL.
2. Assert: W is on the list of waiters in WL.
3. Remove W from the list of waiters in WL.
```

6.3.2.10 RemoveWaiters(WL, c)

When RemoveWaiters is called with *WaiterList* §6.3.2.5 WL and nonnegative integer c, then the following steps are taken.

```
1. Assert: The calling agent is in the critical section for WL.
2. Let L be the empty list.
3. Let S be a reference to the list of waiters in WL.
4. While $c > 0$ and S is not the empty list:
    a. Let W be the first waiter in S.
    b. Add W to the end of L.
    c. Remove W from S.
    d. Subtract 1 from c.
5. Return L.
```

6.3.2.11 Suspend(WL, W, timeout)

When Suspend is called with *WaiterList* §6.3.2.5 WL, agent signifier W, and nonnegative, non-`NaN` Number timeout, then the following steps are taken.

```
1. Assert: The calling agent is in the critical section for WL.
2. Assert: W is equal to AgentSignifier §6.3.2.3().
3. Assert: W is on the list of waiters in WL.
4. Assert: AgentCanSuspend §6.3.2.4() is equal to true.
5. Perform LeaveCriticalSection §6.3.2.7(WL) and suspend W for up to timeout milliseconds, performing the combined operation in such a way that a wakeup that arrives after the critical section is exited but before the suspension takes effect is not lost. W can wake up either because the timeout expired or because it was woken explicitly by another agent calling WakeWaiter §6.3.2.12(WL, W), and not for any other reasons at all.
6. Perform EnterCriticalSection §6.3.2.6(WL).
7. If W was woken explicitly by another agent calling WakeWaiter §6.3.2.12(WL, W), then return true.
8. Return false.
```

6.3.2.12 WakeWaiter(WL, W)

When WakeWaiter is called with *WaiterList* §6.3.2.5 WL and agent signifier W, then the following steps are taken.

1. Assert: The calling agent is in the critical section for `WL`.
2. Assert: `W` is on the list of waiters in `WL`.
3. Wake the agent `W`.

> **Note:** The embedding may delay waking W, eg for resource management reasons, but W must eventually be woken in order to guarantee forward progress.

6.3.2.13 AtomicReadModifyWrite(typedArray, index, value, op)

AtomicReadModifyWrite is a semantic function that atomically loads a value, combines it with another value, and stores the result of the combination. It returns the loaded value. It is parameterized by the (pure) combining operation op that takes two *List* of byte values arguments and returns a *List* of byte values. The following steps are taken:

1. Let `buffer` be ? *ValidateSharedIntegerTypedArray* §6.3.2.1(`typedArray`).
2. Let `i` be ? *ValidateAtomicAccess* §6.3.2.2(`typedArray`, `index`).
3. Let `v` be ? *ToInteger*(`value`).
4. Let `arrayTypeName` be `typedArray`.[[TypedArrayName]].
5. Let `elementSize` be the Number value of the Element Size value specified in *Table 50* for `arrayTypeName`.
6. Let `elementType` be the String value of the Element Type value in *Table 50* for `arrayTypeName`.
7. Let `offset` be `typedArray`.[[ByteOffset]].
8. Let `indexedPosition` be (`i` × `elementSize`) + `offset`.
9. Return *GetModifySetValueInBuffer* §6.1.1.6(`buffer`, `indexedPosition`, `elementType`, `v`, `op`).

6.3.3 Function Properties of the Atomics Object

6.3.3.1 Atomics.add(typedArray, index, value)

Let **add** denote a semantic function of two *List* of byte values arguments that applies the addition operation to the Number values corresponding to the *List* of byte values arguments and returns a *List* of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return *AtomicReadModifyWrite* §6.3.2.13(`typedArray`, `index`, `value`, **add**).

6.3.3.2 Atomics.and(typedArray, index, value)

Let **and** denote a semantic function of two *List* of byte values arguments that applies the bitwise-and operation element-wise to the two arguments and returns a *List* of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return *AtomicReadModifyWrite* §6.3.2.13(`typedArray`, `index`, `value`, **and**).

6.3.3.3 Atomics.compareExchange(typedArray, index, expectedValue, replacementValue)

The following steps are taken:

1. Let `buffer` be ? *ValidateSharedIntegerTypedArray* §6.3.2.1(`typedArray`).
2. Let `i` be ? *ValidateAtomicAccess* §6.3.2.2(`typedArray`, `index`).
3. Let `expected` be ? *ToInteger*(`expectedValue`).
4. Let `replacement` be ? *ToInteger*(`replacementValue`).
5. Let `arrayTypeName` be `typedArray`.[[TypedArrayName]].
6. Let `convOp` be the conversion operation specified in *Table 50* for `arrayTypeName`.
7. Let `expectedBytes` be `convOp` (`expected`).
8. Let `elementSize` be the Number value of the Element Size value specified in *Table 50* for `arrayTypeName`.
9. Let `elementType` be the String value of the Element Type value in *Table 50* for `arrayTypeName`.
10. Let `offset` be `typedArray`.[[ByteOffset]].
11. Let `indexedPosition` be (`i` × `elementSize`) + `offset`.
12. Let **compareExchange** denote a semantic function of two *List* of byte values arguments that returns the second argument if the first argument is element-wise equal to `expectedBytes`.
13. Return *GetModifySetValueInBuffer* §6.1.1.6(`buffer`, `indexedPosition`, `elementType`, `replacement`, **compareExchange**).

6.3.3.4 Atomics.exchange(typedArray, index, value)

Let **second** denote a semantic function of two *List* of byte values arguments that returns its second argument.

The following steps are taken:

1. Return *AtomicReadModifyWrite* §6.3.2.13(`typedArray`, `index`, `value`, **second**).

6.3.3.5 Atomics.isLockFree(size)

1. Let n be ? *ToInteger*(size).
2. Let AR be the *Agent Record* §3.3 of the surrounding agent.
3. If n equals 1 then return AR.[[IsLockFree1]].
4. If n equals 2 then return AR.[[IsLockFree2]].
5. If n equals 4 then return true.
6. Return false.

> **Note:** Atomics.isLockFree() is an optimization primitive. The intuition is that if the atomic step of an atomic primitive (**compareExchange**, **load**, **store**, **add**, **sub**, **and**, **or**, **xor**, or **exchange**) on a datum of size n bytes will be performed without the calling agent acquiring a lock outside the n bytes comprising the datum, then Atomics.isLockFree(n) will return true. High-performance algorithms will use Atomics.isLockFree to determine whether to use locks or atomic operations in critical sections. If an atomic primitive is not lock-free then it is often more efficient for an algorithm to provide its own locking.
>
> Atomics.isLockFree(4) always returns true as that can be supported on all known relevant hardware. Being able to assume this will generally simplify programs.

6.3.3.6 Atomics.load(typedArray, index)

The following steps are taken:

1. Let buffer be ? *ValidateSharedIntegerTypedArray* §6.3.2.1(typedArray).
2. Let i be ? *ValidateAtomicAccess* §6.3.2.2(typedArray, index).
3. Let arrayTypeName be typedArray.[[TypedArrayName]].
4. Let elementSize be the Number value of the Element Size value specified in *Table 50* for arrayTypeName.
5. Let elementType be the String value of the Element Type value in *Table 50* for arrayTypeName.
6. Let offset be typedArray.[[ByteOffset]].
7. Let indexedPosition be (i × elementSize) + offset.
8. Return *GetValueFromBuffer* §6.1.1.3(buffer, indexedPosition, elementType, true, **"SeqCst"**).

6.3.3.7 Atomics.or(typedArray, index, value)

Let **or** denote a semantic function of two *List* of byte values arguments that applies the bitwise-or operation element-wise to the two arguments and returns a *List* of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return *AtomicReadModifyWrite* §6.3.2.13(typedArray, index, value, **or**).

6.3.3.8 Atomics.store(typedArray, index, value)

The following steps are taken:

```
1. Let buffer be ? ValidateSharedIntegerTypedArray §6.3.2.1(typedArray).
2. Let i be ? ValidateAtomicAccess §6.3.2.2(typedArray, index).
3. Let v be ? ToInteger(value).
4. Let arrayTypeName be typedArray.[[TypedArrayName]].
5. Let elementSize be the Number value of the Element Size value specified in Table 50 for arrayTypeName.
6. Let elementType be the String value of the Element Type value in Table 50 for arrayTypeName.
7. Let offset be typedArray.[[ByteOffset]].
8. Let indexedPosition be (i × elementSize) + offset.
9. Perform SetValueInBuffer §6.1.1.5(buffer, indexedPosition, elementType, v, true, ”SeqCst”).
10. Return v.
```

6.3.3.9 Atomics.sub(typedArray, index, value)

Let **subtract** denote a semantic function of two *List* of byte values arguments that applies the subtraction operation to the Number values corresponding to the *List* of byte values arguments and returns a *List* of byte values corresponding to the result of that operation.

The following steps are taken:

```
1. Return AtomicReadModifyWrite §6.3.2.13(typedArray, index, value, subtract).
```

6.3.3.10 Atomics.wait(typedArray, index, value, timeout)

Atomics.wait puts the calling agent in a wait queue and puts it to sleep until it is awoken or the sleep times out. The following steps are taken:

```
1. Let buffer be ? ValidateSharedIntegerTypedArray §6.3.2.1(typedArray, true).
2. Let i be ? ValidateAtomicAccess §6.3.2.2(typedArray, index).
3. Let v be ? ToInt32(value).
4. If timeout is not provided or is undefined then let t be +. Otherwise:
    a. Let q be ? ToNumber(timeout).
    b. If q is NaN then let t be +, otherwise let t be max(0, q).
5. Let B be AgentCanSuspend §6.3.2.4().
6. If B is false then throw a TypeError exception.
7. Let bufferVal be buffer.[[ArrayBufferData]].
8. Let arrayTypeName be typedArray.[[TypedArrayName]].
```

```
 9. Let offset be typedArray.[[ByteOffset]].
10. Let indexedPosition be $(i \times 4) + offset$.
11. Let WL be GetWaiterList §6.3.2.5(block, indexedPosition).
12. Perform EnterCriticalSection §6.3.2.6(WL).
13. Let w be Atomics.load(typedArray, i).
14. If v does not equal w then
    a. Perform LeaveCriticalSection §6.3.2.7(WL).
    b. Return the string "not-equal".
15. Let W be AgentSignifier §6.3.2.3().
16. Perform AddWaiter §6.3.2.8(WL, W).
17. Let awoken be Suspend §6.3.2.11(WL, W, t).
18. Perform RemoveWaiter §6.3.2.9(WL, W).
19. Perform LeaveCriticalSection §6.3.2.7(WL).
20. If awoken is true then return the string "ok".
21. Return the string "timed-out".
```

6.3.3.11 Atomics.wake(typedArray, index, count)

Atomics.wake wakes up some agents that are sleeping in the wait queue. The following steps are taken:

```
 1. Let buffer be ? ValidateSharedIntegerTypedArray §6.3.2.1(typedArray, true).
 2. Let i be ? ValidateAtomicAccess §6.3.2.2(typedArray, index).
 3. If count is not provided or is undefined then let c be +. Otherwise:
    a. Let tmp be ? ToInteger(count).
    b. Let c be max(+0, tmp).
 4. Let bufferVal be buffer.[[ArrayBufferData]].
 5. Let arrayTypeName be typedArray.[[TypedArrayName]].
 6. Let offset be typedArray.[[ByteOffset]].
 7. Let indexedPosition be $(i \times 4) + offset$.
 8. Let WL be GetWaiterList §6.3.2.5(block, indexedPosition).
 9. Let n be 0.
10. Perform EnterCriticalSection §6.3.2.6(WL).
11. Let S be RemoveWaiters §6.3.2.10(WL, c).
12. While S is not the empty list:
    a. Let W be the first agent in S.
    b. Remove W from the front of S.
    c. Perform WakeWaiter §6.3.2.12(WL, W).
    d. Add 1 to n.
13. Perform LeaveCriticalSection §6.3.2.7(WL).
14. Return n.
```

6.3.3.12 Atomics.xor(typedArray, index, value)

Let **xor** denote a semantic function of two *List* of byte values arguments that applies the bitwise-xor operation element-wise to the two arguments and returns a *List* of byte values corresponding to the result of that operation.

The following steps are taken:

1. Return *AtomicReadModifyWrite* §6.3.2.13(`typedArray`, `index`, `value`, **xor**).

6.3.4 Programmer Guidelines

Keep programs *data race free* §6.3.1.25, i.e., make it so that it is impossible for there to be concurrent non-atomic operations on the same memory location. Data *race* §6.3.1.23 free programs have interleaving semantics where each step in the evaluation semantics of each agent are interleaved with each other. For *data race free* §6.3.1.25 programs, it is not necessary to understand the details of the *memory model* §6.3. The details are unlikely to build intuition that will help one to better write JavaScript.

More generally, even if a program is not *data race free* §6.3.1.25 it may have predictable behavior, so long as atomic operations are not involved in any data races and the operations that *race* §6.3.1.23 all have the same access size. The simplest way to arrange for atomics not to be involved in races is to ensure that different memory cells are used by atomic and non-atomic operations and that atomic accesses of different sizes are not used to access the same cells at the same time. Effectively, the program should treat shared memory as strongly typed as much as possible. One still cannot depend on the ordering and timing of non-atomic accesses that *race* §6.3.1.23, but if memory is treated as strongly typed the racing accesses will not "tear" (bits of their values will not be mixed).

6.3.5 Compiler Transformation Guidelines

> **Note 1:** (Spec draft notes)
>
> The *memory model* §6.3 provides precise semantics for shared memory events, but it does without fixing its various relations. It prescribes what memory values may be observed for each event, but does not require, e.g., *agent-order* §6.3.1.7, to be fixed. Indeed, transformations performed by the ECMAScript implementation or hardware may change *agent-order* §6.3.1.7 and the other relations. Consider a single agent running in isolation. In the single-agent case, all semantics-preserving transformations are allowed, but, being semantics-preserving, they are not observable to the agent. To contrast, in the multi-agent case, those same transformations may become observable by other agents via shared memory accesses.
>
> The question arises as to which transformations that are allowed in a multi-agent setting, where transformations that affect shared memory writes are observable by other agents. An agent that writes 0 to **x** and then writes 1 to **y** can be transformed in a single-agent setting to one that writes 1 to **y** and then writes 0 to **x**. But is that a legal transformation if those variables are in shared memory and another agent might observe the order of writes? The transformation would affect the *agent-order* §6.3.1.7 locally and the *happens-before* §6.3.1.12 relation globally. And does the answer depend on whether **x** and **y** are written with **"SeqCst"** operations or not?
>
> It is desirable to allow most program transformations that are valid in a single-agent setting in a multi-agent setting, to ensure that the performance of each agent in a multi-agent program is as good as it would be in a single-agent setting. (There seems to be broad agreement in the programming language design community that that is the right trade-off.) But "most" does not mean "all". In the example above, if **x** or **y** (or both) were intended to be **"SeqCst"** writes, and the writes were reordered, not only might *happens-before* §6.3.1.12 be affected, but also *synchronizes-with* §6.3.1.11 and *memory-order* §6.3.1.20, and a program that before transformations was data-*race* §6.3.1.23-free and sequentially consistent might no longer be so. In contrast, if **x** and **y** were both **"Unordered"** then in a data-*race* §6.3.1.23-free program the change in write order would not be observable at all.

> Frequently these transformations are hard to judge. Can an **"Unordered"** load be moved out of a loop? Even if the loop's termination condition depends on the loaded value? The load is necessarily racy, and the *memory model* §6.3 gives some meaning to races; can we depend on that meaning here?
>
> With that background, we outline some rules about program transformations that are intended to be taken as normative (in that they fall out of the *memory model* §6.3) but which are likely not exhaustive. These rules are intended to apply to program transformations that precede the introductions of the events that make up the *agent-order* §6.3.1.7.

Let an agent-order slice be the subset of the *agent-order* §6.3.1.7 pertaining to a single agent.

Let possible read values of a read event be the set of all values of *ValueOfReadEvent* §6.3.1.15 for that event across all valid executions.

Any transformation of an agent-order slice that is valid in the absence of shared memory is valid in the presence of shared memory, with the following exceptions.

- Atomics are carved in stone: Program transformations must not cause the **"SeqCst"** events in an agent-order slice to be reordered with its **"Unordered"** operations, nor its **"SeqCst"** operations to be reordered with each other, nor may a program transformation remove a **"SeqCst"** operation from the *agent-order* §6.3.1.7. (In practice, the prohibition on reorderings forces a compiler to assume that every **"SeqCst"** operation is a synchronization and included in the final *memory-order* §6.3.1.20, which it would usually have to assume anyway in the absence of inter-agent program analysis. It also forces the compiler to assume that every call where the callee's effects on the *memory-order* §6.3.1.20 are unknown may contain **"SeqCst"** operations.)
- Reads must be stable: Any given shared memory read must only observe a single value in an execution. (For example, if what is semantically a single read in the program is executed multiple times then the program is subsequently allowed to observe only one of the values read. A transformation known as rematerialization can violate this rule.)
- Writes must be stable: All observable writes to shared memory must follow from program semantics in an execution. (For example, a transformation may not introduce certain observable writes, such as by using read-modify-write operations on a larger location to write a smaller datum, writing a value to memory that the program could not have written, or writing a just-read value back to the location it was read from, if that location could have been overwritten by another agent after the read.)
- Possible read values must be nonempty: Program transformations cannot cause the possible read values of a shared memory read to become empty.(Counterintuitively, this rule in effect restricts transformations on writes, because writes have force in *memory model* §6.3 insofar as to be read by read events. For example, writes may be moved and coalesced and sometimes reordered between two **"SeqCst"** operations, but the transformation may not remove every write that updates a location; some write must be preserved.)

> **Note 2:** (Spec draft note) These rules are technically stronger than what the *memory model* §6.3 implies. For example, for the atomics are carved in stone rule, the *memory model* §6.3 only implies that **"SeqCst"** events that *synchronizes-with* §6.3.1.11 other events be totally ordered. In practice, compilers cannot perform inter-agent program analysis and must assume every atomic call gives rise to **"SeqCst"** events that *synchronizes-with* §6.3.1.11 other events.

The rules apply only to shared memory accesses, so an implementation that knows the program is not accessing shared memory will not be affected by them. In practice, the optimizations that are affected by the rules are of limited value, and a practical implementation could avoid those optimizations even for non-shared memory without significant impact.

Examples of transformations that remain valid are: merging multiple non-atomic reads from the same location, reordering non-atomic reads, introducing speculative non-atomic reads, merging multiple non-atomic writes to the same location, reordering non-atomic writes to different locations, and hoisting non-atomic reads out of loops even if that affects termination. Note in general that aliased TypedArrays make it hard to prove that locations are different.

6.3.6 Code Generation Guidelines

For architectures with memory models no weaker than those of ARM or Power, non-atomic stores and loads may be compiled to bare stores and loads on the target architecture. Atomic stores and loads may be compiled down to instructions that guarantee sequential consistency. If no such instructions exist, memory barriers are to be employed, such as placing barriers on both sides of a bare store or load. Read-modify-write operations may be compiled to read-modify-write instructions on the target architectrue, such as **LOCK**-prefixed instructions on x86 and load-link/store-conditional instructions on ARM.

Note: (Spec draft notes)

While the *memory model* §6.3 is prescriptive by nature, it is intended to allow obvious code generation. Non-atomic accesses are intended to map to bare stores and loads and atomic accesses to the obvious instructions on the underlying hardware. The intention is thus descriptive; the model must allow enough executions such that the astonishing effects that bare stores and loads on hardware with weak memory models exhibit are not prohibited. If an obvious code generation strategy is found to be non-compliant, the bug is likely in the *memory model* §6.3 and not the implementation.

Specifically, the *memory model* §6.3 is intended to allow code generation as follows. There are some ground rules:

- Every atomic operation in the program is assumed to be necessary.
- Atomic operations are never rearranged with each other or with non-atomic operations.
- Functions are always assumed to perform atomic operations.
- Atomic operations are never implemented as read-modify-write operations on larger data, but as non-lock-free atomics if the platform does not have atomic operations of the appropriate size. (We already assume that every platform has normal memory access operations of every interesting size.)

With that in mind, naive code generation uses these patterns:

- Regular loads and stores compile to single load and store instructions.
- Lock-free atomic loads and stores compile to a full (sequentially consistent) fence, a regular load or store, and a full fence.
- Lock-free atomic read-modify-write accesses compile to a full fence, an atomic read-modify-write instruction sequence, and a full fence.
- Non-lock-free atomics compile to a spinlock acquire, a full fence, a series of non-atomic load and store instructions, a full fence, and a spinlock release.

That mapping is correct so long as an atomic operation on an address *range* §6.3.1.2 does not *race* §6.3.1.23 with a non-atomic write or with an atomic operation of different size. However, that is all we need: the *memory model* §6.3 effectively demotes the atomic operations involved in a *race* §6.3.1.23 to non-atomic status. On the other hand, the naive mapping is quite strong: it allows atomic operations to be used as sequentially consistent fences, which the *memory model* §6.3 does not actually guarantee.

A number of local improvements to those basic patterns are also intended to be legal:

- There are obvious platform-dependent improvements that remove redundant fences. For example, on x86 the fences around lock-free atomic loads and stores can always be omitted except for the fence following a store, and no fence is needed for lock-free read-modify-write instructions, as these all use LOCK-prefixed instructions. On many platforms there are fences of several strengths, and weaker fences can be used in certain contexts without destroying sequential consistency.
- Most modern platforms support lock-free atomics for all the data sizes required by ECMAScript atomics. Should non-lock-free atomics be needed, the fences surrounding the body of the atomic operation can usually be folded into the lock and unlock steps. The simplest solution for non-lock-free atomics is to have a single lock word per SharedArrayBuffer.

- There are also more complicated platform-dependent local improvements, requiring some code analysis. For example, two back-to-back fences often have the same effect as a single fence, so if code is generated for two atomic operations in sequence, only a single fence need separate them. On x86, even a single fence separating atomic stores can be omitted, as the fence following a store is only needed to separate the store from a subsequent load.

One informative reference about local improvements is the writeup from the Cambridge Relaxed Memory Group.

On the other hand, the restrictions the *memory model* §6.3 places on compiler optimizations are less obvious. To this end please consult the Compiler Transformation Guidelines section above, designed to give informal descriptions of restrictions on optimizations that follow from the *memory model* §6.3.

6.4 Web browser embedding (informative)

This section outlines how the Shared Memory and Atomics specification fits into the current web ecosystem with the minimum amount of change to that ecosystem. This section is not part of the proposal, it is informative only.

In a web browser an agent is an HTML event loop [here]. The event loop is realized as either a main thread (which may be shared among tabs, as it is in Firefox) or some type of worker thread. The event loops are running jobs in the sense of ES262, and the forward-progress requirement of this specification (section3.1.1) is generally met as long as each agent has its own dedicated operating system thread or shared-thread agents can't block.

Browsers will typically let agents that run on the browser's main thread have [[CanBlock]] equal to `false`, to prevent blocking the UI and to allow the main thread to do work on behalf of other threads.

Normal (”dedicated”) **Worker**s must be supported as agents for this proposal to make much sense.

Memory will be shared among agents by using **postMessage** to transmit SharedArrayBuffer objects or TypedArray objects that have SharedArrayBuffer buffers; this requires an extension to the Structured Clone mechanism. There is agreement on the general syntax and semantics for that extension.

Agent-to-agent communication extends the *synchronizes-with* §6.3.1.11 relation of the program, as follows:

- The call to the **Worker** constructor in the parent *synchronizes-with* §6.3.1.11 the execution of the main script in the worker.
- (Worker termination is not directly observable in current browsers, but this is a moot point; see below.)
- A **postMessage** to another agent *synchronizes-with* §6.3.1.11 the event that fires in the agent.

So long as the browser does not allow a **ServiceWorker** or **SharedWorker** (”non-page worker”) to share memory with a **Worker** it will not violate the suspend/wake cohort rule of this specification (section3.3). The restriction on sharing memory can be implemented in the extension to the Structured Clone mechanism. For example, a non-page worker may simply not be allowed to receive shared memory (leading to a null value or an error signal). There is not yet any agreement on this point, but it is clear that the restriction must be on the receiving side of the communication, as the sender may be sending on a **MessagePort** that is not yet connected, but may in the future be connected to either a valid or invalid recipient.

At the moment, I believe dedicated workers are in the same process as their owning tab in all browsers, so if the process crashes then the workers will crash too. I also don't know any reason a browser should forcibly terminate a worker except when a tab is closed. In sum, the termination signaling requirement of this specification (section3.3) is probably met by existing browsers.

The web platform should evolve to serve the shared memory use case better: by incorporating inspectable worker state and worker lifecycle events, and by tightening the wording in the HTML spec (currently the browser is allowed to kill a worker at any time for any reason, which is not quite what we want). WebIDL should evolve to allow Web APIs to be described as to when they can and cannot receive shared memory parameters. However, only an extension to the Structured Clone algorithm is needed to support the Shared Memory and Atomics specification in practice on current browsers.

Appendix A

Copyright & Software License

Copyright Notice

Software License

Bibliography

[FSAB16] Alan Jeffrey. A WIP memory model for ECMAScript shared memory, 2016. **Link**.

[SAB16] ECMA TC39 Committee. ECMAScript Shared Memory and Atomics: Shared Array Buffer Extension, 2016. **Link**.