ecma INTERNATIONAL

Standard ECMA-419

1st Edition / June 2021

ECMAScript® embedded systems API specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

Contents

Page

Introduction

This Standard, ECMAScript embedded systems API specification, defines APIs for use on embedded systems. Embedded systems are far more diverse than personal computers, smartphones, and web servers where ECMAScript is most widely used. The diversity of embedded hardware is a consequence of devices being optimized for a specific product or class of products.

It is not enough for these APIs to support the features embedded systems have in common. To be truly useful, they must allow access to the unique hardware capabilities of each embedded system. This requirement makes this Standard very different from that of a computer language which is grounded in the formality and rigor of mathematics. Hardware can be inconsistent, even sometimes messy, but it needs to be accommodated.

The ability for scripts to access unique hardware capabilities has an important consequence. It means that not all correct scripts will run correctly on all hardware. If a script requires a feature that is unavailable, it cannot run. While it is common in ECMAScript to emulate missing language and runtime features with a “polyfill”, this is usually impractical, if not impossible, for hardware capabilities. Therefore, the goal of this Standard is to make it possible to write portable scripts for specific operations, not to guarantee that all scripts execute correctly on any conformant deployment.

One important consideration when designing hardware products is cost. The APIs are designed to allow efficient execution with minimal resource use. They assume no minimum or maximum configuration. Advances in the state-of-the-art of ECMAScript engines, microcontrollers, and runtime libraries will determine where these APIs may be used.

This Standard is influenced by the Extensible Web Manifesto. It aims to provide low-level APIs that do things — primarily related to hardware and communication — that the ECMAScript language cannot do by itself. These low-level APIs are functional, simple, and efficient. The APIs may be used directly. However, it is expected that many developers will interact with them indirectly through higher-level modules and frameworks that build upon the low-level APIs. This layered approach keeps the low-level APIs small and focused while allowing a variety of uses and API styles to be built upon them.

This Ecma Standard was developed by Technical Committee 53 and was adopted by the General Assembly of June 2021.

ECMAScript® embedded systems API specification

1 Scope

This Standard defines application programming interfaces (APIs) for ECMAScript modules that support programs executing on embedded systems.

This Standard defines APIs for capabilities found in common across embedded systems. Implementations for embedded systems that include additional capabilities are encouraged to provide ECMAScript APIs for those using the many extensibility options provided by this Standard.

This Standard does not make any changes to the ECMAScript language as defined by ECMAScript language specification (ECMA-262). It does strongly encourage all deployments to execute only in strict-mode. It recommends hosts incorporate an engine that supports Secure ECMAScript and that script code is written to conform to the Secure ECMAScript runtime constraints.

2 Conformance

A conforming implementation of the ECMAScript Embedded Systems API Specification must conform to ECMA-262 and must provide and support all the objects, properties, functions, and program semantics required by this specification.

A conforming implementation of the ECMAScript Embedded Systems API Specification is permitted to provide additional objects, properties, and functions beyond those described in this specification.

In particular, a conforming implementation of this Standard is permitted to provide properties not described herein, and values for those properties, for objects that are described in this specification. A conforming implementation is permitted to add optional arguments to the functions defined in this specification only where noted.

Because implementation differences are permitted (for example, to accommodate differentiating hardware features), this Standard does not guarantee that all scripts execute correctly on every conformant deployment.

Self-hosted implementations are permitted as long as they conform to the requirements of this Standard (for example, ensuring internal properties are not visible).

3 Normative references

The following referenced documents are required for the application of this document. For dated references, only the edition cited applies. For references without a date or version number, the latest edition of the referenced document (including any amendments) applies.

ECMA-262, *ECMAScript language specification*
https://www.ecma-international.org/publications/standards/Ecma-262.htm

ECMA-402, *ECMAScript internationalization API*
https://www.ecma-international.org/publications/standards/Ecma-402.htm

RFC 2119, *Key words for use in RFCs to Indicate Requirement Levels*
https://tools.ietf.org/html/rfc2119

4 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

4.1
address
an identifier for interfacing with a specific component, device, or board

4.2
baud rate
the rate at which information is transferred, measured in bits per second

4.3
bus
a communications system that transfers data. A “Bus” includes hardware, software, and the protocol.

4.4
connected sensing device
a sensing device that communicates with a remote endpoint

4.5
direct measurement
a sample that has been captured from a configured sensor without alteration

4.6
expander
a device that provides additional inputs and/or outputs

4.7
instance
an object that has been created by a function constructor, class constructor, or function factory

4.8
microcontroller
a single integrated circuit with one or more CPUs, memory, and programmable input/output

4.9
protocol
a system of rules that define how data is exchanged between systems

4.10
register
locations in a device’s memory that can be written to or read from. These memory locations may contain configuration settings or the current state of the device.

4.11
remote endpoint
a computing system in communication with the microcontroller

4.12
sensing device
a system comprising an embedded controller with at least one attached sensor

4.13
sensor
a device that detects and responds to some type of input from the physical environment, attached to a microcontroller used to capture data.

4.14
sensor classification
sensor type, as determined by the real quantity that is, or quantities that are, subject to measurement, e.g. mass, power, or humidity. Uses names of Sensor Classes defined by this Standard. If a sensor measures real quantities defined as properties in multiple unique Sensor Classes, the name of any applicable Sensor Class may be used.

4.15
sensor configuration
user-defined parameters impacting the sampling, processing, representation, and/or transmission of peripheral data.

4.16
synthetic measurement
a direct measurement that has been modified in some form so as to potentially lose accuracy, precision, or fidelity.

5 Notational conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

6 Overview

6.1 ECMAScript

This Standard builds on Standard ECMAScript as defined by Ecma TC39. As of this writing, that is ECMAScript 2020.

This Standard is not an extension or subset of ECMAScript 2020. It is a set of APIs to be used within that Standard. The relationship between ECMA-419 and ECMAScript is analogous to the relationship between ECMA-402 (ECMAScript internationalization API) and ECMAScript.

This Standard is intended to be used in strict mode only. Sloppy mode has known issues that detract from building a robust system. Sloppy mode is maintained primarily for web compatibility and provides no benefit to embedded systems.

6.2 Class patterns

A Class Pattern, as used in this Standard, is a combination of requirements and guidelines for a class. For example, the IO Class Pattern defines behaviors for all IO classes.

The Standard defines classes in terms of Class Patterns. In the future, there may be true formal classes as found in the ECMAScript Language.

The requirements of a Class Pattern are behaviors defined by this Standard and must be adhered to for a conformant implementation. A Class Pattern can be seen as similar to a collection of Abstract Operations in the ECMA-262.

Guidelines are primarily for extensibility. Extensibility is essential to this Standard as it must be possible to access unique hardware capabilities. Extensibility is problematic because of the potential for collisions. This Standard provides requirements for how extensibility may be implemented.

Unless stated, there are no requirements about class inheritance. An implementation of a class pattern may inherit from **`Object`** or any other class, so long as it conforms.

6.3 Independent implementations

This Standard is intended to facilitate multiple independent implementations of the APIs. A given API may warrant an entirely different implementation depending on a variety of factors that include the host hardware, operating system, and ECMAScript engine.

6.4 Self-hosting

The ECMAScript language is defined in terms of a host that provides the runtime environment for the execution of scripts. This Standard does not change that. The APIs defined herein are provided by a host. However, this Standard does anticipate that portions of the runtime environment provided by the host may themselves be implemented in ECMAScript. This Standard refers to a host that includes ECMAScript code in its implementation as self-hosting.

One challenge of self-hosting is fully separating host scripts from hosted scripts to eliminate security, robustness, and compatibility problems. The Compartment model in the Secure ECMAScript proposal is a tool to separate host scripts from hosted scripts. Compartments also allow separation of modules within a host which mitigates supply-chain attacks.

Self-hosted implementations must ensure that no internal properties or methods are visible to client scripts using the implementation. Private fields and private methods as defined by ECMA-262 are one way to shield internal properties and methods from client code.

NOTE Self-hosting is not required.

6.5 Module specifiers

This Standard defines classes which are accessed through modules. Because many embedded systems lack a file system, using file paths to access modules is impractical and contrived. Instead, modules are accessed using bare module specifiers. While such specifiers are currently forbidden in a web browser, they are permitted in other environments.

A namespace prefix is used to minimize the chance of name collisions with other bare module specifiers. This Standard uses the namespace prefix **`embedded:`**.

```
import Digital from "embedded:io/digital";
```

The “embedded:” namespace prefix is registered as a URI scheme with IANA to reduce the possibility of collisions.

The use of module namespaces in this Standard is intended to be compatible with the Built In Modules Proposal.

For the avoidance of doubt, the use of bare module specifiers by this Standard does not prevent a host from also supporting other kinds of module specifiers for modules not defined by this specification.

6.6 Secure ECMAScript

The Secure ECMAScript (SES) proposal extends the ECMAScript language to support provably secure execution of scripts in an environment that includes both trusted and untrusted scripts. The two foundations of Secure ECMAScript are immutability and compartments. SES makes all primordials immutable prior to the execution of any untrusted script code. This ensures built-in objects behave as defined by the language and disables common attack vectors including prototype poisoning. Compartments allow scripts to sandbox other scripts to limit the globals and modules that are available in the sandbox.

The security guarantees provided by SES reduce vulnerabilities in systems that combine code from multiple sources, some of which may contain security flaws. The mechanisms proposed by SES allow for an efficient implementation. Further, the immutability requirement for SES allows primordials to be stored in read-only memory, reducing RAM use and enabling them to be securely shared by multiple virtual machines.

This Standard is designed to be used with SES when a runtime security solution is required. If and when the SES proposal is an approved standard, this Standard will reference it normatively.

SES consists of two major execution phases — pre-lockdown and post-lockdown. Prior to lockdown, primordials are mutable; afterwards, they are immutable. A host is not required to support pre-lockdown on an embedded system. It may instead complete lockdown during the build process, for example.

6.7 Naming

This Standard uses the lower camel case naming convention (e.g. **`exampleProperty`**) for property names.

It follows the ECMAScript convention of naming classes with upper camel case (e.g. **`ExampleClass`**) and methods with lower camel case (e.g. **`exampleMethod`**).

Callback function names begin with **`on`** (e.g. **`onExampleCallback`**).

Words are preferred over abbreviations and acronyms (e.g. **`address`** instead of **`addr`**, **`clock`** instead of **`scl`**, **`receive`** instead of **`rx`**), though common acronyms are acceptable.

7 Requirements for standard built-in ECMAScript objects

Unless specified otherwise in this document, the objects, functions, and constructors described in this Standard are subject to the generic requirements and restrictions specified for standard built-in ECMAScript objects in ECMA-262, 10th edition, clause 17, or successor.

8 Base Class Pattern

The Base Class Pattern defines common behaviors used by other class patterns. The Base Class Pattern is purely abstract and cannot be instantiated directly.

Classes conforming to the Base Class Pattern may be subclassed.

See Annex A for the formal algorithms of the Base Class Pattern.

8.1 `constructor`

The constructor of the Base Class Pattern takes an options object as its first argument.

The **`target`** property is the only property the Base Class Pattern defines in the options object.

Typically, there are no other arguments as additional configuration options can and should be added to the options object. However, additional arguments are not prohibited.

It is an error to invoke the constructor without the options object. An exception will be thrown.

The implementation of the constructor should validate all supported option properties before allocating any resources. This behavior avoids enabling or changing the state of any hardware should the constructor fail due to invalid parameters.

The implementation must ignore any unrecognized properties on the options object.

If the constructor fails to complete execution successfully, it must release any resources allocated prior to exiting.

The constructor must not modify the options object. It must accept an immutable options object.

Once the instance has been successfully constructed, it must not be eligible for garbage collection until it is explicitly released by calling **`close`**. This is done so scripts do not need to maintain a reference to the object to prevent it from being collected, similar to **`setInterval`**/**`clearInterval`** and the W3C Generic Sensor specification.

8.2 `close` method

The **`close`** method releases all resources associated with the instance.

Once **`close`** completes, the object is eligible for garbage collection.

Once **`close`** completes, an **`Error`** exception is thrown if any other instance methods are called. It is not an error to call the **`close`** method more than once.

No callbacks may be invoked after the **`close`** method is called.

8.3 `target` property

The **`target`** property is opaque to the object's implementation. It may be initialized by the constructor using the **`target`** property in the options object. Scripts may both read and write the target property, though it is typically only set at construction.

8.4 Callbacks

Instances of the Base Class Pattern typically use function callbacks to deliver asynchronous events.

Callback functions are provided to the instance as properties in the options object.

```
new Button({
    onPush() {
    },
    onRelease() {
    }
});
```

Callback functions are invoked with **`this`** set to the instance. This can be overridden using standard ECMAScript features, such as arrow functions:

```
new Button({
    onPush: () => {
    },
    onRelease: () => {
    }
});
```

The callbacks are stored internally by the implementation. They are not public methods. The callback functions cannot be read and are only set using the constructor's options object.

A callback function may only be invoked when no script is running in its host virtual machine to respect the single-thread evaluation semantics of ECMAScript. This means that callbacks may not be invoked by the instance from within its public method calls, including the constructor.

Callbacks must be invoked in the same virtual machine in which they were created.

9 IO Class Pattern

The IO Class Pattern builds on the Base Class Pattern to provide a foundation for implementing access to a variety of hardware inputs and outputs.

All IO is non-blocking, consistent with ECMAScript API behavior on the web platform. That said, not all operations are instantaneous. Implementations determine how long is too long for a given operation.

Non-blocking IO is facilitated by two callback functions, **`onReadable`** and **`onWritable`**, which eliminate the need for polling in most cases.

See Annex A for the formal algorithms of the IO Class Pattern.

9.1 Pin specifier

A **pin specifier** is a JavaScript value used by IO classes to refer to hardware connections represented by pins. Typically, these pins correspond to a particular connection point on the hardware package, although this is not required.

The value of a pin specifier is host-dependent. It is often a number corresponding to the logical GPIO pin number as per the hardware data sheet (e.g. GPIO **`5`**), but it may be a string (**`"D1"`**) or even an object (**`{port: 1, pin: 5}`**).

9.2 Port specifier

A **port specifier** is a JavaScript value used by IO classes to refer to a hardware interface. Port specifier values are defined by the host and are usually either a number or string.

For example, consider a microcontroller may support two serial connections, each with different capabilities that may be configured to be available on a set of pins. The port specifier indicates which serial connection to use.

9.3 constructor

The options object contains the specification of the hardware resources to be used by the instance. For example, the digital class indicates the physical pin to use with a **`pin`** property that has a pin specifier value.

If the constructor requires a resource that is already in use — whether by a script or the native host — an **`Error`** exception is thrown.

This Standard allows but does not require, an implementation to open multiple instances for the same hardware resource if the instances cannot interfere with each other's operation. For example, this can work for a digital input but would not for a digital output.

The IO Class Pattern is designed to be used both with IO types that have only a current value (e.g. Digital, analog, PWM) and IO types that use streams of data (e.g. serial, SPI).

The IO Class Pattern reserves the **`io`** property name in the options object. If present, it must be ignored by IO implementations.

9.4 `read` method

The **`read`** method returns data from the IO instance. If no data is available, it returns **`undefined`**. The type of the data returned depends on the value of the **`format`** property.

The **`read`** method may take any number of arguments, including zero. The arguments are defined by the specific IO type.

If the instance does not support reading (because the IO type is inherently unreadable or because it is configured for write-only) an exception is thrown.

When the **format** property is **"buffer"**, the **read** method may take a data argument that is a **Number**, **ArrayBuffer**, or **TypedArray**. When it is a **Number**, **read** allocates and returns an **ArrayBuffer** with up to as many bytes as the **Number** argument. When it is an **ArrayBuffer** or **TypedArray**, **read** fills in as many bytes as possible and returns a **Number** with the number of bytes read. These two behaviors are provided to allow both efficient and convenient use of read with **"buffer"** because it is very common.

9.5 `write` method

The **write** method sends data to the IO instance.

The following conditions cause an **Error** exception to be thrown: the device cannot accept the data because its buffers are full, the data is incompatible, or a hardware error.

The **write** method may take any number of arguments, including zero. The arguments are defined by the specific IO type. The type of data accepted by **write** depends on the value of the **format** property.

If this instance does not support writing (because the IO type cannot be written or because it is configured for read-only) an **Error** exception is thrown.

9.6 `format` property

The **format** property is a string that indicates the type of data used by the **read** and **write** methods. It is initialized by the constructor to the default defined for its IO type. The **format** property may be set by the script at any time to change how it reads and writes data.

The following values are defined by the IO Class Pattern for the **format** property. IO types may choose to support one or more and may define others.

- **number** - an ECMAScript number value, typically used for bytes
- **buffer** - an **ArrayBuffer** instance. For convenience, **TypedArray** values are accepted as arguments, and the **byteOffset** and **byteLength** properties of the **TypedArray** restrict the bytes accessed. Implementations allocate **ArrayBuffer** instances for return values, never a **TypedArray**.
- **object** - an ECMAScript object, for data representing a data structure (e.g. JSON)
- **string;ascii** - an ECMAScript string, for reading from and writing to IO using 7-bit ASCII data
- **string;utf8** - an ECMAScript string, for reading from and writing to IO using UTF-8 formatted data

The **format** property is implemented as a getter and setter. Attempting to set the **format** property to an unsupported value does not change the value and instead throws an **Error** exception.

9.7 Callbacks

The IO Class Pattern specifies three callbacks which are set by the options object passed to the constructor. Most IO types operate with or without these callbacks installed, but a particular IO type may require one or more callbacks.

9.7.1 onReadable

The **`onReadable`** callback is invoked when the instance has data available to be read. Data is retrieved using the **`read`** method.

The **`onReadable`** callback may receive one or more arguments with information about the data available to read. The arguments are defined by the specific IO type.

The **`onReadable`** callback is invoked once when data arrives and not again until additional data is available to read.

9.7.2 onWritable

The **`onWritable`** callback is invoked when the instance is able to accept more data for output.

The **`onWritable`** callback may receive one or more arguments with information about the amount of data that may be written. The arguments are defined by the specific IO type.

9.7.3 onError

The **`onError`** callback is invoked when a non-recoverable error occurs. The instance is no longer usable. The only method that should be called is **`close`**.

Details of the error may be passed to the callback using arguments defined by the specific IO type.

10 IO classes

This section defines IO Classes conforming to the IO Class Pattern.

The classes support capabilities commonly supported by hardware and runtimes. Capabilities that are not supported here may be added using the extensibility options of the IO Class Pattern and Base Class Pattern.

10.1 Digital

The **`Digital`** IO class is used for digital inputs and outputs.

```
import Digital from "embedded:io/digital";
```

See Annex A for the formal algorithms of the **`Digital`** IO Class.

10.1.1 Properties of constructor options object

Property	Description
`pin`	A pin specifier indicating the pin to control. This property is required.
`mode`	A value indicating the mode of the IO. May be **`Digital.Input`**, **`Digital.InputPullUp`**, **`Digital.InputPullDown`**, **`Digital.InputPullUpDown`**, **`Digital.Output`**, or **`Digital.OutputOpenDrain`**. This property is required.
`edge`	A value indicating the conditions for invoking the **`onReadable`** callback. Values are **`Digital.Rising`**, **`Digital.Falling`**, and **`Digital.Rising	Digital.Falling`**. This value is required if the **`onReadable`** property is present and ignored otherwise.

10.1.2 Callbacks

`onReadable()`

Invoked when the input value changes depending on the value of the **`mode`** property.

10.1.3 Data format

The **`Digital`** class data format is always **`"number"`** with a value of either 0 or 1.

10.1.4 Notes

A digital IO instance configured as an input does not implement write; one configured as an output does not implement read.

10.2 Digital bank

The **`DigitalBank`** class provides simultaneous access to a group of digital inputs or outputs.

```
import DigitalBank from "embedded:io/digitalbank";
```

See Annex A for the formal algorithms of the **`DigitalBank`** bank IO Class.

10.2.1 Properties of constructor options object

Property	Description
`pins`	A bitmask with pins to include in the bank set to 1. This property is required.
`mode`	A value indicating the mode of the IO, May be **`Digital.Input`**, **`Digital.InputPullUp`**, **`Digital.InputPullDown`**, **`Digital.InputPullUpDown`**, **`Digital.Output`**, or **`Digital.OutputOpenDrain`**. All pins in the bank use the same mode. This property is required.
`rises`	A bitmask indicating the pins in the bank that should trigger an **`onReadable`** callback when transitioning from 0 to 1. When an **`onReadable`** callback is provided, at least one pin must be set in **`rises`** and **`falls`**.
`falls`	A bitmask indicating the pins in the bank that should trigger an **`onReadable`** callback when transitioning from 1 to 0. When an **`onReadable`** callback is provided, at least one pin must be set in **`rises`** and **`falls`**.
`bank`	For implementations with more than a single digital bank, a number or string value specifying the digital bank for this instance. This property is optional.

10.2.2 Callbacks

`onReadable(triggers)`

Invoked when the input value changes depending on the value of the **`mode`**, **`rises`**, and **`falls`** properties. The **`onReadable`** callback receives a single argument, **`triggers`**, which is a bitmask indicating each pin that triggered the callback with a 1.

10.2.3 Data format

The **`DigitalBank`** class data format is always **`"number"`**. The value is a bitmask. On a read operation, any bit positions that are not included in the **`pins`** bitmask are set to 0.

NOTE The requirement to zero bit positions not included in the bitmask prevents leaking the state of pins unused by this bank.

10.2.4 Notes

A digital IO bank instance configured as an input does not implement **`write`**; one configured as an output does not implement **`read`**.

A bitmask contains at least one, and not more than, thirty-two bits. Digital banks may distribute their pins across multiple banks using the **`bank`** property of the constructor dictionary.

10.3 Analog input

The **`Analog`** IO class represents an analog input source.

```
import Analog from "embedded:io/analog";
```

See Annex A for the formal algorithms of the **Analog** IO Class.

10.3.1 Properties of constructor options object

Property	Description
pin	A pin specifier indicating the analog input pin. This property is required.
resolution	The requested number of bits of resolution of the input. This property is optional.

10.3.2 Data format

The **Analog** class data format is always a number. The value returned is an integer from 0 to a maximum value based on the resolution of the analog input.

10.3.3 `resolution` property

The read-only **resolution** property indicates the number of bits of resolution provided in values returned by the instance.

10.4 Pulse-width modulation

The **PWM** IO class provides access to the pulse-width modulation capability of pins.

```
import PWM from "embedded:io/pwm";
```

See Annex A for the formal algorithms of the **PWM** IO Class.

10.4.1 Properties of constructor options object

Property	Description
pin	A pin specifier indicating the pin to operate as a PWM output. This property is required.
hz	A number specifying the requested frequency of the PWM output in hertz. This property is optional.

10.4.2 Data format

The **PWM** class data format is always a number. The **write** call accepts integers between 0 and a maximum value based on the resolution of the PWM output.

10.4.3 `resolution` property

The read-only **resolution** property indicates the number of bits of resolution in values passed to the **write** method.

10.4.4 `hz` property

The read-only **`hz`** property returns the frequency of the PWM.

10.4.5 Notes

A PWM instance defaults to a duty cycle of 0% until **`write`** is called with a different value.

10.5 I²C

The **`I2C`** class implements an I²C Initiator to communicate with an I²C Peripheral over I²C bus.

```
import I2C from "embedded:io/i2c";
```

See Annex A for the formal algorithms of the **`I2C`** IO Class.

10.5.1 Properties of constructor options object

Property	Description
`data`	Pin specifier for the I²C data pin. This property is required.
`clock`	Pin specifier for the I²C clock pin. This property is required.
`hz`	The speed of communication on the I²C bus. This property is required.
`address`	The 7-bit address of the target I²C Peripheral to communicate with. This property is required.
`port`	Port specifier for the I²C instance. This property is optional.

NOTE The property name **`timeout`** is reserved for future use.

10.5.2 Data format

The **`I2C`** class data format is always buffer. The **`write`** call accepts an **`ArrayBuffer`** or a **`TypedArray`**. The **`read`** call always returns an **`ArrayBuffer`**.

10.5.3 Specifying stop bit with `read` and `write` methods

The I²C protocol is transaction-based. At the end of each read and write operation, a stop bit is sent. If the stop bit is 1, it indicates the end of the transaction; if 0, it indicates that the transaction has additional operations pending.

The **`read`** and **`write`** methods set the stop bit to 1 by default. An optional argument to the **`read`** and **`write`** methods allows the stop bit to be specified. Pass **`false`** to set the stop bit to 0, and **`true`** to set the stop bit to 1.

10.5.4 Methods

```
read(byteLength | buffer[, stop])
```

The first argument follows the behavior of the IO Class Pattern **`read`** method for the **`"buffer"`** data format. The optional second argument is a **`Boolean`** specifying the stop bit behavior.

write(buffer[, stop])

The first argument to the **write** method is a buffer. The optional second argument is a **Boolean** specifying the stop bit behavior.

NOTE The **read** and **write** methods may operate synchronously. Doing so does not violate the requirement that IO is non-blocking because these operations typically complete within a short period of time. Additionally, synchronous operation is required for microcontrollers which do not support asynchronous I²C IO.

10.6 System management bus (SMBus)

The **SMBus** class extends the **I2C** class with additional methods to communicate with devices that implement the SMBus protocol.

```
import SMBus from "embedded:io/smbus";
```

See Annex A for the formal algorithms of the **SMBus** IO Class.

10.6.1 Properties of constructor options object

Property	Description
stop	A boolean value indicating whether to set the stop bit when writing the SMBus register number. This property is optional and defaults to **false**.

10.6.2 Methods

readUint8(register)

Reads and returns an unsigned 8-bit integer value from the specified register.

writeUint8(register, value)

Writes the unsigned 8-bit integer **value** to the specified register.

readUint16(register[, bigEndian])

Reads and returns an unsigned 16-bit integer value from the specified register. By default, the value is read in little-endian byte order. If the optional **bigEndian** argument is **true** the value is read in big-endian byte order.

writeUint16(register, value[, bigEndian])

Writes the unsigned 16-bit integer value to the specified register. By default, the value is written in little-endian byte order. If the optional **bigEndian** argument is **true** the value is written in big-endian byte order.

readBuffer(register, byteLength | buffer)

Reads a stream of bytes starting at the specified **register**. The second argument to **readBuffer** follows the behavior of the IO Class Pattern **read** method for the **"buffer"** data format.

writeBuffer(register, buffer)

Write a stream of bytes from the **ArrayBuffer** instance in the **buffer** argument starting at the specified **register**. The number of bytes written is equal to the **byteLength** of the buffer.

NOTE The method names **`readUint32`**, **`writeUint32`**, **`readUint64`**, and **`writeUint64`** are reserved for 32 and 64-bit SMBus operations in the future.

10.7 Serial

The **`Serial`** class implements bi-directional serial (UART) communication.

```
import Serial from "embedded:io/serial";
```

See Annex A for the formal algorithms of the **`Serial`** IO Class.

10.7.1 Properties of constructor options object

Property	Description
`receive`	Pin specifier for the receive pin. This property is required by some implementations to use the serial connection to read data.
`transmit`	Pin specifier for the transmit pin. This property is required by some implementations to use the serial connection to write data.
`baud`	A number specifying the baud rate of the connection. This property is required.
`flowControl`	A string specifying the kind of flow control, if any, used on the connection. The valid values are **`"hardware"`** and **`"none"`**. This property is optional and defaults to **`"none"`**.
`dataTerminalReady`	Pin specifier for the data terminal ready pin. This property is optional.
`requestToSend`	Pin specifier for the request to send pin. This property is optional.
`clearToSend`	Pin specifier for the clear to send pin. This property is optional.
`dataSetReady`	Pin specifier for the data set ready pin. This property is optional.
`port`	Port specifier for the serial connection. This property is optional.

NOTE The serial connection is eight data bits, no parity bit, and one stop bit (8N1). The property names **`parity`**, **`stop`**, and **`data`** are reserved to support other communication configurations in the future.

10.7.2 Methods

```
read([byteLength | buffer])
```

When using the **`"number"`** data format, read always returns the next available byte as a **`Number`** (from 0 to 255).

When using the **"buffer"** data format, **read** follows the behavior of the IO Class Pattern **read** method for the **"buffer"** data format with one addition: if there are no arguments, **read** returns one or more bytes (implementation-dependent).

If no data is available, **read** returns **undefined**.

The **read** method must not wait for additional bytes to arrive.

write(byteValue | data)

When using the **"number"** data format, the first argument is a byte value to transmit. If the output buffer is full, **write** throws.

When using the **"buffer"** data format, the first argument is an **ArrayBuffer** or **TypedArray** containing one or more bytes to transmit. If the output buffer cannot accept all the bytes in the buffer, an exception is thrown – partial data must not be written.

flush([input, output])

Flushes the input and/or output queues of the serial instance. If no arguments are passed, both input and output queues are flushed. If both arguments are provided, the corresponding queues are flushed based on the value of the arguments. An exception is thrown if one argument is passed.

If flushing the output causes the serial instance to be able to accept data for output, the **onWritable** callback will be invoked.

set(options)

The **set** method controls the value of the data terminal ready and request to send pins of the serial connection together with the break. The sole argument is an options object which contains optional **dataTerminalReady**, **requestToSend**, and **break** properties with boolean values.

If **dataTerminalReady**, **requestToSend**, or **break** is not specified in the dictionary, the corresponding serial behavior is left unchanged.

get([options])

The **get** method returns the value of the clear to send and data set ready pins. It returns the state of the pins as booleans in an options object using the **clearToSend** and **dataSetReady** properties.

If the optional options object property is provided, **get** sets the **clearToSend** and **dataSetReady** properties on the options object and returns the provided options object as the result of **get**.

10.7.3 Callbacks

onReadable(bytes)

The **onReadable** callback is invoked when new data is available to read. The callback receives a single argument that indicates the number of bytes available.

onWritable(bytes)

The **onWritable** callback is first invoked when the serial instance is ready for use.

The **onWritable** callback is invoked when space has been freed in the output buffer. The callback receives a single argument that indicates the number of bytes that may be written without overflowing the output buffer.

10.7.4 Data format

The **`Serial`** class data format is either **`"number"`** for individual bytes or **`"buffer"`** for groups of bytes. The default data format is **`"number"`**. When using the **`"buffer"`** format, the **`write`** call accepts an **`ArrayBuffer`** or a **`TypedArray`**, and the **`read`** call always returns an **`ArrayBuffer`**.

10.8 Serial Peripheral Interface (SPI)

The **`SPI`** class implements a Serial Peripheral Interface (SPI) controller to communicate with a single SPI peripheral.

```
import SPI from "embedded:io/spi";
```

See Annex A for the formal algorithms of the **`SPI`** IO Class.

10.8.1 Properties of constructor options object

Property	Description
`out`	Pin specifier for the Serial Data Out pin. This property is required when using the SPI bus to write data.
`in`	Pin specifier for the Serial Data In pin. This property is required when using the SPI bus to read data.
`clock`	Pin specifier for the clock pin. This property is required.
`select`	Pin specifier for the chip select pin. This property is optional and should not be specified if chip select will be managed by the caller.
`active`	The value to write to the **`select`** pin when the SPI instance is active. Must be 1 or 0. This property is optional and defaults to 0.
`hz`	The speed of communication on the SPI bus. This property is required.
`mode`	The SPI bus mode, a two-bit mask that specifies the SPI clock polarity (bit 1) and phase (bit 0). This property is optional and defaults to 0b00.
`port`	Port specifier for the SPI connection. This property is optional.

If both **`out`** and **`in`** are unspecified, a **`TypeError`** is thrown by the constructor during validation.

The **`in`** and **`out`** properties may refer to the same physical pin (e.g. 3-wire SPI).

10.8.2 Data format

The data format for the **`SPI`** class is always **`"buffer"`**.

10.8.3 Methods

```
read(byteLength | buffer)
```

The first argument follows the behavior of the IO Class Pattern **`read`** method for the **`"buffer"`** data format.

If the **buffer** argument has a **bitLength** property, it specifies the number of bits to read, overriding the **byteLength** property to allow reading of partial bytes. **buffer.bitLength** must be less than or equal to the number of bits in the buffer (i.e. **buffer.byteLength * 8**). Bits are read into the start of **buffer** (i.e. bit offset zero).

The behavior of the Serial Data Out pin is implementation-dependent during the read operation.

write(buffer)

Write **buffer** to the SPI bus. Any input data is discarded.

If the **buffer** argument has a **bitLength** property, it specifies the number of bits to write, overriding the **byteLength** property to allow writing of partial bytes. **buffer.bitLength** must be less than or equal to the number of bits in the buffer (i.e. **buffer.byteLength * 8**). Bits are written from the start of **buffer** (i.e. bit offset zero).

transfer(buffer)

Write **buffer** to the SPI bus while simultaneously reading **buffer.byteLength** 8-bit bytes from the SPI bus. The results of the read are placed into **buffer**, replacing the original contents.

If the **buffer** argument has a **bitLength** property, it specifies the number of bits of the buffer to swap in the transfer, overriding the **byteLength** property to allow transfer of partial bytes. **buffer.bitLength** must be less than or equal to the number of bits in the buffer (i.e. **buffer.byteLength * 8**). Bits are transferred from the start of **buffer** (i.e. bit offset zero).

flush([deselect])

Flushes any buffers of the SPI controller instance. The flush operation is synchronous and completes before returning.

Some SPI peripherals require that the chip select pin be set inactive at specific times (for instance, to mark the end of a transaction). The **flush** method supports this with the optional **deselect** argument which, when present and **true**, causes the chip select pin to be set to inactive after the flush completes.

10.9 Pulse count

The **PulseCount** class implements a bi-directional counter typically used with a rotary encoder.

```
import PulseCount from "embedded:io/pulsecount";
```

See Annex A for the formal algorithms of the **PulseCount** IO Class.

10.9.1 Properties of constructor options object

Property	Description
signal	Pin specifier for the signal input pin. This property is required.
control	Pin specifier for the control input pin. This property is required.

10.9.2 Data format

The **PulseCount** class data format is always a number. The values are always integers.

10.9.3 Methods

`read()`

The **`read`** method returns the current count. It takes no arguments.

The count is initialized to zero at the time of instantiation. Note that the initial call to **`read`** may return a non-zero value if pulses have been counted in the intervening interval.

`write(count)`

The **`write`** method sets the current count.

10.9.4 Callbacks

`onReadable()`

The **`onReadable`** callback is invoked when the value of the counter has changed. Multiple changes to the counter may be combined into a single invocation of the callback.

`onError()`

The **`onError`** callback is invoked when an error is detected, for example, underflow or overflow of the counter.

10.10 TCP socket

The **`TCP`** network socket class implements a general-purpose, bi-directional TCP connection.

```
import TCP from "embedded:io/socket/tcp";
```

The TCP socket is not a TCP listener, as in some networking libraries. The TCP listener is a separate class.

See Annex A for the formal algorithms of the **`TCP`** IO Class.

10.10.1 Properties of constructor options object

Property	Description
address	A string with the IP address of the remote endpoint to connect to. Either the **address** or **host** property must be provided.
host	A string with the DNS name of the remote endpoint to connect to. Either the **address** or **host** property must be provided.
port	A number specifying the remote port to connect to. This property is required.
noDelay	A boolean indicating whether to disable Nagle's algorithm on the socket. This property is equivalent to the **TCP_NODELAY** option in the BSD sockets API. This property is optional and defaults to false.
keepAlive	A number specifying the keep-alive interval of the socket in milliseconds. This property is optional and if not present, the keep-alive capability of the socket is not used.
from	An existing TCP socket instance from which the native socket instance is taken to use with the newly created socket instance. This property is optional and intended for use with a TCP listener. When the **from** property is present, the **address**, **host**, and **port** properties are not required and are ignored if specified. The original instance is closed with ownership of the native socket transferred to the new instance.

10.10.2 Methods

read((byteLength | buffer))

When using the **"number"** data format, read always returns the next available byte as a **Number** (from 0 to 255).

When using the **"buffer"** data format, **read** follows the behavior of the IO Class Pattern **read** method for the **"buffer"** data format with one addition: if there are no arguments, **read** returns one or more bytes (implementation-dependent).

The **read** method must not wait for additional bytes to arrive.

write(byteValue | buffer)

When using the **"number"** data format, the first argument is a byte value to transmit. If the output buffer is full, **write** throws.

When using the **"buffer"** data format, the first argument is an **ArrayBuffer** or **TypedArray** containing one or more bytes to transmit. If the output buffer cannot accept all the bytes in the buffer, an exception is thrown – partial data must not be written.

10.10.3 Callbacks

`onReadable(bytes)`

Invoked when new data is available to be read. The callback receives a single argument that indicates the number of bytes available to read.

`onWritable(bytes)`

Invoked when space has been made available to output additional data. The callback receives a single argument that indicates the total number of bytes that may be written to the TCP socket without overflowing the output buffers.

The **`onWritable`** callback is first invoked when the socket successfully connects to the remote endpoint and it is possible to write data.

`onError()`

The **`onError`** callback is invoked when an error occurs or the TCP socket disconnects. Once **`onError`** is invoked, the connection is no longer usable. Reporting the error type is an area for future work.

10.10.4 Data format

The **`TCP`** class data format is either **`"number"`** for individual bytes or **`"buffer"`** for groups of bytes. The default data format is **`"buffer"`**. When using the **`"buffer"`** format, the **`write`** call accepts an **`ArrayBuffer`** or a **`TypedArray`**. The **`read`** call always returns an **`ArrayBuffer`**.

10.11 TCP listener socket

The TCP **`Listener`** class listens for and accepts incoming TCP connection requests.

```
import Listener from "embedded:io/socket/listener";
```

See Annex A for the formal algorithms of the **`Listener`** IO Class.

10.11.1 Properties of constructor options object

Property	Description
`port`	A number specifying the port to listen on. This property is optional.
`address`	A string with the IP address of the network interface to bind to. This property is optional.

10.11.2 Methods

`read()`

The **`read`** function returns a **`TCP`** Socket instance. The instance is already connected to the remote endpoint. There are no callback functions installed.

NOTE To set the callbacks and configure the socket, pass the socket to the **`TCP`** Socket constructor using the **`from`** property.

`write()`

Unsupported.

10.11.3 Callbacks

`onReadable(requests)`

Invoked when one or more new connection requests are received. The callback receives a single argument that indicates the total number of pending connection requests.

10.11.4 Data format

The TCP **`Listener`** class uses **`socket/tcp`** as its sole data format.

10.12 UDP socket

The **`UDP`** network socket class implements the sending and receiving of UDP packets.

```
import UDP from "embedded:io/socket/udp";
```

See Annex A for the formal algorithms of the **`UDP`** IO Class.

10.12.1 Properties of constructor options object

Property	Description
`port`	The local port number to bind the UDP socket to. This property is optional.
`address`	A string with the IP address of the network interface to bind to. This property is optional.
`multicast`	A string with the IP address of a multicast address to bind to. This property is optional.
`timeToLive`	A number with the multicast time-to-live value as a number from 1 to 255. This property is required if the **`multicast`** property is provided and otherwise ignored.

10.12.2 Methods

`read([buffer])`

The **`read`** call reads a complete UDP packet.

If there are no arguments, **`read`** allocates an **`ArrayBuffer`** the size of the packet, copies the packet data to the buffer, and returns the buffer. If first argument is an **`ArrayBuffer`** or **`TypedArray`**, the packet data is copied to the buffer and the number of bytes copied is returned. If the buffer is too small to hold the packet, an exception is thrown.

The following properties are attached to the buffer containing the packet data:

- **`address`**, a string containing the packet sender's IP address

- **port**, the port number used to send the packet.

```
write(buffer, address, port,)
```

The **write** call takes three arguments: the packet data as an **ArrayBuffer** or **TypedArray**, the remote address string, and the remote port number. If there is insufficient memory to transmit the packet, the **write** call throws an exception.

10.12.3 Callbacks

```
onReadable(packets)
```

Invoked when one or more packets are received. The callback receives a single argument that indicates the total number of packets available to read.

10.12.4 Data format

The **UDP** class data format is always **"buffer"**. The **write** call accepts an **ArrayBuffer** or a **TypedArray**. The **read** call always returns an **ArrayBuffer**.

11 IO Provider Class Pattern

The IO Provider Class Pattern builds on the Base Class Pattern to provide a foundation to access a collection of IO Classes.

An IO Provider contains one or more IO Classes. The IO Provider may be connected to the host in any way, including:

- A direct hardware connection such as I²C or SPI
- A local wireless connection such as BLE using the Automation IO Service profile
- A TCP/IP connection to an internet cloud service

It is anticipated, but not required, that implementations of the IO Provider Class Pattern will perform IO using instances conforming to the IO Class Pattern. To facilitate that, the constructor uses IO constructor properties to specify their IO connections.

An IO Provider instance contains IO Classes which conform to the IO Class Pattern. The following code is an example of using an IO Provider to access a Digital pin on a GPIO expander connected via I²C.

```
import I2C from "embedded:io/i2c";

const expander = new Expander({
    io: I2C,
    data: 5,
    clock: 4,
    hz: 1_000_000,
    address: 0x20,
});

const led = new expander.Digital({
    pin: 13,
    mode: expander.Digital.Output,
});
led.write(1);
```

Here the **`data`** and **`clock`** pins passed to the **`Expander`** constructor refer to pins of the host whereas the **`pin`** passed to the **`expander.Digital`** constructor refers to a pin of the GPIO expander.

See Annex A for the formal algorithms of the IO Provider Class Pattern.

11.1 `constructor`

Following the Base Class Pattern, the constructor has a single options object argument. The options object defines the hardware connections of the sensor. These use the same properties as the IO types corresponding to the hardware connection. As in the Peripheral Class Pattern, the IO properties in the Provider Class Pattern are grouped to avoid collisions.

The options object is not limited to IO connection information and must contain all information needed by the implementation to establish the connection.

11.2 `close` method

In addition to releasing all resources as required by the Base Class Pattern, the **`close`** method causes the **`onError`** callback to be invoked on all open instances. Note that **`onError`** may not be invoked from within **`close`** (see Callbacks section).

11.3 Callbacks

`onReady()`

The **`onReady`** callback is invoked once the IO Provider instance is ready for use.

The IO provider may not know what IO resources are available until it has successfully established a connection to the remote resource. For this reason, a provider may not have any IO constructors on its instance until the **`onReady`** is invoked.

The IO constructors of an IO Provider, if present on the instance, may be used prior to **`onReady`** being invoked.

`onError()`

The **`onError`** callback is invoked on a non-recoverable error to indicate that the provider instance can no longer be used.

When a provider fails, its IO instances also become unusable, and consequently **`onError`** must also be invoked on each instance.

12 Peripheral Class Pattern

The Peripheral Class Pattern builds on the Base Class Pattern to provide a foundation for implementing access to different kinds of peripheral devices. The Peripheral Class Pattern is purely abstract and cannot be instantiated directly.

See Annex A for the formal algorithms of the Peripheral Class Pattern.

12.1 `constructor`

Following the Base Class Pattern, the constructor has a single options object argument. The options object defines the hardware connections of the peripheral. These use the same properties as the IO types corresponding to the hardware connection. For example, an I²C peripheral:

```
import I2CPeripheral from "embedded:example/i2cperipheral";
import I2C from "embedded:io/i2c";

let t = new I2CPeripheral({
    io: I2C,
    data: 4,
    clock: 5,
    address: 0x30
});
```

The **io** property specifies the constructor for the IO Class.

If the peripheral has multiple hardware connections, the options object separates them to avoid collisions. For example, here the peripheral has an I²C connection for primary communication and a digital connection for an interrupt:

```
import I2CPeripheralWithInterrupt from
"embedded:example/i2cperipheralwithinterrupt";
import I2C from "embedded:io/i2c";
import Digital from "embedded:io/digital";

let t = new I2CPeripheralWithInterrupt({
    communication: {
        io: I2C,
        data: 4,
        clock: 5,
        address: 0x30
    },
    interrupt: {
        io: Digital,
        pin: 5
    }
});
```

The constructor must reset the peripheral hardware to a consistent initial state so the peripheral's behavior is not dependent on a previous instantiation. This reset may include calling the instance's **configure** method.

12.2 `close` method

The **close** method, as required by the Base Class Pattern, releases all IO connections in use by the instance.

12.3 `configure` method

The **configure** method modifies how the peripheral operates. It has a single argument, an options object.

The **configure** method follows the same rules regarding the options argument as the **constructor** and therefore may not modify its content.

Because peripherals have many features, the **configure** method may implement support for many properties. A given call to the **configure** method should only modify the features specified in the options object.

The Peripheral Class Pattern does not require a script call the **configure** method to use the peripheral, however specific implementations may require **configure** to be called.

The **configure** method may be called more than once to allow scripts to reconfigure the peripheral.

12.4 Accessors for configuration

Classes that follow the Peripheral Class Pattern may choose to provide accessors, e.g. setters and getters, for configuration properties. A setter should behave in the same way as the **`configure`** method invoked with a single property. For example, a setter for a property named **`resolution`** could be implemented as follows:

```
class ExamplePeripheral {
    ...
    set resolution(value) {
        this.configure({resolution: value});
    }
}
```

A getter for the same property could be implemented as follows:

```
class ExamplePeripheral {
    ...
    get resolution() {
        this.configuration.resolution;
    }
}
```

13 Sensor Class Pattern

The Sensor Class Pattern builds on the Peripheral Class Pattern to provide a foundation for implementing access to a variety of sensors.

It is anticipated, but not required, that instances conforming to the Sensor Class Pattern will perform IO using instances conforming to the IO Class Pattern. The Sensor Class Pattern is therefore non-blocking, like IO. Additionally, the constructor uses IO constructor properties to specify their IO connections.

The Sensor Class Pattern provides low-level sensor access, similar to a sensor driver provided by a sensor manufacturer, to support access to all the unique capabilities of the sensor. As with IO, where a given type of device (e.g. a temperature sensor) has common capabilities across manufacturers, the individual sensor types define a common way to access that functionality.

Higher-level sensor APIs may be built using instances of the Sensor Class Pattern. The W3C Generic Sensor specification, for example, may be implemented using sensors conforming to The Sensor Class Pattern.

The Sensor Class Pattern may be used together with the Sensor Data Provenance Rules to improve the usability of the data collected.

See Annex A for the formal algorithms of the Sensor Class Pattern.

13.1 `constructor`

Following the Peripheral Class Pattern, the constructor has a single options object argument. The options object defines the hardware connections of the sensor.

For example, here the temperature sensor has an interrupt on a Digital pin:

```
import I2C from "embedded:io/i2c";
import Digital from "embedded:io/digital";

let t = new Temperature({
    sensor: {
        io: I2C,
        data: 4,
        clock: 5,
        address: 0x30
    },
    interrupt: {
        io: Digital,
        pin: 5
    }
});
```

The constructor must reset the sensor hardware to a consistent initial state so the sensor's behavior is not dependent on a previous instantiation.

13.2 `configure` method

The **`configure`** method is inherited from the Peripheral Class Pattern. For sensors, it modifies how the sensor operates. This may include the hardware's sampling interval, what data is sampled, and the range of the data sampled.

13.3 `sample` method

The **`sample`** method returns readings from the sensor. The Sensor Class Pattern defines no arguments for the **`sample`** method, though individual sensor types may.

The **`sample`** method returns an object containing one or more properties. The returned object is mutable. The implementation must return a different object on each invocation to allow calls to accumulate multiple sensor readings.

NOTE A sensor implementation of **`sample`** may accept an input argument of the object to use for the sensor data as an optimization to reduce memory manager work. If supported, this must be specified for the Sensor Class' **`sample`** method.

If the sample data includes timestamps (e.g. when the sample was collected), those timestamps in the returned sample object should conform to the **`time`** or **`ticks`** properties of the Sample Object specified by the Provenance Sensor Class Pattern.

13.4 Callbacks

The Sensor Class Pattern specifies one callback that is set by the options object passed to the constructor. Individual sensor classes may provide additional callbacks, for instance, to indicate when a sample is available or a sensed condition has been met.

`onError()`

The **`onError`** callback is invoked on a non-recoverable error to indicate that the sensor instance can no longer be used. The only method that should be called is **`close`**.

14 Sensor classes

This section defines Sensor Classes conforming to the Sensor Class Pattern.

The classes support common sensor capabilities. Capabilities that are not supported here may be added using the extensibility options of the Sensor Class Pattern and Base Class Pattern.

14.1 Accelerometer

The **`Accelerometer`** class implements access to a three-dimensional accelerometer.

See Annex A for the formal algorithms of the **`Accelerometer`** sensor class.

14.1.1 Properties of a sample object

These properties are compatible with the attributes of the same name in the W3C Accelerometer draft.

Property	Description
x	A number that represents the sampled acceleration along the x axis in meters per second squared. This property is required.
y	A number that represents the sampled acceleration along the y axis in meters per second squared. This property is required.
z	A number that represents the sampled acceleration along the z axis in meters per second squared. This property is required.

14.2 Ambient light

The **`AmbientLight`** class implements access to an ambient light sensor.

See Annex A for the formal algorithms of the **`AmbientLight`** sensor class.

14.2.1 Properties of sample object

These properties are compatible with the attributes of the same name in the W3C Ambient Light Sensor draft.

Property	Description
`illuminance`	A number that represents the sampled ambient light level in Lux. This property is required.

14.3 Atmospheric pressure

The **`AtmosphericPressure`** class implements access to an atmospheric pressure sensor or barometer.

See Annex A for the formal algorithms of the **`AtmosphericPressure`** sensor class.

14.3.1 Properties of a sample object

Property	Description
`pressure`	A number that represents the sampled atmospheric pressure in Pascal. This property is required.

14.4 Humidity

The **`Humidity`** class implements access to a humidity sensor.

See Annex A for the formal algorithms of the **`Humidity`** sensor class.

14.4.1 Properties of a sample object

Property	Description
`humidity`	A number that represents the sampled relative humidity as a percentage. This property is required.

14.5 Proximity

The **`Proximity`** class implements access to a proximity sensor or range finder.

See Annex A for the formal algorithms of the **`Proximity`** sensor class.

14.5.1 Properties of a sample object

These properties are compatible with the attributes of the same name in the W3C Proximity Sensor draft.

Property	Description
`near`	A boolean that indicates if a proximate object is detected. This property is required.
`distance`	A number that represents the distance to the nearest sensed object in centimeters or **`null`** if no object is detected. This property is optional: some proximity sensors can only provide the **`near`** property.
`max`	A number that represents the maximum sensing range of the sensor in centimeters.

14.6 Temperature

The **`Temperature`** class implements access to a temperature sensor.

See Annex A for the formal algorithms of the **`Temperature`** sensor class.

14.6.1 Properties of a sample object

Property	Description
`temperature`	A number that represents the sampled temperature in degrees Celsius. This property is required.

14.7 Touch

The **`Touch`** class implements access to a touch panel controller.

See Annex A for the formal algorithms of the **`Touch`** sensor class.

14.7.1 Sample object

The **`Touch`** class **`sample`** method returns an array of **`touch`** objects, as specified below. If there is no touch in progress, **`sample`** returns **`undefined`**.

14.7.1.1 Properties of touch object

Property	Description
`x`	Number indicating the X coordinate of the touch point
`y`	Number indicating the Y coordinate of the touch point
`id`	Number indicating which touch point this entry corresponds to

15 Display Class Pattern

The Display Class Pattern builds on the Peripheral Class Pattern to provide a foundation for implementing access to displays represented by a two-dimensional array of pixels.

The Display Class Pattern is designed to support displays independent of hardware architecture. For example, it may be used efficiently with both frame buffers stored in local host memory and frame buffers connected with the MIPI Display Serial Interface.

See Annex A for the formal algorithms of the Display Class Pattern.

15.1 constructor

Following the Peripheral Class Pattern, the constructor has a single options object argument. The options object defines the hardware connections of the display. These use the same properties as the IO types corresponding to the hardware connection.

A Display Class is not required to have properties to configure its hardware connections. For example, a memory-mapped display may have no external connections. Or, a Display Class may be preconfigured for the hardware of a specific host.

15.2 `configure` method

The following table enumerates the properties defined for the options object argument:

Property	Description
format	A number indicating the format of pixel data passed to the instance (for example, to the **send** method). This property is optional. If the format provided is not supported by the Display Class, a **RangeError** is thrown.
rotation	The clockwise rotation of the display as a number. This property is optional. If the value provided is not 0, 90, 180, or 270, or is unsupported by the Display Class, a **RangeError** is thrown.
brightness	The relative brightness of the display from 0 (off) to 1.0 (full brightness). This property is optional.
flip	A string indicating whether the pixels should be flipped horizontally and/or vertically. Allowed values are **""**, **"h"**, **"v"**, and **"hv"**. The empty string indicates that neither horizontal nor vertical flip is applied. This property is optional.

Note that no default values are defined by the Display Class Pattern for these configuration properties to allow the host to provide default values that are appropriate for its hardware.

Proposed Because the defaults may be configured by the host, it is sometimes useful to retrieve the current configuration. One solution is to return the current configuration from the **configure** call, perhaps as an option signalled by a property (e.g. **{get: true}**). This approach is convenient for scripts that need to check if particular configuration options are applied. Another approach is to have a separate method (e.g. **getConfiguration()**) to return the current configuration.

15.3 begin method

The **begin** method starts the process of updating the display's pixels. If no arguments are passed, the entire frame buffer is updated starting at the top-left corner (coordinate **{0, 0}**), proceeding left-to-right, top-to-bottom, ending at the bottom-right corner (coordinate **{width, height}**).

If an options object is passed as the sole argument, the object may contain **x**, **y**, **width**, and **height** properties that define a rectangular area to update. The rectangle must fit within the bounds of the display (e.g. **{0, 0, width, height}**) or a **RangeError** is thrown.

A display may not support all possible update areas. For example, a display may only support updates aligned to even horizontal pixels. A **RangeError** is thrown if an unsupported update area is passed to **begin**. Prior to calling begin, the **adaptInvalid** method may be used to adjust the update area to the capabilities of the display.

The options object has an optional **continue** property to support discontiguous updates on displays that use page flipping to swap between multiple frame buffers. When **continue** is **false**, the default value, the call to the **begin** method starts to update a new frame. Calling **begin** with **continue** set to **true** continues updating the same frame rather than starting a new one.

An **Error** exception is thrown if the **begin** method is called more than once without an intervening call to the **end** method, unless **continue** is set to true in the successive calls. For example, this is a valid call sequence to update three horizontal slices of the display.

```
display.begin({x: 0, y: 0, width: 240, height: 10});
display.send(pixels);
display.begin({x: 0, y: 20, width: 240, height: 10, continue: true});
display.send(pixels);
display.begin({x: 0, y: 40, width: 240, height: 10, continue: true});
display.send(pixels);
display.end();
```

15.4 `send` method

The **`send`** method delivers one or more horizontal scan lines of pixel data to the display. The sole argument to **`send`** is a buffer of pixels stored either in an **`ArrayBuffer`** or an ArrayBuffer view. The pixels are stored in a packed array with no padding between scan lines. The format of the pixels matches the **`format`** property of the options object of the **`configure`** method.

15.5 `end` method

The **`end`** method finishes the process of updating the display's pixels, by making all pixels visible on the display. If the display instance buffers pixels, all pixels musts be flushed. If the display uses page flipping, the page must be flipped to the most recently updated buffer.

15.6 `adaptInvalid` method

The **`adaptInvalid`** method accepts a single options object argument that includes **`x`**, **`y`**, **`width`**, and **`height`** properties that describe an area of the display to be updated. It adjusts these properties as necessary so that the result is valid for the display and encloses the original update area.

Consider a display which limits the update area horizontally to even pixel positions. The following code calls a display's **`adaptInvalid`** method with odd numbers for both left and right edges of the update area:

```
const area = {x: 3, y: 20, width: 10, height: 20};
display.adaptInvalid(area);
display.begin(area);
display.send(pixels);
display.end();
```

An implementation of **`adaptInvalid`** to apply the rules above, if implemented in JavaScript, would be:

```
function adaptInvalid(options) {
    if (options.x & 1) {
        options.x -= 1;
        options.width += 1;
    }
    if (options.width & 1) {
        options.width += 1;
    }
}
```

Some displays require that the update area only include full scan lines. The following function shows the implementation for such a display, assuming a scanline width of 128 pixels:

```
function adaptInvalid(options) {
    options.x = 0;
    options.width = 128;
}
```

For displays that only support full screen updates, **`adaptInvalid`** updates the rectangle to be the full display dimensions. The following function shows the implementation for a QVGA (320 x 240) display:

```
function adaptInvalid(options) {
    options.x = 0;
    options.y = 0;
    options.width = 320;
    options.height = 240;
}
```

15.7 Instance properties

`width`

The width of the display in pixels as a number. This property is read-only. This value may change based on the configuration, for example, when changing the rotation causes the orientation to change from portrait to landscape.

`height`

The height of the display in pixels as a number. This property is read-only. This value may change based on the configuration, for example, when changing the rotation causes the orientation to change from portrait to landscape.

15.8 Pixel `format` values

Property	Description
3	1-bit monochrome
4	4-bit grayscale (0 black, 15 white)
5	8-bit grayscale (0 black, 255 white)
6	8-bit RGB 3:3:2
7	16-bit RGB 5:6:5 little-endian
8	16-bit RGB 5:6:5 big-endian
9	24-bit RGB 8:8:8
10	32-bit RGBA 8:8:8:8
12	12-bit xRGB 4:4:4:4 (x is unused)

16 Host provider instance

The Host Provider instance aggregates data and code available to scripts from the host. The host provider instance is available as a module import:

```
import device from "embedded:provider/builtin";
```

The Host Provider instance is instantiated before hosted scripts are executed. Only a single instance of the host provider may be created, and the host provider cannot be closed or garbage collected.

The following sections define properties of the Host Provider instance. The Host Provider instance has no required properties.

16.1 Global variable

Hosts are not required to make the host provider instance available in a global variable. A host that does should use the global variable named **`device`**.

16.2 Pin name property

The **`pins`** property is an object that maps pin names to pin specifiers. More than one pin name may map to the same pin specifier.

```
import Digital from "embedded:io/digital";

let led = new Digital({
    pin: device.pin.led,
    mode: Digital.Output
})
```

16.3 IO bus properties

An IO Bus is two or more pins used to implement a communication protocol such as Serial, SPI, or I²C. There may be one or more instances of an IO Bus and one may be designated as the default bus of that type.

The Host Provider instance may contain properties corresponding to each bus type. The following bus types are defined for those host provider instance.

Bus Type	Property Name
I²C	**`i2c`**
Serial	**`serial`**
SPI	**`spi`**

Each bus type may contain one or more buses. Each bus may have one or more names. It is recommended to provide a property named **`default`** when there is a default bus.

```
// example host implementation
const A = {
    in: 12,
    out: 13,
    clock: 14,
    select: 15,
    hz: 10_000_000
};

const B = {
```

```
    in: 0,
    out: 1,
    clock: 2,
    select: 3,
    hz: 20_000_000
};

device.spi = {
    A,
    B,
    default: B
}

// example hosted script use

import SPI from "embedded:io/spi";

let spi = new SPI(device.spi.default);
```

16.4 IO classes

The host provider instance may provide access to its IO constructors through its **`io`** property. This is analogous to the IO constructors available from an IO Provider.

```
// example host provider implementation

import Digital from "embedded:io/digital";
import I2C from "embedded:io/i2c";
import SPI from "embedded:io/spi";

export default {
    pin: {
        button: 0,
        led: 2
    },
    io: {
        Digital,
        I2C,
        SPI
    }
};

// example hosted script use

import device from "embedded:provider/builtin";

let spi = new device.io.SPI(device.spi.default);
```

16.5 IO Providers

The host provider instance should include its IO Provider constructors in its **`provider`** property.

16.6 Sensors

The host provider instance should include its Sensor constructors in its **`sensor`** property.

16.7 Displays

The host provider instance should include its Display constructors through its **`display`** property.

17 Provenance Sensor Class Pattern

Sensor data provenance is metadata associated with sensor samples. It encapsulates the specific, instance source of data, the data transmission mechanism(s), and data transformations occurring at any point between the sensor and the end-user or end-use application. Provenance applies both to direct and synthetic measurements.

This section specifies the Provenance Sensor Class Pattern, which builds on the Sensor Class Pattern by specifying an API for making sensor metadata available to scripts.

The Provenance Sensor Class Pattern adds one optional property to the **`constructor`** options object, two required instance properties, and three properties to the object returned by the **`sample`** method.

The additions the Provenance Sensor Class Pattern makes to the Sensor Class Pattern are a lightweight means of enabling provenance-aware scripts using Sensor Classes. Provenance-aware scripts may support more robust analytics and/or high-assurance tasks.

A separate Technical Report, ECMA TR/110, Recommendations and best practices for scripts on connected sensing devices, describes the best practices for using the Provenance Sensor Class Pattern to support scripts running on connected sensing devices, for propagating static and dynamic device and state metadata, and for accurately propagating sensor samples.

17.1 Properties of constructor options object

Property	Description
`onConfiguration`	Callback to invoke when a new sensor configuration has been applied. The configuration details are obtained from the **`configuration`** property of the instance. This property is optional.

The **`onConfiguration`** callback is invoked whenever configuration parameters are changed from the originally-constructed instance.

17.2 `configuration` property

The required read-only **`configuration`** property indicates the current configuration of the sensor. Non-default values must be reported. All configured parameters may optionally be included.

The data format of this property is implementation-dependent. For instance, the data may be a binary value or may be human-readable. The data do not have to be interoperable to the connected sensing device if they can be parsed by the relevant endpoint.

Configuration information recommended for the **`configuration`** property includes, but is not limited to:

Property	Description
calibration	Calibration factors / parameters that impact samples presented as raw.
mode	Sampling operating mode.
scaling	Scaling factors that impact samples presented as raw.
units	Configured sample unit.

17.3 identification property

The required read-only **identification** property provides static identification information about the physical sensor and/or sensor driver.

The data format of this property is implementation-dependent. For instance, the data may be a binary value or may be human-readable. The data do not have to be interoperable to the connected sensing device if they can be parsed by the relevant endpoint.

Identification information recommended for the **identification** property includes, but is not limited to:

Property	Description
model	Identification of the manufacturer and part number of the sensor. Required.
classification	Identification of the sensor classification of the sensor instance. Required for instances of defined classes.
uniqueID	Hard-coded unique identifiers associated with the sensor part. This includes serial numbers, time and date of manufacture, etc. Optional.

17.3.1 Properties of sample Object

The Provenance Sensor Class Pattern extends the sample object described in the Sensor Class Pattern to include the following properties.

Property	Description
`time`	Number originating from an absolute clock describing the instant that the sample returned was captured. If reported, **`time`** must be represented as a time value as defined in ECMA-262 in “Time Values and Time Range” (https://tc39.es/ecma262/#sec-time-values-and-time-range). The time should originate from the most accurate clock associable to the start of a sampling event, or be derived from the same.
`ticks`	Number originating from a non-absolute clock describing the instant that the sample returned was captured. If reported, **`ticks`** must be reported as an integer representing the number of time units occurring from an arbitrary, connected sensing device-consistent start time as reported by the sensor instance.
`faults`	Object representing a record of any sensor-level faults that occurred during this sensor sample or since the previously reported sample. Optional.

In the event disparate sensing modalities may be measured from a single sensor as discretely-sampled events (e.g. requesting from an IMU first acceleration and only later angular rate), those modalities are assumed to be treated as independent sensors for the purposes of recording **`time`**, **`ticks`**, and **`faults`**.

See Annex A for the formal algorithms of the Provenance Sensor Class Pattern.

Annex A
(normative)

Formal algorithms

This annex defines formal algorithms for behaviors defined by this specification. These algorithms are useful primarily for implementing the specification and validating implementations.

A.1 Internal fields

Internal fields are implementation-dependent and must not be accessible outside the implementation. For instance, they can be C structure fields, JavaScript private fields, or a combination of both.

Every object conforming to a Class Pattern is expected to have one or several internal fields. This document uses the following operators on internal fields.

A.1.1 CheckInternalFields(object)

1. For each internal field of the class being defined
    1. Let *name* be the name of the internal field
    2. Throw if *object* has no internal field named *name*

CheckInternalFields throws if an internal field is absent. That can be implicit when internal fields are JavaScript private fields, or can be explicit when internal fields are C structure fields. The purpose of **CheckInternalFields** is to ensure that *object* is an instance of the class being defined.

A.1.2 ClearInternalFields(object)

1. For each internal field of the class being defined
    1. Let *name* be the name of the internal field
    2. Clear the internal field named *name* of *object*

ClearInternalFields zeroes all internal fields. That can be storing `null` in JavaScript private fields, or can be storing NULL in C structure fields. The purpose of **ClearInternalFields** is to ensure that *object* is in a consistent state when constructed and closed.

A.1.3 GetInternalField(object, name)

1. Return the value stored in the internal field named *name* of *object*

GetInternalField is trivial for JavaScript private fields, but can involve value conversion for C structure field like converting C `NULL` into JavaScript `null`.

A.1.4 SetInternalField(object, name, value)

1. Store *value* in the internal field named *name* of *object*

SetInternalField is trivial for JavaScript private fields, but can involve value conversion for C structure field like converting JavaScript **`null`** into C **`NULL`**.

A.2 Ranges

A.2.1 Booleans

For boolean ranges, the value is converted into a JavaScript boolean.

A.2.2 Numbers

For number ranges, the value is converted into a JavaScript number, then the value is checked to be in range. The special value **`NaN`** is never in range.

For integer ranges, the value is converted into a JavaScript number, then the value is checked to be an integer, then the value is checked to be in range.

Range	From	To
number	**`-Infinity`**	**`Infinity`**
negative number	**`-Infinity`**	**`-Number.MIN_VALUE`**
positive number	**`Number.MIN_VALUE`**	**`Infinity`**
integer	**`Number.MIN_SAFE_INTEGER`**	**`Number.MAX_SAFE_INTEGER`**
negative integer	**`Number.MIN_SAFE_INTEGER`**	**`-1`**
positive integer	**`1`**	**`Number.MAX_SAFE_INTEGER`**
8-bit integer	**`-128`**	**`127`**
8-bit unsigned integer	**`0`**	**`255`**
16-bit integer	**`-32768`**	**`32767`**
16-bit unsigned integer	**`0`**	**`65535`**
32-bit integer	**`-2147483648`**	**`2147483647`**
32-bit unsigned integer	**`0`**	**`4294967295`**

Further restrictions are specified with **from x to y**, meaning the value must be **>= x** and **<= y**.

A.2.3 Objects

For object ranges like **`ArrayBuffer`**, the value is checked to be an instance of one of specified class.

Further restrictions can be specified, for instance on the **`byteLength`** of the **`ArrayBuffer`** instance.

If the object can be **`null`**, it is explicitly specified like **`Function`** or **`null`**.

A.2.4 Strings

For string ranges like **`"buffer"`**, the value is converted into a JavaScript string, then checked to be strictly equal to one of the specified values.

A.3 Base Class Pattern

A.3.1 `constructor`(*options*)

1. **ClearInternalFields**(**`this`**)
2. Throw if *options* is not an object
3. Let *params* be an empty object
4. For each supported option
    1. Let *name* be the name of the supported option
    2. If **HasProperty**(*options*, *name*)
        1. Let *value* be **GetProperty**(*options*, *name*)
        2. Throw if *value* is not in the valid range of the supported option
    3. Else
        1. Throw if the supported option has no default value
        2. Let *value* be the default value of the supported option
    4. **DefineProperty**(*params*, *name*, *value*)
5. For each supported callback option
    1. Let *name* be the name of the supported callback option
    2. Let *callback* be **GetProperty**(*params*, *name*)
    3. If *callback* is not **`null`**
        1. **SetInternalField**(**`this`**, *name*, *callback*)
6. Let *value* be **GetProperty**(*params*, **`"target"`**)
7. If *value* is not **`undefined`**
    1. **DefineProperty**(**`this`**, **`"target"`**, *value*)

8. Mark **`this`** as ineligible for garbage collection

A.3.1.1 Notes

- Supported options, with their names, default values and valid ranges, are defined by a separate table for each class conforming to the Base Class Pattern.

- The *params* object is unobservable. Its purpose in the algorithm is to ensure that properties of the *options* object are only accessed once and that the *options* object can be frozen. Local variables can be used instead, for instance:

```
let pin = 2;
if (options !== undefined) {
    if ("pin" in options)) {
        pin = options.pin;
        if ((pin < 0) || (3 < pin))
            throw new RangeError(`invalid pin ${pin}`);
    }
}
```

- Most classes conforming to the Base Class Pattern are expected to support one or several callbacks. Callbacks are supported options: their default value is **`null`**, their valid range is **`null`** or a JavaScript function. Callbacks are stored in internal fields and are always called with **`this`** set to the constructed object.

- There is only one option that is always supported: its name is **`"target"`**, its default value is **`undefined`** and its range is any JavaScript value.

A.3.2 `close`**()**

1. **CheckInternalFields**(**`this`**)
2. Mark **`this`** as eligible for garbage collection
3. Cancel any pending callbacks for **`this`**
4. **ClearInternalFields**(**`this`**)

A.4 IO Class Pattern

A.4.1 `constructor`**(*options*)**

1. Execute steps 1 to 7 of the Base Class Pattern constructor
2. Let *value* be **GetProperty**(*params*, **`"format"`**)
3. **SetInternalField**(**`this`**, **`"format"`**, *value*)
4. Try
    1. Let *resources* be the hardware resources specified by *params*
    2. Throw if *resources* are unavailable

3. Allocate and configure *resources*
4. Throw if allocation or configuration failed
5. **SetInternalField**(this, **`"resources"`**, *resources*)

5. Catch *exception*
   1. **Call**(**`this`**, **GetProperty**(**`this`**, “close”`));
   2. Throw *exception*
6. Execute step 8 of the Base Class Pattern constructor

A.4.2 `close`()

1. Execute step 1 of the Base Class Pattern **`close`** method
2. Let *resources* be **GetInternalField**(**`this`**, **`"resources"`**);
3. Return if *resources* is null
4. Execute steps 2 and 3 of the Base Class Pattern **`close`** method
5. Free *resources*
6. Execute step 4 of the Base Class Pattern **`close`** method

A.4.3 `read`([option])

1. **CheckInternalFields**(**`this`**);
2. Let *resources* be **GetInternalField**(**`this`**, **`"resources"`**);
3. Throw if *resources* is null
4. If *resources* is not readable
   1. return **`undefined`**
5. Let *format* be **GetInternalField**(**`this`**, **`"format"`**)
6. If *format* is **`"buffer"`**
   1. Let *available* be the number of readable bytes
   2. If *option* is absent
      1. Throw if *available* is **`undefined`**
      2. Let *n* be *available*
      3. Let *data* be **Construct**(**`"ArrayBuffer"`**, *n*)
      4. Let *pointer* be **GetBytePointer**(*data*)
      5. Read *n* bytes from *resources* into *pointer*
      6. Return *data*.
   3. Else if *option* is a number

1. Throw if *option* is no positive integer
2. Let *n* be *option*
3. If *available* is not **`undefined`** and *n* > *available*
    1. Let *n* be *available*
4. Let *data* be **Construct**(**`"ArrayBuffer"`**, *n*)`
5. Let *pointer* be **GetBytePointer**(*data*)
6. Read *n* bytes from *resources* into *pointer*
7. Return *data*.

4. Else
    1. Let *pointer* be **GetBytePointer**(*option*)
    2. Let *n* be **GetProperty**(*option*, **`"byteLength"`**)
    3. If *available* is not **`undefined`** and *n* > *available*
        1. Let *n* be *available*
    4. Read *n* bytes from *resources* into *pointer*
    5. Return *n*.

7. Throw if *option* is present
8. Read *data* from *resources*
9. Format *data* according to *format*
10. Return *data*.

A.4.4 `write`(*data*)

1. **CheckInternalFields**(**`this`**);
2. Let *resources* be **GetInternalField**(**`this`**, **`"resources"`**);
3. Throw if *resources* is null or not writable
4. Throw if *data* is absent
5. Let *format* be **GetInternalField**(**`this`**, **`"format"`**)
6. If *format* is **`"buffer"`**
    1. Let *pointer* be **GetBytePointer**(*data*)
    2. Let *n* be **GetProperty**(*data*, **`"byteLength"`**)
    3. Throw if *n* bytes would overflow *resources*
    4. Write *n* bytes from *pointer* into *resources*
    5. Return

7. Throw if *data* is not formatted according to *format*
8. Write *data* into *resources*

A.4.5 `set format`**(*value*)**

1. **CheckInternalFields**(**`this`**);
2. Throw if *value* is not in the valid range of **`"format"`**
3. **SetInternalField**(**`this`**, **`"format"`**, *value*)

A.4.6 `get format`**()**

1. **CheckInternalFields**(**`this`**);
2. Return **GetInternalField**(**`this`**, **`"format"`**)

A.4.6.1 Notes

- **GetBytePointer**(*buffer*) is a host specific operator that returns a pointer to the data contained in an **`ArrayBuffer`**, **`SharedArrayBuffer`** or **`TypedArray`** instance. The operator throws if *buffer* is no instance of **`ArrayBuffer`**, **`SharedArrayBuffer`** or **`TypedArray`**, or if *buffer* is detached. For a **`TypedArray`** instance, the pointer takes the view byte offset into account.
- Hardware resources can require one or several internal fields which should be all cleared and checked. The **`"resources"`** internal field is only a convention in this document.
- Several IO classes read/write bytes into/from buffers so the **`read`** and **`write`** methods detail the relevant steps, for instance to optimize the **`read`** method memory usage by passing a buffer.
- IO classes that do not use buffers can skip steps 6 of the **`read`** and **`write`** methods.
- The ranges of **`read`** and **`write`** *data* are defined by a separate table for each class conforming to the IO Class Pattern.
- When the parameters of **`read`** or **`write`** differ from the IO Class Pattern, they are defined by a separate table.

A.5 IO Classes

A.5.1 Digital

A.5.1.1 constructor options

Property	Required	Range	Default
`pin`	yes	pin specifier	
`mode`	yes	**`Digital.Input`**, **`Digital.InputPullUp`**, **`Digital.InputPullDown`**, **`Digital.InputPullUpDown`**, **`Digital.Output`**, or **`Digital.OutputOpenDrain`**.	
`edge`	no*	**`Digital.Rising`**, **`Digital.Falling`**, and **`Digital.Rising	Digital.Falling`**	
`onReadable`	no	**`null`** or **`Function`**	**`null`**
`format`	no	**`"number"`**	**`"number"`**

- If the **`onReadable`** option is not **`null`**, **`edge`** is required to have a non-zero value.

A.5.1.2 `read` / `write` data

Format	Read	Write
`"number"`	**`0`** or **`1`**	**`0`** or **`1`**

A.5.2 Digital bank

A.5.2.1 constructor options

Property	Required	Range	Default
`pins`	yes	32-bit unsigned integer	
`mode`	yes	**`Digital.Input`**, **`Digital.InputPullUp`**, **`Digital.InputPullDown`**, **`Digital.InputPullUpDown`**, **`Digital.Output`**, or **`Digital.OutputOpenDrain`**.	
`rises`	no*	32-bit unsigned integer	0
`falls`	no*	32-bit unsigned integer	0
`bank`	no	number or string	
`onReadable`	no	**`null`** or **`Function`**	**`null`**
`format`	no	**`"number"`**	**`"number"`**

- Both **`rises`** and **`falls`** cannot be **`0`**; at least one pin must be selected.

A.5.2.2 `read` / `write` data

Format	Read	Write
`"number"`	32-bit unsigned integer	32-bit unsigned integer

A.5.3 Analog input

A.5.3.1 constructor options

Property	Required	Range	Default
`pin`	yes	pin specifier	
`resolution`	no	positive integer	host-dependent
`format`	no	**`"number"`**	**`"number"`**

A.5.3.2 `read` / `write` data

Format	Read	Write
`"number"`	all	

A.5.4 Pulse-width modulation

A.5.4.1 constructor options

Property	Required	Range	Default
`pin`	yes	pin specifier	
`hz`	no	positive number	host-dependent
`format`	no	**`"number"`**	**`"number"`**

A.5.4.2 `read` / `write` data

Format	Read	Write
`"number"`		positive integer

A.5.5 I²C

A.5.5.1 constructor options

Property	Required	Range	Default
`data`	yes	pin specifier	
`clock`	yes	pin specifier	
`hz`	yes	positive integer	
`address`	yes	8-bit unsigned integer from 0 to 127	
`port`	no	port specifier	host-dependent
`onReadable`	no	**`null`** or **`Function`**	**`null`**
`format`	no	**`"buffer"`**	**`"buffer"`**

A.5.5.2 `read` / `write` data

Format	Read	Write
`"buffer"`	ArrayBuffer	ArrayBuffer, TypedArray

A.5.5.3 `read`(option[, stop])

Param	Required	Range	Default
option	yes*	positive integer, ArrayBuffer, TypedArray	
stop	no	**`true`** or **`false`**	**`true`**

- The number of readable bytes is undefined so option is required

A.5.5.4 `write`(data[, stop])

Param	Required	Range	Default
data	yes	ArrayBuffer, TypedArray	
stop	no	**`true`** or **`false`**	**`true`**

A.5.6 System management bus (SMBus)

A.5.6.1 constructor options

All properties from I²C plus the following:

Property	Required	Range	Default
`stop`	no	**`true`** or **`false`**	**`false`**

A.5.6.2 `read` / `write` data

Format	Read	Write
`"buffer"`	any	any

A.5.6.3 `read`(option)

Param	Required	Range	Default
option	yes*	positive integer, ArrayBuffer, TypedArray	

- The number of readable bytes is undefined so *option* is required

A.5.6.4 `readUint8`(register)

Param	Required	Range	Default
register	yes	integer	

A.5.6.5 `writeUint8`(register, value)

Param	Required	Range	Default
register	yes	integer	
value	yes	8-bit unsigned integer	

A.5.6.6 `readUint16`(register, bigEndian)

Param	Required	Range	Default
register	yes	integer	
bigEndian	no	**`true`** or **`false`**	**`false`**

A.5.6.7 `writeUint16`(register, value)

Param	Required	Range	Default
register	yes	integer	
value	yes	16-bit unsigned integer	

A.5.6.8 `readBuffer`(register, buffer)

Param	Required	Range	Default
register	yes	integer	
buffer	yes	**`ArrayBuffer`** or **`TypedArray`** or **`Number`**	

A.5.6.9 `writeBuffer`(register, buffer)

Param	Required	Range	Default
register	yes	integer	
buffer	yes	**`ArrayBuffer`** or **`TypedArray`**	

A.5.7 Serial

A.5.7.1 constructor options

Property	Required	Range	Default
`receive`	no*	pin specifier	
`transmit`	no*	pin specifier	
`baud`	yes	positive integer	
`flowControl`	no	**`"hardware"`** and **`"none"`**	**`"none"`**
`dataTerminalReady`	no	pin specifier	
`requestToSend`	no	pin specifier	
`clearToSend`	no	pin specifier	
`dataSetReady`	no	pin specifier	
`port`	no	port specifier	
`onReadable`	no	**`null`** or **`Function`**	**`null`**
`onWritable`	no	**`null`** or **`Function`**	**`null`**
`format`	no	**`"number"`** or **`"buffer"`**	**`"buffer"`**

- A host may require the **receive** and/or **transmit** properties.

A.5.7.2 `read` / `write` data

Format	Read	Write
`"number"`	8-bit unsigned integer	8-bit unsigned integer
`"buffer"`	ArrayBuffer	ArrayBuffer, TypedArray

A.5.7.3 `flush`**([input, output])**

1. **CheckInternalFields**(**`this`**)
2. If *input* and *output* are absent
    1. Let *flushInput* be **`true`**
    2. Let *flushOutput* be **`true`**
3. Else if *input* and *output* are present
    1. Convert *input* into a JavaScript boolean
    2. Let *flushInput* be *input*
    3. Convert *output* into a JavaScript boolean
    4. Let *flushOutput* be *output*
4. Else
    1. Throw
5. If *flushInput* is **`true`**
    1. Flush all received but unread data
6. If *flushOutput* is **`true`**
    1. Flush all written but unsent data

A.5.7.4 `set`**(options)**

1. **CheckInternalFields**(**`this`**)
2. Throw if *options* is not an object
3. If **HasProperty**(*options*, **`"dataTerminalReady"`**)
    1. Let *value* be **GetProperty**(*options*, **`"dataTerminalReady"`**)
    2. Convert *value* into a JavaScript boolean
    3. If *value* is **`true`**, set serial connection’s DTR pin
    4. Else clear serial connection’s DTR pin
4. If **HasProperty**(*options*, **`"requestToSend"`**)
    1. Let *value* be **GetProperty**(*options*, **`"requestToSend"`**)
    2. Convert *value* into a JavaScript boolean
    3. If *value* is **`true`**, set serial connection’s RTS pin
    4. Else clear serial connection’s RTS pin
5. If **HasProperty**(*options*, **`"break"`**)
    1. Let *value* be **GetProperty**(*options*, **`"break"`**)

2. Convert *value* into a JavaScript boolean
3. If *value* is **`true`**, set serial connection's break signal
4. Else clear serial connection's break signal

A.5.7.5 `get`([options])

1. **CheckInternalFields**(**`this`**)
2. If *options* is absent
    1. Let *result* be an empty object
3. Else
    1. Throw if *options* is not an object
    2. Let *result* be *options*
4. If serial connection's CTS pin is set
    1. **SetProperty**(*result*, **`"clearToSend"`**, **`true`**)
5. Else
    1. **SetProperty**(*result*, **`"clearToSend"`**, **`false`**)
6. If serial connection's DSR pin is set
    1. **SetProperty**(*result*, **`"dataSetReady"`**, **`true`**)
7. Else
    1. **SetProperty**(*result*, **`"dataSetReady"`**, **`false`**)
8. Return *result*

A.5.8 Serial Peripheral Interface (SPI)

A.5.8.1 constructor options

Property	Required	Range	Default
`out`	no*	pin specifier	
`in`	no*	pin specifier	
`clock`	yes	pin specifier	
`select`	no*	pin specifier	
`active`	no	0 or 1	0
`hz`	yes	positive integer	
`mode`	no	0, 1, 2, or 3	0
`port`	no	port specifier	
`format`	no	`"buffer"`	`"buffer"`

A.5.8.2 `read` / `write` data

Format	Read	Write
`"buffer"`	ArrayBuffer	ArrayBuffer

A.5.8.3 `read`(option)

Param	Required	Range	Default
option	yes*	positive integer, ArrayBuffer, TypedArray	

- The number of readable bytes is undefined so *option* is required

A.5.8.4 `transfer`(buffer)

1. **CheckInternalFields**(`this`)
2. If *buffer* is an ArrayBuffer
    1. Let *transferBuffer* be *buffer*
    2. Let *transferOffset* be 0
3. Else
    1. Let *transferBuffer* be **GetProperty**(*buffer*, “buffer”)

2. Let *transferOffset* be **GetProperty**(*buffer*, “byteOffset”)

4. If **HasProperty**(*buffer*, “bitLength”)
    1. Let *transferBits* be **GetProperty**(*buffer*, “bitLength”)
    2. Let *availableBits* be **GetProperty**(*buffer*, “byteLength”) * 8
    3. Throw if *transferBits* is greater than *availableBits*
5. Else
    1. Let *transferBits* be **GetProperty**(*buffer*, “byteLength”) * 8
6. Simultaneously write and read *transferBits* bits into *buffer* starting at byte offset *transferOffset*
7. Return *buffer*

A.5.8.5 `flush`([deselect])

1. **CheckInternalFields**(**`this`**)
2. Flush all written but unsent data
3. If *deselect* is present
    1. Convert *deselect* into a JavaScript boolean
    2. If *deselect* is **`true`**
        1. If **GetInternalField**(**`this`**, **`"active"`**) is 0
            1. Set the select pin to 1
        2. Else
            1. Set the select pin to 0

A.5.9 Pulse count

A.5.9.1 constructor options

Property	Required	Range	Default
`signal`	yes	pin specifier	
`control`	yes	pin specifier	
`onReadable`	no	**`null`** or **`Function`**	**`null`**
`format`	no	**`"number"`**	**`"number"`**

A.5.9.2 `read` / `write` data

Format	Read	Write
`"number"`	integer	integer

A.5.10 TCP socket

A.5.10.1 constructor options

Property	Required	Range	Default
`address`	yes*	string	
`host`	yes*	string	
`port`	yes	16-bit unsigned integer	
`noDelay`	no	**`true`** or **`false`**	**`false`**
`keepAlive`	no	positive integer	N/A
`from`	no	instance of TCP Socket	N/A
`onError`	no	**`null`** or **`Function`**	**`null`**
`onWritable`	no	**`null`** or **`Function`**	**`null`**
`onReadable`	no	**`null`** or **`Function`**	**`null`**
`format`	no	**`"number"`** or **`"buffer"`**	**`"buffer"`**

- Either the **`address`** or **`host`** must be present, but not both.

A.5.10.2 `read` / `write` data

Format	Read	Write
`"buffer"`	ArrayBuffer	ArrayBuffer, TypedArray
`"number"`	8-bit unsigned integer	8-bit unsigned integer

A.5.11 TCP listener socket

Property	Required	Range	Default
`port`	yes	16-bit unsigned integer	
`address`	no	string	N/A
`onError`	no	**`null`** or **`Function`**	**`null`**
`onReadable`	no	**`null`** or **`Function`**	**`null`**
`format`	no	**`"socket/tcp"`**	**`"socket/tcp"`**

A.5.11.1 `read` / `write` data

Format	Read	Write
`"socket/tcp"`	instance of TCP Socket	

A.5.12 UDP socket

A.5.12.1 constructor options

Property	Required	Range	Default
`address`	no	string	N/A
`port`	no	16-bit signed integer	N/A
`multicast`	no	string	N/A
`timeToLive`	yes, if multicast used	integer from 1 to 255	N/A
`onError`	no	**`null`** or **`Function`**	**`null`**
`onWritable`	no	**`null`** or **`Function`**	**`null`**
`format`	no	**`"buffer"`**	**`"buffer"`**

- Either the **`address`** or **`host`** must be present, but not both.

A.5.12.2 `read` / `write` data

Format	Read	Write
`"buffer"`	ArrayBuffer	ArrayBuffer, TypedArray

A.5.12.3 `write`(data, address, port)

Param	Required	Range	Default
data	yes	ArrayBuffer, TypedArray	
address	yes	string	
port	yes	16-bit unsigned integer	

A.6 Peripheral Class Pattern

A.6.1 `constructor`(options)

1. Execute steps 1 to 7 of the Base Class Pattern constructor
2. Try
    1. For each supported IO connection
        1. Let *name* be the name of the supported IO connection.
        2. Let *ioOptions* be **GetProperty**(*params*, *name*)
        3. Let *ioConstructor* be **GetProperty**(*ioOptions*, **`"io"`**)
        4. Let *ioConnection* be **Construct**(*ioConstructor*, *ioOptions*)
        5. **SetInternalField**(**`this`**, *name*, *ioConnection*);
    2. Configure the peripheral with *params*
    3. Throw if the communication with the peripheral is not operational
    4. Activate the peripheral
    5. **SetInternalField**(**`this`**, **`"status"`**, **`"ready"`**);
3. Catch *exception*
    1. **Call**(**`this`**, **GetProperty**(**`this`**, “close”`));
    2. Throw *exception*
4. Execute step 8 of the Base Class Pattern constructor

A.6.2 `close`()

1. Execute step 1 of the Base Class Pattern **`close`** method

2. Let *status* be **GetInternalField**(**`this`**, **`"status"`**);
3. Return if *status* is null
4. Execute steps 2 and 3 of the Base Class Pattern **`close`** method
5. Deactivate the peripheral
6. For each supported IO connection
    1. Let *name* be the name of the supported IO connection.
    2. Let *ioConnection* be **GetInternalField**(**`this`**, *name*);
    3. If *ioConnection* is not **`null`**
        1. **Call**(*ioConnection*, **`"close"`**);
7. Execute step 4 of the Base Class Pattern **`close`** method

A.6.3 `configure`**(options)**

1. **CheckInternalFields**(**`this`**);
2. Let *status* be **GetInternalField**(**`this`**, **`"status"`**);
3. Throw if *status* is null
4. Throw if *options* is undefined or null
5. For each supported option
    1. Let *name* be the name of the supported option
    2. If **HasProperty**(*options*, *name*)
        1. Let *value* be **GetProperty**(*options*, *name*)
        2. Throw if *value* is not in the valid range of the supported option
6. Configure the peripheral with *options*

A.6.3.1 Notes

- Supported IO connections are supported options. Their value must be an object with an **`io`** property, which is the class of the IO connection.

A.7 Sensor Class Pattern

A.7.1 `constructor`**(*options*)**

1. Execute all steps of the Peripheral Class Pattern constructor

A.7.2 `close`**()**

1. Execute all steps of the Peripheral Class Pattern **`close`** method

A.7.3 `configure`(*options*)

1. Execute all steps of the Peripheral Class Pattern **`configure`** method

A.7.4 `sample`([*params*])

1. **CheckInternalFields**(**`this`**);
2. Let *status* be **GetInternalField**(**`this`**, **`"status"`**);
3. Throw if *status* is null
4. Throw if *params* are absent but required, or present but not in the valid range
5. If the peripheral is readable
    1. Let *result* be an empty object
    2. For each sample property
        1. Let *name* be the name of the sample property
        2. Let *value* be undefined
        3. Read from the peripheral into *value*
        4. **DefineProperty**(*result*, *name*, *value*);
6. Else
    1. Let *result* be undefined
7. Return *result*.

A.7.4.1 Notes

- The order, requirements and ranges of **`sample`** *params* are defined by a separate table for each class conforming to the Sensor Class Pattern.
- The requirements and ranges of properties in **`sample`** *result* are defined by a separate table for each class conforming to the Sensor Class Pattern.

A.8 Sensor Classes

A.8.1 Accelerometer

A.8.1.1 `sample` params:

None

A.8.1.2 `sample` result:

Property	Required	Range	Description
x	yes	number	acceleration along the x axis in meters per second squared
y	yes	number	acceleration along the y axis in meters per second squared
z	yes	number	acceleration along the z axis in meters per second squared

A.8.2 Ambient light

A.8.2.1 `sample` params:

None

A.8.2.2 `sample` result:

Property	Required	Range	Description
`illuminance`	yes	positive number	ambient light level in lux

A.8.3 Atmospheric pressure

A.8.3.1 `sample` params:

None

A.8.3.2 `sample` result:

Property	Required	Range	Description
`pressure`	yes	number	atmospheric pressure in Pascal

A.8.4 Humidity

A.8.4.1 `sample` params:

None

A.8.4.2 `sample` result:

Property	Required	Range	Description
`humidity`	yes	number from 0 to 1	relative humidity as a percentage

A.8.5 Proximity

A.8.5.1 `sample` params:

None

A.8.5.2 `sample` result:

Property	Required	Range	Description
`near`	yes	boolean	indicator of a detected proximate object
`distance`	yes	positive number or **`null`**	distance to the nearest sensed object in centimeters or **`null`** if no object is detected
`max`	yes	positive number	maximum sensing range of the sensor in centimeters

A.8.6 Temperature

A.8.6.1 `sample` params:

None

A.8.6.2 `sample` result:

Property	Required	Range	Description
`temperature`	yes	number	temperature in degrees Celsius

A.8.7 Touch

A.8.7.1 `sample` params:

None

A.8.7.2 `sample` result:

`Array` of **`touch`** objects or **`undefined`** if no touch is in progress.

A.8.7.3 `touch` object:

Property	Required	Range	Description
`x`	yes	number	X coordinate of the touch point
`y`	yes	number	Y coordinate of the touch point
`id`	yes	positive integer	indicator of which touch point this entry corresponds to

A.9 Display Class Pattern

A.9.1 `constructor`(options)

1. Execute all steps of the Peripheral Class Pattern constructor

A.9.2 `adaptInvalid`(area)

1. **CheckInternalFields**(**`this`**)
2. Throw if *area* is absent
3. If **HasProperty**(*area*, **`"x"`**)
    1. Let *x* be **GetProperty**(*area*, **`"x"`**)
4. Else
    1. Let *x* be **`0`**
5. If **HasProperty**(*area*, **`"y"`**)
    1. Let *y* be **GetProperty**(*area*, **`"y"`**)
6. Else
    1. Let *y* be **`0`**
7. If **HasProperty**(*area*, **`"width"`**)
    1. Let *width* be **GetProperty**(*area*, **`"width"`**)
8. Else
    1. Let *width* be the width of the frame buffer in pixels
9. If **HasProperty**(*area*, **`"height"`**)
    1. Let *height* be **GetProperty**(*area*, **`"height"`**)

10. Else
    1. Let *height* be the height of the frame buffer in pixels
11. Adjust *x*, *y*, *width*, *height* to define a valid area to update
12. **SetProperty**(*area*, **`"x"`**, *x*)
13. **SetProperty**(*area*, **`"y"`**, *y*)
14. **SetProperty**(*area*, **`"width"`**, *width*)
15. **SetProperty**(*area*, **`"height"`**, *height*)

A.9.3 `close`()

1. Execute all steps of the Peripheral Class Pattern **`close`** method

A.9.4 `begin`(*options*)

1. **CheckInternalFields**(**`this`**)
2. Let *status* be **GetInternalField**(**`this`**, **`"status"`**)
3. Throw if *status* is null
4. Let *x* be **`0`**
5. Let *y* be **`0`**
6. Let *width* be the width of the frame buffer in pixels
7. Let *height* be the height of the frame buffer in pixels
8. Let *continue* be **`false`**
9. If *options* is present
    1. If **HasProperty**(*options*, **`"x"`**)
        1. Let *x* be **GetProperty**(*options*, **`"x"`**)
    2. If **HasProperty**(*options*, **`"y"`**)
        1. Let *y* be **GetProperty**(*options*, **`"y"`**)
    3. If **HasProperty**(*options*, **`"width"`**)
        1. Let *width* be **GetProperty**(*options*, **`"width"`**)
    4. If **HasProperty**(*options*, **`"height"`**)
        1. Let *height* be **GetProperty**(*options*, **`"height"`**)
    5. If **HasProperty**(*options*, **`"continue"`**)
        1. Let *continue* be **GetProperty**(*options*, **`"continue"`**)
10. Throw if the area defined by *x*, *y*, *width* and *height* is invalid.

11. If *status* is **`ready`**
    1. **SetInternalField**(**`this`**, **`"status"`**, **`"updating"`**)
12. Else
    1. Throw if *continue* is false
13. Use *x*, *y*, *width*, *height* to prepare the frame buffer to receive scanlines

A.9.5 `configure`(options)

1. Execute all steps of the Peripheral Class Pattern **`configure`** method

A.9.6 `end`()

1. **CheckInternalFields**(**`this`**)
2. Let *status* be **GetInternalField**(**`this`**, **`"status"`**)
3. Throw if *status* is not **`"updating"`**
4. **SetInternalField**(**`this`**, **`"status"`**, **`"finishing"`**)
5. Make updated frame buffer visible
6. **SetInternalField**(**`this`**, **`"status"`**, **`"ready"`**)

A.9.7 `send`(scanlines)

1. **CheckInternalFields**(**`this`**)
2. Let *status* be **GetInternalField**(**`this`**, **`"status"`**)
3. Throw if *status* is not **`"updating"`**
4. Throw if *scanlines* is absent
5. Let *pointer* be **GetBytePointer**(*scanlines*)
6. Let *n* be **GetProperty**(*lines*, **`"byteLength"`**)
7. Transfer *n* bytes from *pointer* to the frame buffer

A.9.8 `get width`()

1. **CheckInternalFields**(**`this`**)
2. Return the width of the frame buffer in pixels

A.9.9 `get height`()

1. **CheckInternalFields**(**`this`**)
2. Return the height of the frame buffer in pixels

A.9.9.1 Notes

- **GetBytePointer**(*buffer*) is a host specific operator that returns a pointer to the data contained in an **`ArrayBuffer`**, **`SharedArrayBuffer`** or **`TypedArray`** instance. The operator throws if *buffer* is no instance of **`ArrayBuffer`**, **`SharedArrayBuffer`** or **`TypedArray`**, or if *buffer* is detached. For a **`TypedArray`** instance, the pointer takes the view byte offset into account.
- When the frame buffer **`rotation`** is 90 or 270 degrees, **`get width`** returns the height of the frame buffer in pixels and **`get height`** returns the width of the frame buffer in pixels.

A.9.9.2 constructor options:

Property	Required	Range	Default
format	no	see text	
rotation	no	0, 90, 180, or 270	
brightness	no	0.0 to 1.0	
flip	no	""“,”h“,”v“, or”hv"	

A.10 Provenance Sensor Class Pattern

A.10.1 `configure`(*options*)

1. Execute all steps of the Sensor Class Pattern configure method

A.10.2 `sample`([*params*])

1. Execute steps 1 to 6 of the Sensor Class Pattern **`sample`** method
2. If *result* is an object
    1. If an absolute clock is available
        1. Let *time* be the value of the absolute clock upon sampling
        2. **DefineProperty**(*result*, **`"time"`**, *time*);
    2. If a relative clock is available
        1. Let *ticks* be the value of a relative clock upon sampling
        2. **DefineProperty**(*result*, **`"ticks"`**, *ticks*);
    3. If faults are readable from the sensor upon sampling
        1. Read from the sensor into *faults*
        2. **DefineProperty**(*result*, **`"faults"`**, *faults*);
3. Execute steps 7 of the Sensor Class Pattern **`sample`** method

A.10.2.1 Notes

- The absolute clock is the most precise clock available to get an absolute time value (since the Epoch), from either the sensor, the microcontroller, or another peripheral.
- The relative clock is any clock available to get a consistent relative time value (for instance since the device started), from either the sensor, the microcontroller, or another peripheral.

A.10.2.2 `sample` params:

None

A.10.2.3 `sample` result:

In addition to the sample results defined in the Sensor Class Pattern, the Provenance Sensor Class Pattern adds properties as follows:

Property	Required	Range	Description
`time`	yes, if available	positive number	number originating from an absolute clock describing the instant that the sample returned was captured
`ticks`	yes, if available	positive number	number originating from a non-absolute clock describing the instant that the sample returned was captured
`faults`	no	boolean, number, or string	object representing a record of any sensor-level faults that occurred during this sensor sample or since the previously reported sample

A.10.2.4 Notes

- The order, requirements, and ranges of options for **configure** extend those found in a separate table for every class conforming to the Sensor Class Pattern, and add the options **`configuration`** and **`identification`** as defined in the Sensor Provenance Class Pattern.
- Metadata (*time*, *ticks*, *faults*) reflect only the metadata associated with the first sample. In cases where multiple samples may be taken from a single device, timing and fault data may be imprecise for subsequent samples.

A.11 IO Provider Class Pattern

A.11.1 `constructor`(***options***)

1. Execute steps 1 to 7 of the Base Class Pattern constructor

2. Let *onReadable* be a function with the following steps:
    1. Let *data* be **Call**(**`this`**, **GetProperty**(**`this`**, “read”`));
    2. Let *provider* be GetProperty(**`this`**, **`"target"`**);
    3. Dispatch *data* among IO objects of *provider*
3. Let *count* be the number of supported IO connection
4. Let *onWritable* be a function with the following steps:
    1. Let *count* be *count* - 1
    2. If *count* is 0
        1. Let *provider* be GetProperty(**`this`**, **`"target"`**);
        2. Configure *provider* with *params*
        3. Add supported IO constructors to *provider*
        4. **SetInternalField**(*provider*, **`"status"`**, **`"ready"`**);
        5. Let *callback* be **GetInternalField**(*provider*, **`"onReady"`**);
        6. If *callback* is not null
            1. **Call**(*provider*, *callback*);
5. Let *onError* be a function with the following steps:
    1. Let *provider* be GetProperty(**`this`**, **`"target"`**);
    2. Dispatch the error to open IO objects of *provider*
    3. **Call**(*provider* , **GetProperty**(*provider*, “close”`));
    4. Let *callback* be **GetInternalField**(*provider* , **`"onError"`**);
    5. If *callback* is not null
        1. **Call**(*provider*, *callback*);
6. Try
    1. For each supported IO connection
        1. Let *name* be the name of the supported IO connection.
        2. Let *ioOptions* be **GetProperty**(*params*, *name*)
        3. Let *ioParams* be a copy of *ioOptions*
        4. Let *ioConstructor* be **GetProperty**(*ioParams*, **`"io"`**)
        5. **DefineProperty**(*ioParams*, **`"onReadable"`**, *onReadable*);
        6. **DefineProperty**(*ioParams*, **`"onWritable"`**, *onWritable*);
        7. **DefineProperty**(*ioParams*, **`"onError"`**, *onError*);

8. **DefineProperty**(*ioParams*, **"target"**, **this**);
9. Let *ioConnection* be **Construct**(*ioConstructor*, *ioParams*)
10. **SetInternalField**(**this**, *name*, *ioConnection*);

7. Catch *exception*
   1. **Call**(**this**, **GetProperty**(**this**, “close”`));
   2. Throw *exception*
8. Execute step 8 of the Base Class Pattern constructor

A.11.2 close()

1. Execute all steps of the Peripheral Class Pattern **close** method

Bibliography

[1] I²C-bus specification and user manual, Rev. 6. https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[2] System Management Bus (SMBus) Specification Version 3.1. http://smbus.org/specs/SMBus_3_1_20180319.pdf

[3] W3C Generic Sensor specification. https://www.w3.org/TR/generic-sensor/

[4] W3C Accelerometer draft. https://w3c.github.io/accelerometer/

[5] W3C Ambient Light Sensor draft. https://www.w3.org/TR/ambient-light/

[6] W3C Proximity Sensor draft. https://w3c.github.io/proximity/

[7] Ecma TC39 - Compartments Proposal. https://github.com/tc39/proposal-compartments

[8] Ecma TC39 - SES Proposal. https://github.com/tc39/proposal-ses

[9] Draft Specification for Standalone SES. https://github.com/Agoric/SES-shim/blob/master/packages/ses/docs/source/draft-standalone-spec.md