

ECMAScript® embedded systems API specification

Final draft

ECMA-419

2nd Edition / April 2023

Ecma/TC53/2023/010  
Ecma/GA/2023/028

Contents Page

1 Scope 1

2 Conformance 1

3 Normative references 1

4 Terms and definitions 2

5 Notational conventions 4

6 Overview 4

6.1 ECMAScript 4

6.2 Class patterns 4

6.3 Independent implementations 4

6.4 Self-hosting 4

6.5 Module specifiers 5

6.6 Secure ECMAScript 5

6.7 Naming 5

6.8 IP address 6

6.9 MAC address 6

6.10 Byte Buffer 6

7 Requirements for standard built-in ECMAScript objects 6

8 Base Class Pattern 6

8.1 Asynchronous methods 6

8.2 constructor 7

8.3 close method 7

8.4 target property 8

8.5 Callbacks 8

9 IO Class Pattern 8

9.1 Pin specifier 9

9.2 Port specifier 9

9.3 constructor 9

9.4 read method 9

9.5 write method 10

9.6 format property 10

9.7 Callbacks 10

9.7.1 onReadable 11

9.7.2 onWritable 11

9.7.3 onError 11

10 IO classes 11

10.1 Digital 11

10.1.1 Properties of constructor options object 12

10.1.2 Callbacks 12

10.1.3 Data format 12

10.1.4 Notes 12

10.2 Digital bank 12

10.2.1 Properties of constructor options object 13

10.2.2 Callbacks 13

10.2.3 Data format 13

10.2.4 Notes 13

10.3 Analog input 13

10.3.1 Properties of constructor options object 14

10.3.2 Data format 14

10.3.3 resolution property 14

10.4 Pulse-width modulation 14

10.4.1 Properties of constructor options object 14

10.4.2 Data format 14

10.4.3 resolution property 14

10.4.4 hz property 14

10.4.5 Notes 15

10.5 I²C – synchronous IO 15

10.5.1 Properties of constructor options object 15

10.5.2 Data format 15

10.5.3 Specifying stop bit with read and write methods 15

10.5.4 Methods 15

10.6 I²C – asynchronous IO 16

10.6.1 Properties of constructor options object 16

10.6.2 Data format 16

10.6.3 Specifying stop bit with read and write methods 16

10.6.4 Methods 16

10.7 System management bus (SMBus) – synchronous IO 17

10.7.1 Properties of constructor options object 17

10.7.2 Methods 17

10.8 System management bus (SMBus) – asynchronous IO 18

10.8.1 Properties of constructor options object 18

10.8.2 Methods 18

10.9 Serial 18

10.9.1 Properties of constructor options object 19

10.9.2 Methods 19

10.9.3 Callbacks 20

10.9.4 Data format 20

10.10 Serial Peripheral Interface (SPI) 20

10.10.1 Properties of constructor options object 21

10.10.2 Data format 21

10.10.3 Methods 21

10.11 Pulse count 22

10.11.1 Properties of constructor options object 22

10.11.2 Data format 22

10.11.3 Methods 22

10.11.4 Callbacks 23

10.12 TCP socket 23

10.12.1 Properties of constructor options object 23

10.12.2 Methods 23

10.12.3 Callbacks 24

10.12.4 Data format 24

10.12.5 remoteAddress property 24

10.12.6 remotePort property 24

10.13 TCP listener socket 24

10.13.1 Properties of constructor options object 25

10.13.2 Methods 25

10.13.3 Callbacks 25

10.13.4 Data format 25

10.14 UDP socket 25

10.14.1 Properties of constructor options object 26

10.14.2 Methods 26

10.14.3 Callbacks 26

10.14.4 Data format 26

10.15 TLS Client socket 26

10.15.1 Properties of constructor options object 27

10.15.2 write(buffer) 27

11 IO Provider Class Pattern 28

11.1 constructor 28

11.2 close method 28

11.3 Callbacks 29

12 Peripheral Class Pattern 29

12.1 constructor 29

12.2 close method 30

12.3 configure method 30

12.4 Accessors for configuration 30

13 Sensor Class Pattern 31

13.1 constructor 31

13.2 configure method 32

13.3 sample method 32

13.4 Callbacks 32

14 Sensor classes 32

14.1 Compound sensors 32

14.2 Accelerometer 33

14.2.1 Properties of a sample object 33

14.3 Ambient light 33

14.3.1 Properties of sample object 33

14.4 Atmospheric pressure 34

14.4.1 Properties of a sample object 34

14.5 Carbon Dioxide 34

14.5.1 Properties of a sample object 34

14.6 Carbon Monoxide 34

14.6.1 Properties of a sample object 34

14.7 Dust 34

14.7.1 Properties of a sample object 35

14.8 Gyroscope 35

14.8.1 Properties of a sample object 35

14.9 Humidity 35

14.9.1 Properties of a sample object 35

14.10 Hydrogen 35

14.10.1 Properties of a sample object 36

14.11 Hydrogen Sulfide 36

14.11.1 Properties of a sample object 36

14.12 Magnetometer 36

14.12.1 Properties of a sample object 36

14.13 Methane 36

14.13.1 Properties of a sample object 37

14.14 Nitric Oxide 37

14.14.1 Properties of a sample object 37

14.15 Nitric Dioxide 37

14.15.1 Properties of a sample object 37

14.16 Oxygen 37

14.16.1 Properties of a sample object 37

14.17 Particulate Matter 38

14.17.1 Properties of a sample object 38

14.18 Proximity 38

14.18.1 Properties of a sample object 38

14.19 Soil Moisture 38

14.19.1 Properties of a sample object 38

14.20 Switch 39

14.20.1 Properties of sample object 39

14.21 Temperature 39

14.21.1 Properties of a sample object 39

14.22 Touch 39

14.22.1 Sample object 39

14.23 Volatile Organic Compounds 39

14.23.1 Properties of a sample object 40

15 Display Class Pattern 40

15.1 constructor 40

15.2 configure method 40

15.3 begin method 41

15.4 send method 41

15.5 end method 41

15.6 adaptInvalid method 41

15.7 Instance properties 42

15.8 Pixel format values 43

16 Real-Time Clock Class Pattern 43

16.1 Properties of constructor options object 43

16.2 configure method 43

16.3 time property 43

16.4 configuration property 44

17 Network Interface Class Pattern 44

17.1 Properties of constructor options object 44

17.2 connect method 44

17.3 disconnect method 44

17.4 connection property 45

17.5 MAC property 45

17.6 address property 45

17.7 Ethernet Network Interface 45

17.7.1 connection property 45

17.8 Wi-Fi Network Interface 45

17.8.1 connect method 46

17.8.2 scan method 46

17.8.3 SSID property 47

17.8.4 BSSID property 47

17.8.5 RSSI property 47

17.8.6 channel property 47

18 Domain Name Resolver Class Pattern 47

18.1 resolve method 47

18.2 Properties of resolve options object 48

18.3 DNS over UDP 48

18.3.1 Properties of constructor options object 48

18.4 DNS over HTTPS (DoH) 48

18.4.1 Properties of constructor options object 48

19 NTP Client 48

19.1 Properties of constructor options object 49

19.2 getTime method 49

20 HTTP Client class pattern 49

20.1 Data format 49

20.2 Properties of constructor options object 49

20.3 close method 49

20.4 request method 49

20.5 HTTP Client Request instance 50

20.5.1 read method 50

20.5.2 write method 50

21 HTTP Server class pattern 51

21.1 Data format 51

21.2 Properties of constructor options object 51

21.3 close method 51

21.4 HTTP Server Connection instance 51

21.4.1 close method 51

21.4.2 detach method 51

21.4.3 accept method 51

21.4.4 respond method 52

21.4.5 read method 53

21.4.6 write method 53

21.4.7 route property 53

22 HTTP Server Connection routes 53

22.1 Static Data route 53

22.1.1 Properties of route 53

22.2 WebSocket Handshake route 54

22.2.1 Properties of route 54

23 WebSocket Client class pattern 54

23.1 Data format 54

23.2 Properties of constructor options object 55

23.3 close method 55

23.4 read method 56

23.5 write method 56

23.6 Static properties of the constructor 56

24 MQTT Client class pattern 56

24.1 Data format 57

24.2 Properties of constructor options object 57

24.3 close method 58

24.4 read method 59

24.5 write method 59

24.6 Static properties of the constructor 60

25 Host provider instance 60

25.1 Global variable 60

25.2 Pin name property 60

25.3 IO bus properties 61

25.4 IO classes 62

25.5 IO Providers 62

25.6 Sensors 62

25.7 Displays 62

25.8 Real-time clocks 62

25.9 Domain Name resolver 62

25.10 NTP client 63

25.11 HTTP client 63

25.12 HTTPS client 63

25.13 HTTP server 63

25.14 MQTT client 63

25.15 MQTTS client 63

25.16 WS (WebSocket) client 63

25.17 WSS (WebSocket Secure) client 63

25.18 TLS client 63

25.19 Network Interfaces 63

26 Provenance Sensor Class Pattern 64

26.1.1 Properties of constructor options object 64

26.2 configuration property 64

26.3 identification property 65

26.3.1 Properties of sample Object 65

Annex A (normative) Formal algorithms 67

A.1 Internal fields 67

A.1.1 CheckInternalFields(*object*) 67

A.1.2 ClearInternalFields(*object*) 67

A.1.3 GetInternalField(*object*, *name*) 67

A.1.4 SetInternalField(*object*, *name*, *value*) 67

A.1.5 Internal methods 68

A.2 Ranges 68

A.2.1 Booleans 68

A.2.2 Numbers 68

A.2.3 Objects 69

A.2.4 Byte buffers 69

A.2.5 Strings 69

A.2.6 Asynchronous operations 69

A.3 Base Class Pattern 69

A.3.1 constructor(*options*) 69

A.3.2 close() 70

A.3.3 close(callback) 71

A.4 IO Class Pattern 71

A.4.1 constructor(options) 71

A.4.2 close() 71

A.4.3 read([*option*]) 72

A.4.4 write(*data*) 73

A.4.5 set format(*value*) 73

A.4.6 get format() 73

A.4.7 Notes 73

A.5 IO Class Pattern – asynchronous 73

A.5.1 close(callback) 73

A.5.2 read(option[, callback]) 74

A.5.3 write(data[, callback]) 74

A.5.4 Notes 75

A.6 IO Classes 76

A.6.1 Digital 76

A.6.2 Digital bank 76

A.6.3 Analog input 77

A.6.4 Pulse-width modulation 77

A.6.5 I²C – synchronous IO 78

A.6.6 I²C – asynchronous IO 78

A.6.7 System management bus (SMBus) – synchronous IO 79

A.6.9 Serial 82

A.6.10 Serial Peripheral Interface (SPI) 85

A.6.11 Pulse count 86

A.6.12 TCP socket 87

A.6.13 TCP listener socket 87

A.6.14 constructor *options* 88

A.7 Peripheral Class Pattern 88

A.7.1 constructor(*options*) 88

A.7.2 close() 89

A.7.3 configure(*options*) 89

A.7.4 Notes 89

A.8 Sensor Class Pattern 90

A.8.1 constructor(options) 90

A.8.2 close() 90

A.8.3 configure(*options*) 90

A.8.4 sample([*params*]) 90

A.8.5 Notes 90

A.9 Sensor Classes 91

A.9.1 Accelerometer 91

A.9.2 Ambient light 91

A.9.3 Atmospheric pressure 91

A.9.4 Carbon Dioxide 92

A.9.5 Carbon Monoxide 92

A.9.6 Dust 92

A.9.7 Gyroscope 92

A.9.8 Humidity 93

A.9.9 Hydrogen 93

A.9.10 Hydrogen Sulfide 93

A.9.11 Magnetometer 93

A.9.12 Methane 94

A.9.13 Nitric Oxide 94

A.9.14 Nitric Dioxide 94

A.9.15 Oxygen 95

A.9.16 Particulate Matter 95

A.9.17 Proximity 95

A.9.18 Soil Moisture 96

A.9.19 Temperature 96

A.9.20 Touch 96

A.10 Display Class Pattern 97

A.10.1 constructor(*options*) 97

A.10.2 adaptInvalid(*area*) 97

A.10.3 close() 98

A.10.4 begin(*options*) 98

A.10.5 configure(*options*) 98

A.10.6 end() 98

A.10.7 send(scanlines) 99

A.10.8 get width() 99

A.10.9 get height() 99

A.10.10 Notes 99

A.10.11 constructor *options*: 99

A.11 Real-Time Clock Class Pattern 99

A.11.1 constructor(*options*) 99

A.11.2 close() 100

A.11.3 configure(*options*) 100

A.11.4 get time() 100

A.11.5 set time(*time*) 100

A.11.6 constructor *options* 100

A.11.7 configure *options* 101

A.12 Network Interface Class Pattern 101

A.12.1 constructor(*options*) 101

A.12.2 close() 101

A.12.3 connect(*options*) 101

A.12.4 disconnect() 101

A.12.5 get MAC() 101

A.12.6 get address() 102

A.12.7 get connection() 102

A.12.8 constructor *options* 102

A.13 Ethernet Network Interface 102

A.13.1 connect(*options*) 102

A.14 Wi-Fi Network Interface 102

A.14.1 connect(*options*) 102

A.14.2 scan(options) 103

A.14.3 get BSSID() 105

A.14.4 get RSSI() 105

A.14.5 get SSID() 105

A.14.6 get channel() 105

A.15 Domain Name Resolver Class Pattern 106

A.15.1 constructor(*options*) 106

A.15.2 close() 106

A.15.3 resolve(options[, callback]) 106

A.16 DNS over UDP 106

A.16.1 constructor *options* 106

A.16.2 Notes 107

A.17 DNS over HTTPS 107

A.17.1 constructor *options* 107

A.17.2 Notes 107

A.18 NTP Client 107

A.18.1 constructor(*options*) 107

A.18.2 close() 107

A.18.3 getTime(*callback*) 107

A.18.4 constructor *options* 108

A.18.5 Notes 108

A.19 TCP Client Class Pattern 108

A.19.1 constructor(*options*) 108

A.19.2 close() 108

A.19.3 #resolveCallback(*error*, *name*, *address*) 109

A.19.4 read(count) 109

A.19.5 write(*data*[,options]) 109

A.19.6 #tcpError(*error*) 109

A.19.7 #tcpReadable(*count*) 109

A.19.8 #tcpWritable(*count*) 109

A.19.9 read / write *data* 109

A.19.10 Notes 110

A.20 HTTP Client 110

A.20.1 constructor(*options*) 110

A.20.2 close() 110

A.20.3 request(*options*) 110

A.20.4 constructor *options* 110

A.20.5 Notes 111

A.21 HTTP Client Request 111

A.21.1 constructor *options* 111

A.21.2 read / write *data* 111

A.22 MQTT Client 111

A.22.1 constructor *options* 111

A.22.2 write options 112

A.22.3 Notes 113

A.23 WebSocket Client 113

A.23.1 constructor(*options*) 113

A.23.2 constructor *options* 113

A.23.3 write options 113

A.23.4 Notes 114

A.24 TCP Server Class Pattern 114

A.24.1 constructor(*options*) 114

A.24.2 close() 114

A.24.3 #tcpReadable(*count*) 114

A.24.4 constructor *options* 115

A.24.5 Notes 115

A.25 TCP Server Connection Class Pattern 115

A.25.1 constructor(*server*, *from*) 115

A.25.2 close() 115

A.25.3 read(*count*) 116

A.25.4 write(*data*[, options]) 116

A.25.5 #tcpError(*error*) 116

A.25.6 #tcpReadable(*count*) 116

A.25.7 #tcpWritable(*count*) 116

A.25.8 read / write *data* 116

A.25.9 Notes 116

A.26 HTTP Server 116

A.26.1 Notes 116

A.27 HTTP Server Connection 116

A.27.1 detach() 116

A.27.2 accept(*options*) 117

A.27.3 get route() 117

A.27.4 set route(*options*) 117

A.27.5 respond(*options*) 117

A.27.6 Notes 118

A.28 Provenance Sensor Class Pattern 118

A.28.1 configure(*options*) 118

A.28.2 sample([*params*]) 118

A.28.3 Notes 119

A.29 IO Provider Class Pattern 119

A.29.1 constructor(*options*) 119

A.29.2 close() 120

Bibliography 121

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](https://github.com/extensibleweb/manifesto#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.

The first edition of this Ecma Standard was adopted by the Ecma General Assembly of June 2021. It was built around the IO Class Pattern which provides consistent, efficient, extensible access to the IO capabilities of embedded systems. Driver-style classes for IO extenders, sensors, and displays build on the IO foundation.

The second edition extends IO with asynchronous capabilities used by I²C and the system management bus. It introduces new sensor classes, including many gas sensors; classes to manage and monitor network interfaces; client support for common network protocols including HTTP, MQTT, NTP, DNS, WebSocket, and TLS; server support for the HTTP and WebSocket protocols; and a real-time clock peripheral class.

This Ecma Standard was developed by Technical Committee 53 and was adopted by the General Assembly of <month> <year>.

*COPYRIGHT NOTICE*

*© 2023 Ecma International*

*This document may be copied, published and distributed to others, and certain derivative works of it may be prepared, copied, published, and distributed, in whole or in part, provided that the above copyright notice and this Copyright License and Disclaimer are included on all such copies and derivative works. The only derivative works that are permissible under this Copyright License and Disclaimer are:*

*(i) works which incorporate all or portion of this document for the purpose of providing commentary or explanation (such as an annotated version of the document),*

*(ii) works which incorporate all or portion of this document for the purpose of incorporating features that provide accessibility,*

*(iii) translations of this document into languages other than English and into different formats and*

*(iv) works by making use of this specification in standard conformant products by implementing (e.g. by copy and paste wholly or partly) the functionality therein.*

*However, the content of this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as required to translate it into languages other than English or into a different format.*

*The official version of an Ecma International document is the English language version on the Ecma International website. In the event of discrepancies between a translated version and the official version, the official version shall govern.*

*The limited permissions granted above are perpetual and will not be revoked by Ecma International or its successors or assigns.*

*This document and the information contained herein is provided on an "AS IS" basis and ECMA INTERNATIONAL DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.*

ECMAScript® embedded systems API specificationINTERNATIONAL STANDARD© ISO/IEC 2008 – All rights reservedISO/IEC 10995:2008(E) 63Test method for the estimation of the archival lifetime of optical mediaInformation technology — Digitally recorded media for information interchange and storageTechnologies de l'information — Supports enregistrés numériquement pour l'échange et le stockage d'information — Méthode d'essai pour l'estimation de la durée de vie d'archivage des supports optiquesInformation technology — Digitally recorded media for information interchange and storage — Test method for the estimation of the archival lifetime of optical mediaE2008-02-22(60) PublicationISO/IECISO/IEC    International Standard 2008ISO/IEC 10995ISO/IEC 10995ISO/IEC 10995   Information technology  JTC1 2Heading 2Heading 1     0 CR6STD Version 2.260   4O:\Documents\JTC001\SC023\046554 - ISO\_IEC CD 10995 (Ed 1)\60.00\310\046554e\C046554e.doc

# 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.

# 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).

# Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, 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>

RFC 7230 - 7240, *Hypertext Transfer Protocol (HTTP/1.1)*   
<https://tools.ietf.org/html/rfc7230>

RFC 6455, *The WebSocket Protocol*  
<https://tools.ietf.org/html/rfc6455>

RFC 4346, *The Transport Layer Security (TLS) Protocol Version 1.1*  
<https://tools.ietf.org/html/rfc4346>

RFC 5246, *The Transport Layer Security (TLS) Protocol Version 1.2*  
<https://tools.ietf.org/html/rfc5246>

RFC 8446, *The Transport Layer Security (TLS) Protocol Version 1.3*  
<https://tools.ietf.org/html/rfc8446>

RFC 6066, *Transport Layer Security (TLS) Extensions: Extension Definitions*  
<https://tools.ietf.org/html/rfc6066>

RFC 7301, *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension*  
<https://tools.ietf.org/html/rfc7301>

ITU X.690, *Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*   
<https://www.itu.int/rec/T-REC-X.690>

RFC 7468, *Textual Encodings of PKIX, PKCS, and CMS Structures*  
<https://www.rfc-editor.org/rfc/rfc7468>

RFC 1035, *DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION*  
<https://www.rfc-editor.org/rfc/rfc1035>

RFC 8484, *DNS Queries over HTTPS (DoH)*   
<https://www.rfc-editor.org/rfc/rfc8484>

RFC 5905, *Network Time Protocol Version 4: Protocol and Algorithms Specification*  
<https://www.rfc-editor.org/rfc/rfc5905>

IEEE 802.  
<https://standards.ieee.org/featured/ieee-802/>

MQTT 3.1.1 Standard.  
<http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>

# 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

IO

an abbreviation for “Input and Output”

4.9

Microcontroller

a single integrated circuit with one or more CPUs, memory, and programmable IO

4.10

protocol

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

4.11

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.

4.12

remote endpoint

a computing system in communication with the microcontroller

4.13

sensing device

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

4.14

sensor

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

4.15

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.

4.16

sensor configuration

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

4.17

synthetic measurement

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

# 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.

# Overview

## ECMAScript

This Standard builds on Standard ECMAScript as defined by Ecma TC39. As of this writing, that is ECMAScript 2022.

This Standard is not an extension or subset of ECMAScript 2022. It is a set of APIs to be used with 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.

## 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.

## 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.

## 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.

## 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](https://www.iana.org/assignments/uri-schemes/prov/embedded) 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](https://github.com/tc39/proposal-built-in-modules#namespace).

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.

## 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.

## 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 (e.g. hz instead of hertz).

## IP address

This Standard represents an IP address value as a string.

An IPv4 address has the form x.x.x.x, where x is a decimal value from 0 to 255 and the values are separated by periods.

An IPv6 address has the form y:y:y:y:y:y:y:y, where y is a hexadecimal value from 0x0000 to 0xFFFF and the values are separated by colons.

## MAC address

This Standard represents a media access control address (MAC address) value as a string. The value has the form zz:zz:zz:zz:zz:zz, where zz is a two-digit hexadecimal value from 0x00 to 0xFF and the values are separated by colons.

## Byte Buffer

This Standard uses the term “Byte Buffer” to mean an instance of the following JavaScript types: ArrayBuffer (resizable or not), SharedArrayBuffer (growable or not), Uint8Array, Int8Array, and DataView.

# 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](https://tc39.es/ecma262/#sec-ecmascript-standard-built-in-objects), or successor.

# 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](#alg-base-class-pattern) of the Base Class Pattern.

## Asynchronous methods

By default, methods are synchronous: they consume their inputs, perform their work, and generate their result by the time they return. A class may provide asynchronous methods.

Asynchronous methods take an optional final argument which is a completion callback function. A completion callback function is called once, at the completion of the operation to indicate success or failure and deliver the result of the operation.

The first argument to the completion callback is always a result code. A value of null indicates success; an Error object indicates failure. Additional arguments may be specified by the method.

Whether or not a completion callback is provided, the method is performed asynchronously.

If an instance provides any asynchronous methods, it should provide an asynchronous close method.

NOTE As defined here, an “asynchronous method” is **not** a JavaScript function declared with the async keyword. Here “asynchronous” refers only to the operation being performed in without blocking the current thread of execution.

## **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.

## **close** method

The close method releases all resources associated with the instance before completing.

Once the close operation 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.

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

Once close has been called, calling other methods on the instance throws an exception. The sole exception is close itself which is safe to call multiple times.

For synchronous close:

* the close operation is complete when it returns
* no callbacks may be invoked after the close method is called

For asynchronous close:

* the close operation completes some time after it returns
* the completion callback, if provided, is invoked after close completes
* callbacks may be invoked after close is called and before the completion callback is invoked
* if possible, pending asynchronous operations should be canceled

## **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.

## 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](https://tc39.es/ecma262/#sec-happens-before). 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.

# 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](#alg-io-class-pattern) of the IO Class Pattern.

## 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}).

## 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.

## 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.

## **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 accepts a data argument that is a Number or Byte Buffer. When it is a Number, read allocates the result as an ArrayBuffer with up to as many bytes as the Number argument. When it is a Byte Buffer, read fills in as many bytes as possible and the result is the number of bytes read as a Number .

For synchronous read, the result is the return value. For asynchronous read, the result is passed to the completion callback as the second argument.

If a resizable Byte Buffer is passed to an asynchronous read and the buffer shrinks so that it cannot hold the number of bytes requested at the time the operation is queued, an error is passed to the completion callback. It is implementation dependent if and how the content of the buffer is modified.

## **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.

When the format property is "buffer", the write method accepts a data argument that is a Byte Buffer.

Calls to write must write all the data provided. If all the data cannot be output, write must not output any data and instead must throw an exception.

If a non-shared Byte Buffer is passed to an asynchronous write method, the implementation sends the contents of the buffer from the time the operation is queued. If a shared buffer is passed, the implementation may read from the buffer at any time; the caller is responsible for ensuring that the bytes are not modified until the completion callback is invoked.

## **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 - a Byte Buffer. For buffer types with defined **byteOffset** and **byteLength** properties, these restrict the bytes accessed in views. Implementations always allocate **ArrayBuffer** instances for return values.
* 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.

## 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.

### 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.

### onWritable

The onWritable callback is invoked when the instance can 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.

### 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.

# 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.

## 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](#alg-io-digital) of the Digital IO Class.

### 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. |

### Callbacks

onReadable()

Invoked when the input value changes depending on the value of the edge property.

### Data format

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

### Notes

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

## 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](#alg-io-digital-bank) of the DigitalBank bank IO Class.

### 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 DigitalBank.Input, DigitalBank.InputPullUp, DigitalBank.InputPullDown, DigitalBank.InputPullUpDown, DigitalBank.Output, or DigitalBank.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. |

### 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.

### 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.

### 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.

## Analog input

The Analog IO class represents an analog input source.

import Analog from "embedded:io/analog";

See Annex A for the [formal algorithms](#alg-io-analog-input) of the Analog IO Class.

### 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. |

### 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.

### **resolution** property

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

## 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](#alg-io-pulse-width-modulation) of the PWM IO Class.

### 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. |

### 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.

### **resolution** property

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

### **hz** property

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

### Notes

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

## I²C – synchronous IO

The I2C class implements an I²C Initiator to communicate with an I²C Peripheral over I²C bus. The I2C class performs synchronous IO.

import I2C from "embedded:io/i2c";

If synchronous IO is not supported, the constructor throws.

See Annex A for the [formal algorithms](#alg-io-i2c) of the I2C IO Class.

### 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.

### Data format

The I2C class data format is always "buffer".

### 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.

### Methods

NOTE When number of bytes to read or write is zero the target device address is sent over the I²C bus but no data bytes follow.

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.

read(byteLength | buffer[, stop])

The first argument follows the [behavior](#io-pattern-read) 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.

## I²C – asynchronous IO

The I2C.Async class implements an I²C Initiator to communicate with an I²C Peripheral over I²C bus using asynchronous IO.

import I2C from "embedded:io/i2c";

The I2C class provides an implementation using asynchronous IO through the I2C.Async constructor. The I2C.Async constructor is only present if asynchronous IO is supported.

Asynchronous operations occur serially in the order issued.

See Annex A for the [formal algorithms](#alg-io-i2c-async) of the I2C.Async IO Class.

### Properties of constructor options object

Same as synchronous I2C.

### Data format

Same as synchronous I2C.

### Specifying stop bit with read and write methods

Same as synchronous I2C.

### Methods

read(byteLength | buffer)  
read(byteLength | buffer, stop)  
read(byteLength | buffer, callback)  
read(byteLength | buffer, stop, callback)

The byteLength, buffer, and stop arguments are the same as synchronous I2C. There is no return value. The callback property is a completion callback function. The second argument is either the byteLength or buffer that would be returned by synchronous read.

write(buffer)  
write(buffer, stop)  
write(buffer, callback)  
write(buffer, stop, callback)

The buffer and stop arguments are the same as synchronous I2C. The callback property is a completion callback function.

## System management bus (SMBus) – synchronous IO

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

import SMBus from "embedded:io/smbus";

If synchronous IO is not supported, the constructor throws.

See Annex A for the [formal algorithms](#alg-io-smbus) of the SMBus IO Class.

### 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. |

### 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](#io-pattern-read) of the IO Class Pattern read method for the "buffer" data format.

writeBuffer(register, buffer)

Write a stream of bytes from the buffer argument starting at the specified register. The buffer argument to writeBuffer follows the [behavior](#io-pattern-write) of the IO Class Pattern write method for the "buffer" data format.

readQuick()

Send an SMBus Quick command with the Read/Write bit set to 1.

writeQuick()

Send an SMBus Quick command with the Read/Write bit set to 0.

receiveByte()

Read an 8-bit unsigned value.

sendByte(command)

Send the 8-bit unsigned command byte.

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

## System management bus (SMBus) – asynchronous IO

The SMBus.Async class extends the I2C.Async class with additional methods to communicate with devices that implement the SMBus protocol using asynchronous IO.

import SMBus from "embedded:io/smbus";

The SMBus class provides an implementation using asynchronous IO through the SMBus.Async constructor. The SMBus.Async constructor is only present if asynchronous IO is supported.

Asynchronous operations occur serially in the order issued.

See Annex A for the [formal algorithms](#alg-io-smbus-async) of the SMBus.Async IO Class.

### Properties of constructor options object

Same as synchronous SMBus.

### Methods

readUint8(register[, callback])  
readUint16(register[, bigEndian][, callback])  
readBuffer(register, byteLength | buffer[, callback])  
readQuick([callback])  
receiveByte([callback])

All asynchronous SMBus methods read methods accept an optional final completion callback argument that behaves consistently with the read behavior of the IO Class Pattern.

writeUint8(register, value[, callback])  
writeUint16(register, value[, bigEndian][, callback])  
writeBuffer(register, buffer[, callback])  
writeQuick([callback])  
sendByte(command[, callback])

All asynchronous SMBus methods write methods accept an optional final completion callback argument that behaves consistently with the write behavior of the IO Class Pattern.

## Serial

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

import Serial from "embedded:io/serial";

See Annex A for the [formal algorithms](#alg-io-serial) of the Serial IO Class.

### 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.

### 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](#io-pattern-read) of the IO Class Pattern read method for the "buffer" data format with one addition: if there are no arguments and data is available to read, 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 | buffer)

When using the "number" data format, the first argument is a byte value to transmit.

If the output buffer cannot accept all the bytes to be written, 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.

### 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.

### 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".

## 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](#alg-io-spi) of the SPI IO Class.

### 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).

### Data format

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

### Methods

read(byteLength | buffer)

The first argument follows the [behavior](#io-pattern-read) 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.

## 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](#alg-io-pulse-count) of the PulseCount IO Class.

### 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. |

### Data format

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

### 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.

### 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.

## 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](#alg-io-tcp-socket) of the TCP IO Class.

### Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| address | A string with the IP address of the remote endpoint to connect to. This property is required. |
| 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, 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. |

### 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](#io-pattern-read) of the IO Class Pattern read method for the "buffer" data format with one addition: if there are no arguments and data is available to read, 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.

### 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.

### 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".

### **remoteAddress** property

The read-only remoteAddress property provides the IP address of the remote end-point as a string. If the remote address is not available, the value is undefined.

### **remotePort** property

The read-only remotePort property provides the port of the remote end-point as a number. If the remote port is not available, the value is undefined.

## 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](#alg-io-tcp-listener) of the Listener IO Class.

### 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. |

### 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.

### 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.

### Data format

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

## 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](#alg-io-udp-socket) of the UDP IO Class.

### 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. |

### 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 a Byte Buffer, 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 a Byte Buffer, the remote address string, and the remote port number.

### 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.

### Data format

The UDP class data format is always "buffer".

## TLS Client socket

The TLSClient network socket class implements a logical subclass of the TCP class that secures the connection using Transport Layer Security (TLS).

import TLS from "embedded:io/socket/tcp/tls";

A TLS implementation may use certificates from a certificate store. The certificate store is implementation dependent and not specified by this Standard.

All certificate and key data uses DER (binary) format, not PEM (Base64 encoded text).

### Properties of constructor options object

The TLS Client socket extends the TCP socket’s options object with a required tls property set to an object that contains the TLS options.

The following TLS version strings are defined: "TLSv1.3", "TLSv1.2", "TLSv1.1".

|  |  |
| --- | --- |
| **Property** | **Description** |
| tls | An object with the following properties. This property is required. |
| tls.host | Supports Server Name Indication (SNI). A string with the host name of the remote endpoint. This property is required. |
| tls.minimumVersion | A TLS version string indicating the minimum acceptable TLS version for the connection. This property is optional and the default is implementation dependent. |
| tls.maximumVersion | A TLS version string indicating the maximum acceptable TLS version for the connection. This property is optional and the default is implementation dependent. |
| tls.applicationLayerProtocol | Supports Application-Layer Protocol Negotiation Extension (ALPN). A String or Byte Buffer to indicate support for a single application layer protocol or an Array of one or more String and Byte Buffers to indicate support for multiple application layer protocols. This property is optional. |
| tls.maximumFragmentLength | Supports Maximum Fragment Length. A number indicating the maximum fragment size in bytes. This property is optional. If not present, the maximum fragment length is not negotiated. |
| tls.ca | A Byte Buffer or an Array of Byte Buffer containing certificates chains for the connection. This property is optional. |
| tls.clientKey | A Byte Buffer or an Array of Byte Buffers containing client keys for the connection. This property is optional. |
| tls.clientCertificate | A Byte Buffer or an Array of Byte Buffers containing client certificates for the connection. This property is optional. |

### write(buffer)

The write method returns the updated writable count. This may be reduced by more than the size of the buffer written because of TLS protocol overhead.

# 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](#alg-io-provider-class-pattern) of the IO Provider Class Pattern.

## **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.

## **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).

A class may specify that close accepts an optional callback function to invoke after the close operation completes. The callback must the last argument to close. The first argument to the callback is a Number with 0 indicating success and other values indicating failure. Pending callbacks from other operations are invoked before the callback passed to close.

## 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.

# 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](#alg-peripheral-class-pattern) of the Peripheral Class Pattern.

## **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.

## **close** method

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

## **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.

## 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;  
 }  
}

# 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](#alg-sensor-class-pattern) of the Sensor Class Pattern.

## **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.

## **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.

**To do**: note on structuring the options object for compound sensors

## **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 callers 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.

## 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.

# 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.

## Compound sensors

A single physical sensor may provide more than one kind of sensor reading. For example, a single sensor package may include both a temperature sensor and a humidity sensor. When a single physical sensor contains two or more logical sensors, the Sample object returned by the sample method must contain a sub-object for each logical sensor. For example, a physical sensor that includes both temperature and humidity sensors would return a Sample object with the following properties:

{  
 hygrometer: {  
 humidity: 0.5  
 },  
 thermometer: {  
 temperature: 23  
 }  
}

The name of the property that contains the sub-object is defined by the sensor class. Here thermometer is defined by the Temperature sensor class and hygrometer is defined by the Humidity sensor class. Each sub-object contains a Sample object as defined by its sensor class.

## Accelerometer

The Accelerometer class implements access to a three-dimensional accelerometer. The property name accelerometer is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-accelerometer) of the Accelerometer sensor class.

### Properties of a sample object

These properties are compatible with the attributes of the same name in the [W3C Accelerometer draft](https://w3c.github.io/accelerometer/).

|  |  |
| --- | --- |
| **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. |

## Ambient light

The AmbientLight class implements access to an ambient light sensor. The property name lightmeter is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-ambient-light) of the AmbientLight sensor class.

### Properties of sample object

These properties are compatible with the attributes of the same name in the [W3C Ambient Light Sensor draft](https://www.w3.org/TR/ambient-light/).

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

## Atmospheric pressure

The AtmosphericPressure class implements access to an atmospheric pressure sensor or barometer. The property name barometer is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-atmospheric-pressure) of the AtmosphericPressure sensor class.

### Properties of a sample object

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

## Carbon Dioxide

The CarbonDioxide class implements access to a sensor that detects the amount of carbon dioxide in air. The property name carbonDioxideDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-carbonDioxide) of the CarbonDioxide sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| CO2 | A number that represents the sampled carbon dioxide in parts per million. This property is required. |

## Carbon Monoxide

The CarbonMonoxide class implements access to a sensor that detects the amount of carbon monoxide in air. The property name carbonMonoxideDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-carbonMonoxide) of the CarbonMonoxide sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| CO | A number that represents the sampled carbon monoxide in parts per million. This property is required. |

## Dust

The Dust class implements access to a sensor that detects the amount of dust suspended in air. The property name dustDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-dust) of the Dust sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| dust | A number that represents the sampled dust levels in micrograms per cubic meter. This property is required. |

## Gyroscope

The Gyroscope class implements access to a three-dimensional gyroscope. The property name gyroscope is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-gyroscope) of the Gyroscope sensor class.

### Properties of a sample object

These properties are compatible with the attributes of the same name in the [W3C Gyroscope draft](https://www.w3.org/TR/gyroscope/).

|  |  |
| --- | --- |
| **Property** | **Description** |
| x | A number that represents the sampled angular velocity around the x axis in radian per second. This property is required. |
| y | A number that represents the sampled angular velocity around the y axis in radian per second. This property is required. |
| z | A number that represents the sampled angular velocity around the z axis in radian per second. This property is required. |

The sign of the sampled angular velocity depends on the rotation direction, with a positive number indicating a clockwise rotation and a negative number indicating a counterclockwise rotation.

## Humidity

The Humidity class implements access to a humidity sensor. The property name hygrometer is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-humidity) of the Humidity sensor class.

### Properties of a sample object

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

## Hydrogen

The Hydrogen class implements access to a sensor that detects the amount of hydrogen in air. The property name hydrogenDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-hydrogen) of the Hydrogen sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| H | A number that represents the sampled hydrogen in parts per million. This property is required. |

## Hydrogen Sulfide

The HydrogenSulfide class implements access to a sensor that detects the amount of hydrogen sulfide in air. The property name hydrogenSulfideDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-hydrogenSulfide) of the HydrogenSulfide sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| H2S | A number that represents the sampled hydrogen sulfide in parts per million. This property is required. |

## Magnetometer

The Magnetometer class implements access to a three-dimensional magnetometer. The property name magnetometer is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-magnetometer) of the Magnetometer sensor class.

### Properties of a sample object

These properties are compatible with the attributes of the same name in the [W3C Magnetometer draft](https://www.w3.org/TR/magnetometer/).

|  |  |
| --- | --- |
| **Property** | **Description** |
| x | A number that represents the sampled magnetic field around the x axis in microtesla. This property is required. |
| y | A number that represents the sampled magnetic field around the y axis in microtesla. This property is required. |
| z | A number that represents the sampled magnetic field around the z axis in microtesla. This property is required. |

## Methane

The Methane class implements access to a sensor that detects the amount of methane in air. The property name methaneDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-methane) of the Methane sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| CH4 | A number that represents the sampled methane in parts per million. This property is required. |

## Nitric Oxide

The NitricOxide class implements access to a sensor that detects the amount of nitric oxide in air. The property name nitricOxideDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-nitricOxide) of the NitricOxide sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| NO | A number that represents the sampled nitric oxide in parts per million. This property is required. |

## Nitric Dioxide

The NitricDioxide class implements access to a sensor that detects the amount of nitric dioxide in air. The property name nitricDioxideDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-nitricDioxide) of the NitricDioxide sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| NO2 | A number that represents the sampled nitric dioxide in parts per million. This property is required. |

## Oxygen

The Oxygen class implements access to a sensor that detects the amount of oxygen in air. The property name oxygenDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-oxygen) of the Oxygen sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| O | A number that represents the sampled oxygen in parts per million. This property is required. |

## Particulate Matter

The ParticulateMatter class implements access to a sensor that detects the amount of particulate matter suspended in air. The property name particulateMatterDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-particulateMatter) of the ParticulateMatter sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| particulateMatter | A number that represents the sampled particulate matter levels in micrograms per cubic meter. This property is required. |

## Proximity

The Proximity class implements access to a proximity sensor or range finder. The property name proximity is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-proximity) of the Proximity sensor class.

### Properties of a sample object

These properties are compatible with the attributes of the same name in the [W3C Proximity Sensor draft](https://w3c.github.io/proximity/).

|  |  |
| --- | --- |
| **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. |

## Soil Moisture

The SoilMoisture class implements access to a soil moisture detector. The property name soilMoistureDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-soilmoisture) of the SoilMoisture sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| moisture | A number between 0 and 1 (inclusive) that represents the sampled relative soil moisture level, with 0 being the most dry and 1 the most wet. This property is required. |

## Switch

The Switch class implements access to a switch sensor. The property name switch is used when part of a compound sensor.

### Properties of sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| position | A number that represents the current state of the switch. This property is required. |

## Temperature

The Temperature class implements access to a temperature sensor. The property name thermometer is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-temperature) of the Temperature sensor class.

### Properties of a sample object

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

## Touch

The Touch class implements access to a touch panel controller. The property name touch is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-touch) of the Touch sensor class.

### 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.

#### 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 |

## Volatile Organic Compounds

The VolatileOrganicCompounds class implements access to a sensor that detects the amount of volatile organic compounds suspended in air. The property name vocDetector is used when part of a compound sensor.

See Annex A for the [formal algorithms](#alg-sensor-VolatileOrganicCompounds) of the VolatileOrganicCompounds sensor class.

### Properties of a sample object

|  |  |
| --- | --- |
| **Property** | **Description** |
| tvoc | A number that represents the sampled total volatile organic compounds in parts per billion. This property is required. |

# 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](https://mipi.org/specifications/dsi).

See Annex A for the [formal algorithms](#alg-display-class-pattern) of the Display Class Pattern.

## 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.

## **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. |

The Display Class Pattern does not define default values for these properties to allow the host to provide default values that are appropriate for its hardware. Implementations may provide the current configuration through the configuration property defined by the Provenance Sensor Class Pattern.

## **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();

## **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 Byte Buffer of pixels. 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.

## **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.

## **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;  
}

## 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.

## Pixel format values

|  |  |
| --- | --- |
| **Value** | **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) |

# Real-Time Clock Class Pattern

A Real-Time Clock (RTC) provides a time-of-day clock. An RTC is commonly used to initialize time on a microcontroller. An RTC is usually a separate hardware component from the microcontroller. It usually maintains the time using a battery so the time survives power being removed from the device.

The RTC Class Pattern conforms to the Peripheral Class Pattern.

## Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| clock | A class constructor options object that describes the hardware connection for the RTC. This property is required. |
| interrupt | A Digital class constructor options object that describes the hardware connection to the RTC’s interrupt. This property is optional. |
| onAlarm() | A function to invoke when an alarm is triggered by the RTC. This property is optional. |

## **configure** method

The following property is defined for the options object.

|  |  |
| --- | --- |
| **Property** | **Description** |
| alarm | The time in milliseconds to set the RTC’s alarm. This value is an ECMAScript time value as a Number. |

## **time** property

The current time of the RTC. Set this property to change the current time of the RTC. This value is an ECMAScript time value contained in a Number.

The resolution of the RTC component may impact the values. For example, an RTC with one-second resolution may return time values with a milliseconds of zero.

If the time is unavailable (for example, because it has not been set or is otherwise invalid on the RTC), the returned value is undefined.

## **configuration** property

The configuration property returns an object containing the current configuration of the RTC. It contains the alarm property, if supported.

The configuration property is [introduced](https://419.ecma-international.org/#-17-provenance-sensor-class-pattern-configuration-property) in the Provenance Sensor Class Pattern.

# Network Interface Class Pattern

The Network Interface Class Pattern builds on the Base Class Pattern to provide access to the network interfaces of a device to monitor the connection state and perform operations.

The physical network interfaces may be physically built into the microcontroller or a separate peripheral. The logical network interfaces are managed by the host.

Creating an instance of a Network Interface class binds to the host’s network interface; it does not initialize the network interface. Closing an instance of a Network Interface class unbinds from the host’s network interface; it does not uninitialize the network interface.

There may be multiple simultaneous instances of a Network Interface class, all bound to the same logical network interface.

See Annex A for the [formal algorithms](#alg-network-interface-class-pattern) of the Network Interface Class Pattern.

## Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| onChanged(name) | A function to invoke when the network interface’s state changes. The name argument is the name of the property that changed. The onChanged property is optional. |
| port | A port specifier that indicates the logical network interface to bind to. This property may be optional or required depending on the implementation of the network interface. |

## **connect** method

Initiates the process of connecting to a network. If a connection attempt is already in progress, connect throws an exception.

The sole argument is an options object. Each Network Interface class defines properties for the options object.

## **disconnect** method

Disconnects from the currently connected network. If in the process of connecting, the connection attempt is abandoned. If already disconnected, does nothing. No arguments are specified.

## **connection** property

The read-only connection property indicates the current connection state of the network interface as a number. The following values are defined:

|  |  |
| --- | --- |
| **Value** | **Description** |
| 0 | unavailable |
| 100 | initializing |
| 200 | disconnected |
| 300 | connecting |
| 400 | connected |
| 500 | IP address assigned |

Larger values indicate a later stage in the connection process. This allows values to be compared with greater and less than operators. Additional states may be added by specific types of network interfaces.

## **MAC** property

The read-only MAC property is the MAC address assigned to the network interface as a string. If the MAC address is unavailable, the value is undefined.

## **address** property

The read-only address property is the IP address assigned to the network interface as a string. If the address has not yet been assigned, the value is undefined.

## Ethernet Network Interface

The Ethernet Network Interface is a logical subclass of the Network Interface Class Pattern for Ethernet network interfaces.

import Ethernet from "embedded:network/interface/ethernet";

See Annex A for the [formal algorithms](#alg-ethernet-network-interface) of the Ethernet Network Interface.

### **connection** property

For an Ethernet network interface, connection 200 (“disconnected”) indicates that the physical Ethernet link has been lost and connection 400 (“connected”) indicates that the physical Ethernet link has been established. Ethernet network interfaces add the following value for connection.

|  |  |
| --- | --- |
| **Value** | **Description** |
| 150 | Ethernet IO initialized |

## Wi-Fi Network Interface

The Wi-Fi Network Interface is a logical subclass of the Network Interface Class Pattern for Wi-Fi network interfaces.

import WiFi from "embedded:network/interface/wifi";

See Annex A for the [formal algorithms](#alg-wifi-network-interface) of the Wi-Fi Network Interface.

### **connect** method

Initiates the process of connecting to a Wi-Fi base station. The connection is defined by the properties of the options object. If a connection attempt is already in progress, connect throws an exception.

|  |  |
| --- | --- |
| **Property** | **Description** |
| SSID | Name of the base station as a String. This property is optional. |
| BSSID | BSSID of the base station as a MAC address formatted string. This property is optional. |
| password | The base station’s password as a string. This property is optional. |
| secure | Boolean that indicates if connections to open access points are allowed. This property is optional and defaults to false. |
| channel | Wi-Fi channel of the base station as a number. This property is optional. |

Either the SSID or BSSID property is required. If both are provided, BSSID is used.

### **scan** method

Initiates a scan for Wi-Fi base stations. The scan is time-limited to no more than 10 seconds. A continuous scan may be performed by repeated scans. If a scan is already active when scan is called, an exception is thrown. The sole argument is an options object.

Properties of the scan options object:

|  |  |
| --- | --- |
| **Property** | **Description** |
| onFound(options) | A callback function to invoke with information about an access point discovered by the scan. This property is required. |
| onComplete() | A callback function invoked when the scan is complete. This property is optional. |
| channel | Wi-Fi channel number to scan as a number. This property is optional. |
| frequency | Wi-Fi frequency to scan: 2.4 or 5. This property is optional and the default is implementation dependent. |
| secure | Limit scan results to secure access points, omitting open access points, as a boolean. This property is optional and defaults to false. |

Properties of the onFound options object for each access point found by the scan:

|  |  |
| --- | --- |
| **Property** | **Description** |
| SSID | Service Set Identifier of the access point as a string. |
| BSSID | Basic Service Set Identifier of the access point as a MAC address formatted string. |
| RSSI | Radio Signal Strength Indicator of the access point as a number. |
| channel | Channel number of the access point as a number. |
| security | Security mode of the access point as a string. |

The scan cannot be cancelled. If the instance is closed while scanning, the host may complete the scan but must not invoke the callbacks.

### **SSID** property

The Service Set Identifier of the connected access point as a string or undefined if not connected. Read-only.

### **BSSID** property

The Basic Service Set Identifier of the connected access point as a MAC address formatted string or undefined if not connected. Read-only.

### **RSSI** property

The Radio Signal Strength Indicator of the connected access point as a number or undefined if not connected. Read-only.

### **channel** property

The channel number of the connected access point as a number or undefined if not connected. Read-only.

# Domain Name Resolver Class Pattern

The Domain Name Resolver Class Pattern resolves DNS names to IP addresses. It conforms to the Base Class Pattern. The Domain Name Resolver Class Pattern is not instantiated directly. Logical subclasses of the Domain Name Resolver are instantiated, such as DNS over UDP and DNS over HTTPS.

## **resolve** method

The resolve method begins the process of resolving a DNS name to an address. Several resolve operations may be queued and be pending at the same time. The resolve requests complete in an implementation dependent order, which may not be the order requested.

The first argument is a required options object. The second argument is a required completion callback function that is invoked when resolution completes. If successful, the resolved address is provided in the second argument and the requested hostname in the third.

## Properties of resolve options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| host | A string containing the hostname to resolve. This property is required. |

The host property may be either a Domain Name or an IP address. If it is an IP address, the completion callback is invoked with the resolved address and request hostname arguments set to that IP address.

## DNS over UDP

DNS over UDP is a logical subclass of the Domain Name Resolver Class Pattern that resolves DNS names over UDP.

import Resolver from "embedded:network/dns/resolver/udp";

### Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| socket | A UDP class constructor options object for a UDP socket. This property is required. |
| servers | Array of one or more IP address strings to use as DNS servers. This property is required. |

## DNS over HTTPS (DoH)

DNS over HTTPS is a logical subclass of the Domain Name Resolver Class Pattern that resolves DNS names using an HTTPS connection (DoH).

import Resolver from "embedded:network/dns/resolver/doh";

### Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| http | An HTTP Client class constructor options object. This property is required. |
| servers | An array of one or more objects containing host and address properties to use as DoH servers. This property is required. |

# NTP Client

The NTP Client retrieves the current time from a network time source using the Network Time Protocol (NTP). It conforms to the Base Class Pattern.

Implementations may use the Simple Network Time Protocol (SNTP).

import NTP from "embedded:network/ntp/client";

## Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| socket | UDP class constructor options object. This property is required. |
| servers | An array of one or strings indicating the NTP hosts to use to synchronize time. This property is required. |

## **getTime** method

The getTime method initiates a time synchronization operation. Only one time synchronization operation may be active at a time. If a second request is made before the current request completes, getTime throws. The sole argument is a required completion callback function that is invoked when synchronization completes. If successful, the time value is provided in the second argument.

# HTTP Client class pattern

The HTTP Client class pattern makes one or more Hypertext Transfer Protocol (HTTP/1.1) requests to a single host. It conforms to the Base Class Pattern.

import HTTPClient from "embedded:network/http/client";

## Data format

The HTTPClient class data format is always "buffer".

## Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| socket | An object containing a TCP class constructor options object. This property is required. |
| port | The remote port number to connect to as a number. This property is optional and defaults to 80. |
| host | The remote hostname to connect to as a string. This property is required. |
| dns | A Domain Name Resolver class constructor options object to use to resolve the host. This property is required. |
| onError | A function to invoke when the remote connection closes. This property is optional. |

## **close** method

In addition to the behaviors defined in the Base Class Pattern, all outstanding requests are cancelled.

## **request** method

Queues an HTTP request described by the required options object, the sole argument.

The options object supports the following properties:

|  |  |
| --- | --- |
| **Property** | **Description** |
| method | The HTTP method to use to access the resource as a string. This property is optional and defaults to "GET". |
| path | The HTTP resource to access as a string. This property is optional and defaults to "/". |
| headers | A Map instance containing request headers. The map keys are the header names and their values are the header values. This property is optional. |
| onHeaders(status, headers) | A function to invoke to provide the HTTP status result code and a Map containing the response headers. The map keys are the header names normalized to lowercase and their values are the header values. This property is optional. |
| onReadable(count) | A function to invoke when bytes are available to read from the HTTP response body. The count argument is a number indicating the number of bytes available to read. This property is optional. |
| onWritable(count) | A function to invoke when the HTTP request is ready to receive bytes for the request body. The count argument is a number indicating the maximum number of bytes that may be written. The onWritable callback is only invoked if a request has a request body. To signal that a request has a request body, set either the content-length header to a non-zero value or the transfer-encoding header to "chunked". This property is optional. |
| onDone() | A function to invoke when the HTTP request has been completed successfully. This property is optional. |

The return value of the request method is an HTTP Client Request instance. This instance is the receiver when the callback functions of the options object are invoked. The request instance has read and write methods.

## HTTP Client Request instance

The HTTP Client Request instance conforms to the IO Class Pattern. It is instantiated by the HTTP Client and so has no constructor. No close method is available because the protocol does not support cancelling a request in progress.

### **read** method

Reads payload body from the HTTP request’s response. If this HTTP Request instance is not currently receiving the response body, returns undefined.

### **write** method

Writes to the payload body of the HTTP request’s request. If this HTTP Request instance is not currently sending the request body, write throws an exception.

For HTTP requests using chunked transfer-encoding, calling write with no arguments signals the end of the request body.

The write method returns the number of bytes that may be written. This may be reduced by more than the size of the payload due to overhead in the protocol.

# HTTP Server class pattern

The HTTP Server class pattern responds to Hypertext Transfer Protocol (HTTP/1.1) requests. It conforms to the Base Class Pattern.

import HTTPServer from "embedded:network/http/server";

## Data format

The HTTPServer class data format is always "buffer".

## Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| io | An object containing a TCP Listener class constructor options object. This property is required. |
| port | The port number to listen on to as a number. This property is optional and defaults to 80. |
| onConnect(connection) | A function to invoke when a new connection is initiated. It is passed an HTTP Server Connection instance as the sole argument. This property is required. |

## **close** method

In addition to the behaviors defined in the Base Class Pattern, all active connections are closed.

## HTTP Server Connection instance

The HTTP Server Connection instance conforms to the IO Class Pattern. It is instantiated by the HTTP Server and so has no constructor.

### **close** method

Connections are automatically closed when the request is complete. Calling the close method before that terminates a connection prematurely (for example, for a connection timeout).

### **detach** method

The detach method returns the TCP socket instance used by this connection. There are no arguments. On return, the HTTP Server Connection instance maintains no reference to the instance and is effectively closed. The detach capability is useful for protocols that use the HTTP Upgrade mechanism.

### **accept** method

The accept method accepts the incoming connection so that processing of the HTTP request may begin. The sole argument is an options object which contains callback functions to invoke as the HTTP request is processed.

#### Properties of accept options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| onRequest(method, path, headers) | A callback function to invoke after the HTTP request headers have been received. The first argument is the HTTP request method as a string. The second argument is the HTTP request path as a string. The third argument is a map containing the headers. The map keys are the header names normalized to lowercase and their values are the header values. This property is optional. |
| onReadable(count) | A callback function to invoke when data is available to read from the request body. This property is optional. |
| onResponse(response) | A callback function to invoke when the request body has been fully received. The sole argument is an options object with a status property set to 200 and a headers property set to an empty map. The callback may update these values. The option object is passed to the respond method to begin transmitting the HTTP response. This property is optional. |
| onWritable(count) | A callback function to invoke when there is room in the output buffers to transmit part of the response body. This property is optional. |
| onDone() | A callback function to invoke when the request successfully completes. This property is optional. |
| onError(error) | A callback function to invoke if an error occurs before the response is complete, such as the connection being terminated. This property is optional. |

### **respond** method

The respond method is called to begin transmitting the HTTP response. The respond method may only be called once for a given instance and must be called after the request body has been fully received. The sole argument to the respond method is an options object.

#### Properties of respond options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| status | A number indicating the status code for the HTTP response. This property is required. |
| headers | A map containing the HTTP response headers. The map keys are the header names and their values are the header values. This property is required. |

### **read** method

Reads the payload body from the HTTP request body. If this HTTP Server Connection instance is not currently receiving the request body, returns undefined.

### **write** method

Writes to the payload body of the HTTP response body. If this HTTP Server Connection instance is not currently sending the response body, write throws an exception.

For HTTP Server Connection instances using chunked transfer-encoding, calling write with no arguments signals the end of the response body.

The write method returns the number of bytes that may be written. This may be reduced by more than the size of the payload due to overhead in the protocol.

### **route** property

The route of an HTTP Server Connection instance is an object that is used to override the callbacks set in the call to accept. This may be used to dispatch incoming requests to different handlers based on the request method, path, and request headers.

If the route is set from within the onRequest callback, the onRequest callback of the route is called immediately.

The instance copies the callback functions. Changes to the properties of the route after setting the route are ignored.

# HTTP Server Connection routes

## Static Data route

The Static route sends a buffer or string as an HTTP Response body.

import StaticRoute from "embedded:network/http/server/route/static";

connection.route = {  
 ...StaticRoute,  
 data: "hello, world"  
};

### Properties of route

|  |  |
| --- | --- |
| **Property** | **Description** |
| data | A Byte Buffer or string to be transmitted as the HTTP Response body. If the value is a string, it is transmitted as UTF-8 data. This property is required. |
| contentType | The MIME type of response body to be set as the HTTP Content-Type header. This property is optional and defaults to “text/html”. |

## WebSocket Handshake route

The WebSocket Handshake route implements the server side of the WebSocket handshake to upgrade an HTTP connection to the WebSocket protocol.

import WebSocketHandshake from "embedded:network/http/server/route/ws/handshake";

The onDone callback of the route is invoked when the handshake completes successfully; onError, if the handshake fails. After the handshake succeeds, the TCP socket may be detached and used with a WebSocket implementation.

connection.route = {  
 ...WebSocketHandshake,  
 onDone() {  
 const ws = new WebSocketClient({  
 socket: this.detach(),  
 onReadable(count, options) {  
 }  
 });  
 }  
 onError() {  
 console.log("failed");  
 }  
};

### Properties of route

|  |  |
| --- | --- |
| **Property** | **Description** |
| protocol | Array of strings. This property is optional. |

# WebSocket Client class pattern

The WebSocket Client class pattern establishes a connection to an endpoint hosting a WebSocket server and exchanges messages using the WebSocket protocol. The WebSocket Client class pattern conforms to the IO Class Pattern.

import WebSocketClient from "embedded:network/ws/client";

The WebSocket Client class pattern replies to ping and close messages by replying with a pong or close message with the same payload received, as required by the protocol.

## Data format

The WebSocketClient class data format is always "buffer".

## Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| socket | An object containing a TCP Class constructor options object. This property is optional. |
| host | The remote hostname to connect to as a string. This property is optional. |
| attach | An instance of a TCP Class. This property is optional. |
| port | The remote port number to connect to as a number. This property is optional and defaults to 80. |
| protocol | The WebSocket sub-protocol as a string. This property is optional. |
| headers | A Map of HTTP headers to add to the request. The map keys are the header names and their values are the header values. This property is optional. |
| dns | A Domain Name Resolver class constructor options object to use to resolve the host. This property is required. |
| onReadable(count, options) | A function to invoke when part of a WebSocket binary or text message is available to read. The first argument is the number of bytes available to read. The second argument is an options object. It has a more property set to false if this is the last fragment of a message and true if there is at least one more fragment. It has a binary property set to true for binary messages and false for text messages. This property is optional. |
| onWritable(count) | A function to invoke when more data may be written to the connection. The sole argument indicates the number of bytes that maybe written. This property is optional. |
| onError(error) | A function to invoke when the remote connection terminates unexpectedly. This property is optional. |
| onControl(opcode, payload) | A function to invoke when a control message is received. The first argument is the control message opcode. The second argument is an ArrayBuffer containing the complete control message payload. This property is optional. |
| onClose() | A function to invoke when the connection closes cleanly. This property is optional. |

Either both socket and host are required or attach is required. The attach property takes precedence.

## **close** method

The close method does not initiate a clean close, as defined by the WebSocket protocol, of the connection (use write with a close opcode instead).

## **read** method

A single call to read returns bytes from the current message. Once the current message has been completely read, the onReadable callback is invoked when the next message is available to read.

## **write** method

The write method sends both message data and control messages. The first argument contains the message payload in a Byte Buffer. The second argument is an options object that has the following properties to specify the message to send.

|  |  |
| --- | --- |
| **Property** | **Description** |
| binary | A boolean value set to true for a binary payload and false for a text payload. This property is optional and defaults to true. |
| more | A boolean value set to false for the last fragment of a message and true for all others. This property is optional and defaults to false. |
| opcode | This property is a number specifying the opcode of a control message (the data argument is the control message’s payload). This property is optional and must not be set for text and binary messages. Because control messages cannot be fragmented, the more property is ignored when opcode is present. |

The write method may be used to send all or part of a single binary or text message based on the properties of the options object.

The options object is optional. If not provided, the default values are used.

The return value is the number of bytes that may be written. This may be reduced by more than the size of the payload due to overhead in the protocol.

## Static properties of the constructor

The following properties are present on the constructor. The property names and values correspond to WebSocket opcodes. The values are numbers and the properties are read-only.

|  |  |
| --- | --- |
| **Property** | **Value** |
| text | 1 |
| binary | 2 |
| close | 8 |
| ping | 9 |
| pong | 10 |

# MQTT Client class pattern

The MQTT Client class pattern establishes a connection to a remote endpoint hosting an MQTT server (broker) and exchanges messages using the MQTT protocol ([MQTT Version 3.1.1, OASIS Standard, 29 October 2014 6455](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.doc)). It allows messages of unlimited size to be sent and received, and supports all control messages.

import MQTTClient from "embedded:network/mqtt/client";

The MQTT Client class pattern must implement the following:

* Transmit keep alive message if configured with a non-zero keep-alive interval
* Reply to PINGREQ messages with PINGREQ

The MQTT Client class pattern should implement the following. If a quality level is not implemented, sending a PUBLISH or SUBSCRIBE message with that quality of service level must throw an exception.

* Reply to PUBLISH with PUBACK for quality of service 1
* Reply to PUBLISH with PUBREC for quality of service 2
* Reply to PUBREL with PUBCOMP for quality of service 2
* Reply to PUBREC with PUBREL for quality of service 2

The MQTT Client class pattern may not implement the following. They may be provided by layers built on the MQTT Client class pattern.

* caching messages and, consequently, message retransmit messages after disconnect
* reconnect
* maintaining a list of active subscriptions

NOTE This specification supports MQTT Version 3. It is designed to be extensible to support MQTT Version 5.

## Data format

The MQTTClient class data format is always "buffer".

## Properties of constructor options object

|  |  |
| --- | --- |
| **Property** | **Description** |
| socket | An object containing a TCP Class constructor options object. This property is required. |
| port | The remote port number to connect to as a number. This property is optional and defaults to 1883. |
| host | The remote hostname to connect to as a string. This property is required. |
| dns | A Domain Name Resolver class constructor options object to use to resolve the host. This property is required. |
| onReadable(count, options) | A function to invoke when part of an MQTT message is available to read. The first argument is the number of bytes available to read. The second argument is an options object. It has a more property set to false if this is the last fragment of a message and true if there is at least one more fragment. For the first fragment of a message, the options object contains topic property with a string indicating the message topic, a QoS property with a number indicating the quality of service, and a byteLength property with a number indicating the total number of bytes in the message. The onReadable property is optional. |
| onWritable(count) | A function to invoke when more data may be written to the connection. The sole argument is a number indicating how many bytes may be written. This property is optional. |
| onError(error) | A function to invoke when the remote connection terminates. This property is optional. |
| onControl(opcode, message) | A function to invoke when a control message is received. The first argument is the control message opcode. The second argument is an object containing an operation property indicating the control message (CONNACK, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBACK, UNSUBACK, PINGREQ, etc.) and an id property if included in the message. The SUBACK message payload is provided on the payload property as an array of byte values. The onControl property is optional. |
| id | The MQTT client identifier as a string. This property is optional and defaults to an empty string. |
| user | The MQTT user for establishing a connection as a string. This property is optional and defaults to an empty string. |
| password | The MQTT password for establishing a connection as a string or Byte Buffer. This property is optional and defaults to an empty string. |
| keepAlive | The MQTT connection keep-alive time in milliseconds as a number. This property is optional and defaults to 0. (This value is in milliseconds as required for durations in this specification. The MQTT protocol uses seconds for the keep-alive value.) |
| clean | The MQTT clean session flag as a boolean. This property is optional and defaults to true. |
| will | An object with the following properties. This property is optional. |
| will.topic | The topic for the MQTT will for this connection as a string. This property is optional. |
| will.message | The message for the MQTT will for this connection as a String or Byte Buffer. This property is optional. |
| will.QoS | The requested quality of service for the will message for this connection as number with values 0, 1, or 2. This property is optional and defaults to 0. |
| will.retain | A Boolean indicating whether the will message should be retained by the server. This property is optional and defaults to false. |

## **close** method

The close method does not send an MQTT close message (use write with a DISCONNECT opcode to initiate a clean disconnect).

## **read** method

A single call to read returns bytes from the current message. Once the current message has been completely read, the onReadable callback is invoked when the next message is available to read.

## **write** method

The write method sends both message data and control messages. The first argument contains the message payload in a Byte Buffer. The second argument is an options object that has the following properties to specify the message to send.

The options object has the following properties to specify the message to publish or control message to send.

|  |  |
| --- | --- |
| **Property** | **Description** |
| operation | This property is a number specifying the opcode of a control message (the data argument is the control message’s payload). This property is optional and defaults to PUBLISH. |
| id | A number specifying the id for the message. If an id is not provided the MQTT client generates one. As a rule, either the caller should provide the id for all messages or none to avoid the possibility of values colliding. This property is optional. |
| topic | A string specifying an MQTT topic for a PUBLISH message. This property is required for PUBLISH messages. |
| QoS | A number specifying the quality of service of PUBLISH message. Allowed values are 0, 1, and 2. This property is for PUBLISH messages only. It is optional and defaults to 0. |
| retain | A boolean indicating if a PUBLISH message should be retained by the server. This property is for PUBLISH messages only. It is optional and default to false. |
| duplicate | A boolean indicating if a PUBLISH message is being retransmitted by the client. This property is for PUBLISH messages only. It is optional and default to false. |
| byteLength | A number indicating the total size of a PUBLISH message to allow a single PUBLISH message payload to be split across two or more calls to write. This property is optional and defaults to data.byteLength |
| items | An array of objects indicating the topics to subscribe or unsubscribe from. Each object must contain a topic property with a string indicating the topic and, for SUBSCRIBE messages, may contain an optional QoS property with the requested quality of service as a value of 0, 1, or 2. This property is required for SUBSCRIBE and UNSUBSCRIBE messages and must contain at least one element. |

The options object is required, except when writing fragments of a PUBLISH message after the first fragment.

It is an error to call write before the CONNACK control message has been received.

The return value is the number of bytes that may be written. This may be reduced by more than the size of the payload due to overhead in the protocol.

## Static properties of the constructor

The following properties are present on the constructor. The property names and values correspond to MQTT Control Packet types in Section 2.1.1 of the MQTT 3.1.1 Standard. The values are numbers and the properties are read-only.

|  |  |
| --- | --- |
| **Property** | **Value** |
| CONNECT | 1 |
| CONNACK | 2 |
| PUBLISH | 3 |
| PUBACK | 4 |
| PUBREC | 5 |
| PUBREL | 6 |
| PUBCOMP | 7 |
| SUBSCRIBE | 8 |
| SUBACK | 9 |
| UNSUBSCRIBE | 10 |
| UNSUBACK | 11 |
| PINGREQ | 12 |
| PINGRESP | 13 |
| DISCONNECT | 14 |

# 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.

## 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.

## Pin name property

The pin 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  
})

## 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);

## 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);

## IO Providers

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

## Sensors

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

## Displays

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

## Real-time clocks

The host provider instance should include a default Real-time clock constructor options object on its rtc property.

## Domain Name resolver

The host provider instance should include a default Domain Name Resolver class constructor options object on its network.dns.resolver property.

## NTP client

The host provider instance should include a default NTP Client class constructor options object on its network.ntp.client property.

## HTTP client

The host provider instance should include a default HTTP Client class constructor options object on its network.http.client property.

## HTTPS client

The host provider instance should include a default secure HTTP Client class constructor options object on its network.https.client property.

## HTTP server

The host provider instance should include a default HTTP Client class constructor options object on its network.http.server property.

## MQTT client

The host provider instance should include a default MQTT Client class constructor options object on its network.mqtt.client property.

## MQTTS client

The host provider instance should include a default secure MQTT Client class constructor options object on its network.mqtts.client property.

## WS (WebSocket) client

The host provider instance should include a default WebSocket Client class constructor options object on its network.ws.client property.

## WSS (WebSocket Secure) client

The host provider instance should include a default secure WebSocket Client class constructor options object on its network.wss.client property.

## TLS client

The host provider instance should include a default TLS Client class constructor options object on its network.tls.client property.

## Network Interfaces

The host provider instance should include a Network Interface class constructor options object for each of its network interfaces on its network.interface property.

const Ethernet0 = device.network.interface.Ethernet0;  
const eth0 = new Ethernet0.io(Ethernet0);

# 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.

### 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.

## **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. |

## **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. |

### 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](#alg-sensor-prov-class-pattern) of the Provenance Sensor Class Pattern.

1. (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.

* 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.

* + 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. Internal methods

Internal methods are implementation-dependent and must not be accessible outside the implementation. This document uses JavaScript private method syntax to indicate internal methods, prefixing the names of internal methods with #.

* 1. Ranges
     1. Booleans

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

* + 1. 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**.

* + 1. 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.

* + 1. Byte buffers

For byte buffer ranges, the value is checked to be an instance of ArrayBuffer, SharedArrayBuffer, Uint8Array, Int8Array or DataView.

Further restrictions can be specified, for instance on the byteLength.

To access the data contained in a byte buffer, algorithms uses a host specific operator:

GetBytePointer(*buffer*)

The operator throws if buffer is not an instance of ArrayBuffer, SharedArrayBuffer, Uint8Array, Int8Array, or DataView, or if buffer is detached. For a TypedArray and DataView instances, the pointer takes the view’s byte offset into account.

* + 1. 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.

* + 1. Asynchronous operations

Asynchronous operations are never synchronous: the callback is never invoked directly by the method that starts the asynchronous operation, but indirectly at the end of the asynchronous operation.

To emphasize such a rule, the algorithms uses steps like:

1. Queue a task that performs
   1. **Call**(this, *callback*)

The mechanism can be similar to what is necessary to implement setTimeout.

print(1);  
setTimeout(0, () => print(3));  
print(2);  
// 1 2 3

* 1. Base Class Pattern
     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
   * + 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.
  + 1. close()

1. **CheckInternalFields**(this)
2. Mark this as eligible for garbage collection
3. Cancel any pending callbacks for this
4. **ClearInternalFields**(this)
   * 1. close(callback)
5. **CheckInternalFields**(this)
6. Throw if *callback* is not undefined and not **IsCallable**(*callback*)
7. Optionally, cancel asynchronous operations
8. When all asynchronous operations succeeded or failed
   1. Mark this as eligible for garbage collection
   2. **ClearInternalFields**(this)
   3. If *callback* is not undefined
      1. Queue a task that performs
         1. **Call**(this, *callback*, null)
   4. IO Class Pattern
      1. **constructor**(options)
9. Execute steps 1 to 7 of the Base Class Pattern constructor
10. Let *value* be **GetProperty**(*params*, "format")
11. **SetInternalField**(this, "format", *value*)
12. 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*)
13. Catch *exception*
    1. **Call**(this, **GetProperty**(this, "close"))
    2. Throw *exception*
14. Execute step 8 of the Base Class Pattern constructor
    * 1. close()
15. Execute step 1 of the Base Class Pattern close method
16. Let *resources* be **GetInternalField**(this, "resources")
17. Return if *resources* is null
18. Execute steps 2 and 3 of the Base Class Pattern close method
19. Free *resources*
20. Execute step 4 of the Base Class Pattern close method
    * 1. read([*option*])
21. **CheckInternalFields**(this)
22. Let *resources* be **GetInternalField**(this, "resources")
23. Throw if *resources* is null
24. If *resources* is not readable
    1. return undefined
25. Let *format* be **GetInternalField**(this, "format")
26. 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*.
27. Throw if *option* is present
28. Read *data* from *resources*
29. Format *data* according to *format*
30. Return *data*.
    * 1. write(*data*)
31. **CheckInternalFields**(this)
32. Let *resources* be **GetInternalField**(this, "resources")
33. Throw if *resources* is null or not writable
34. Throw if *data* is absent
35. Let *format* be **GetInternalField**(this, "format")
36. 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
37. Throw if *data* is not formatted according to *format*
38. Write *data* into *resources*
    * 1. set format(*value*)
39. **CheckInternalFields**(this)
40. Throw if *value* is not in the valid range of "format"
41. **SetInternalField**(this, "format", *value*)
    * 1. get format()
42. **CheckInternalFields**(this)
43. Return **GetInternalField**(this, "format")
    * 1. Notes

* 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.
  1. IO Class Pattern – asynchronous
     1. **close**(callback)

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. Optionally, cancel asynchronous operations
5. When all asynchronous operations succeeded or failed
   1. Mark this as eligible for garbage collection
   2. **ClearInternalFields**(this)
   3. Free *resources*
   4. Execute step 5.2 and 5.3 of the Base Class Pattern close method
      1. read(option[, callback])
6. **CheckInternalFields**(this)
7. Let *resources* be **GetInternalField**(this, "resources")
8. Throw if *resources* is null or not readable
9. Throw if *option* is absent
10. If *option* is a number
    1. Throw if *option* is no positive integer
    2. Let *n* be *option*
    3. Let *data* be **Construct**("ArrayBuffer", *n*)
11. Else
    1. Let *data* be *option*
    2. Let *pointer* be **GetBytePointer**(*data*)
    3. Let *n* be **GetProperty**(*data*, "byteLength")
12. Throw if *callback* is not undefined and not **IsCallable**(*callback*)
13. Start an input operation to read *n* bytes into *data*
    1. When the input operation succeeded
       1. If *callback* is not undefined
          1. Queue a task that performs
             1. **Call**(this, *callback*, null, *data*, *n*)
    2. When the input operation failed
       1. If *callback* is not undefined
          1. Let *error* be a JavaScript Error object describing the failure
          2. Queue a task that performs
             1. **Call**(this, *callback*, *error*)
       2. write(data[, callback])
14. **CheckInternalFields**(this)
15. Let *resources* be **GetInternalField**(this, "resources")
16. Throw if *resources* is null or not writable
17. Throw if *data* is absent
18. Let *pointer* be **GetBytePointer**(*data*)
19. Let *n* be **GetProperty**(*data*, "byteLength")
20. Throw if *callback* is not undefined and not **IsCallable**(*callback*)
21. Start an output operation to write *n* bytes from *data*
    1. When the output operation succeeded
       1. If *callback* is not undefined
          1. Queue a task that performs
             1. **Call**(this, *callback*, null, *data*, *n*)
    2. When the output operation failed
       1. If *callback* is not undefined
          1. Let *error* be a JavaScript Error object describing the failure
          2. Queue a task that performs
             1. **Call**(this, *callback*, *error*)
       2. Notes

* The input and output operations represent the implementation dependent mechanism that ensures that asynchronous read and write operations happen in the order issued.
* Step 4 of the close method is optional since operations can be cancellable or not. Cancelled operations fail with a corresponding Error object.
* Step 6.2 of the read method and step 5 of the write method ensures *data* is a byte buffer.
  1. IO Classes
     1. Digital
        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.
  + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "number" | 0 or 1 | 0 or 1 |

* + 1. Digital bank
       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.
  + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "number" | 32-bit unsigned integer | 32-bit unsigned integer |

* + 1. Analog input
       1. constructor options

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

* + - 1. read / write data

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "number" | all |  |

* + 1. Pulse-width modulation
       1. constructor options

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

* + - 1. read / write data

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "number" |  | positive integer |

* + 1. I²C – synchronous IO
       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" |

* + - 1. read / write data

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "buffer" | ArrayBuffer | byte buffer |

* + - 1. read(*option*[, *stop*])

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *option* | yes\* | positive integer, byte buffer |  |
| *stop* | no | true or false | true |

* The number of readable bytes is undefined so *option* is required
  + - 1. write(*data*[, *stop*])

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *data* | yes | byte buffer |  |
| *stop* | no | true or false | true |

* + 1. I²C – asynchronous IO
       1. read(option[, stop][, callback])

1. Execute steps 1 to 7 of the IO.Async Class Pattern read method
2. If *callback* is not undefined
   1. Throw if not **IsCallable**(*callback*)
   2. Convert *stop* to a JavaScript boolean
3. Else if *stop* is not undefined
   1. If **IsCallable**(*stop*)
      1. Let *callback* be *stop*
      2. Let *stop* be **true**
   2. Else
      1. Convert *stop* to a JavaScript boolean
4. Else
   1. Let *stop* be **true**
5. Execute step 8 of the IO.Async Class Pattern read method
   * + 1. write(data[, stop][, callback])
6. Execute steps 1 to 6 of the IO.Async Class Pattern write method
7. If *callback* is not undefined
   1. Throw if not **IsCallable**(*callback*)
   2. Convert *stop* to a JavaScript boolean
8. Else if *stop* is not undefined
   1. If **IsCallable**(*stop*)
      1. Let *callback* be *stop*
      2. Let *stop* be **true**
   2. Else
      1. Convert *stop* to a JavaScript boolean
9. Else
   1. Let *stop* be **true**
10. Execute step 8 of the IO.Async Class Pattern write method
    * + 1. Notes

* The read and write methods algorithms describe how to handle an optional argument before the optional *callback* argument.
  + 1. System management bus (SMBus) – synchronous IO
       1. constructor *options*

All properties from I²C plus the following:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| stop | no | true or false | false |

* + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "buffer" | any | any |

* + - 1. read(*option*)

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *option* | yes\* | positive integer, byte buffer |  |

* The number of readable bytes is undefined so *option* is required
  + - 1. readUint8(*register*)

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer |  |

* + - 1. writeUint8(*register*, *value*)

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

* + - 1. readUint16(register, bigEndian)

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

* + - 1. writeUint16(*register*, *value*)

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

* + - 1. readBuffer(*register*, *buffer*)

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer |  |
| *buffer* | yes | byte buffer |  |

* + - 1. writeBuffer(*register*, *buffer*)

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer |  |
| *buffer* | yes | byte buffer |  |

* + 1. **System management bus (SMBus) – asynchronous IO**

All properties from I²C.Async plus the following:

* + - 1. readUint8(register[, callback])

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer | N/A |
| *callback* | no | Function | null |

* + - 1. writeUint8(register, value[, callback])

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer |  |
| *value* | yes | 8-bit unsigned integer | N/A |
| *callback* | no | Function | null |

* + - 1. readUint16(register[, bigEndian][, callback])

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer | N/A |
| *bigEndian* | no | true or false | false |
| *callback* | no | Function | null |

* + - 1. writeUint16(register, value[, bigEndian][, callback])

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer | N/A |
| *value* | yes | 16-bit unsigned integer | N/A |
| *callback* | no | Function | null |

* + - 1. readBuffer(register, option[, callback])

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer | N/A |
| *buffer* | yes | number or byte buffer | N/A |
| *callback* | no | Function | null |

* + - 1. writeBuffer(register, buffer[, callback])

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *register* | yes | integer | N/A |
| *buffer* | yes | byte buffer | N/A |
| *callback* | no | Function | null |

* + - 1. Notes
* The asynchronous methods to read and write data behaves analogously to the I2C.Async read and write method.
  + 1. Serial
       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.
  + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "number" | 8-bit unsigned integer | 8-bit unsigned integer |
| "buffer" | ArrayBuffer | byte buffer |

* + - 1. 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
      * 1. set(options)
7. **CheckInternalFields**(this)
8. Throw if *options* is not an object
9. 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
10. 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
11. 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
       * 1. get([*options*])
12. **CheckInternalFields**(this)
13. If *options* is absent
    1. Let *result* be an empty object
14. Else
    1. Throw if *options* is not an object
    2. Let *result* be *options*
15. If serial connection’s CTS pin is set
    1. **SetProperty**(*result*, "clearToSend", true)
16. Else
    1. **SetProperty**(*result*, "clearToSend", false)
17. If serial connection’s DSR pin is set
    1. **SetProperty**(*result*, "dataSetReady", true)
18. Else
    1. **SetProperty**(*result*, "dataSetReady", false)
19. Return *result*
    * 1. Serial Peripheral Interface (SPI)
         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" |

* + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "buffer" | ArrayBuffer | byte buffer |

* + - 1. read(*option*)

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *option* | yes\* | positive integer, byte buffer |  |

* The number of readable bytes is undefined so *option* is required
  + - 1. 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*
   * + 1. flush([*deselect*])
8. **CheckInternalFields**(this)
9. Flush all written but unsent data
10. 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
       3. Pulse count
          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" |

* + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "number" | integer | integer |

* + 1. TCP socket
       1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| address | 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" |

* + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "buffer" | ArrayBuffer | byte buffer |
| "number" | 8-bit unsigned integer | 8-bit unsigned integer |

* + 1. 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" |

* + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "socket/tcp" | instance of TCP Socket |  |

* + - 1. UDP socket
    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" |

* + - 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "buffer" | ArrayBuffer | byte buffer |

* + - 1. write(data, address, port)

|  |  |  |  |
| --- | --- | --- | --- |
| **Param** | **Required** | **Range** | **Default** |
| *data* | yes | byte buffer |  |
| *address* | yes | string |  |
| *port* | yes | 16-bit unsigned integer |  |

* 1. Peripheral Class Pattern
     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
   * 1. close()
5. Execute step 1 of the Base Class Pattern close method
6. Let *status* be **GetInternalField**(this, "status")
7. Return if *status* is null
8. Execute steps 2 and 3 of the Base Class Pattern close method
9. Deactivate the peripheral
10. 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")
11. Execute step 4 of the Base Class Pattern close method
    * 1. configure(*options*)
12. **CheckInternalFields**(this)
13. Let *status* be **GetInternalField**(this, "status")
14. Throw if *status* is null
15. Throw if *options* is undefined or null
16. 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
17. Configure the peripheral with *options*
    * 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.
  1. Sensor Class Pattern
     1. constructor(options)

1. Execute all steps of the Peripheral Class Pattern constructor
   * 1. close()
2. Execute all steps of the Peripheral Class Pattern close method
   * 1. configure(*options*)
3. Execute all steps of the Peripheral Class Pattern configure method
   * 1. sample([*params*])
4. **CheckInternalFields**(this)
5. Let *status* be **GetInternalField**(this, "status")
6. Throw if *status* is null
7. Throw if *params* are absent but required, or present but not in the valid range
8. 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*)
9. Else
   1. Let *result* be undefined
10. Return *result*.
    * 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.
  1. Sensor Classes
     1. Accelerometer
        1. sample *params*:

None

* + - 1. 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 |

* + 1. Ambient light
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| illuminance | yes | positive number | ambient light level in lux |

* + 1. Atmospheric pressure
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| pressure | yes | number | atmospheric pressure in Pascal |

* + 1. Carbon Dioxide
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| CO2 | yes | number | carbon dioxide in parts per million |

* + 1. Carbon Monoxide
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| CO | yes | number | carbon monoxide in parts per million |

* + 1. Dust
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| dust | yes | number | dust levels in micrograms per cubic meter |

* + 1. Gyroscope
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| x | yes | number | angular velocity around the x axis in radian per second |
| y | yes | number | angular velocity around the y axis in radian per second |
| z | yes | number | angular velocity around the z axis in radian per second |

* + 1. Humidity
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| humidity | yes | number from 0 to 1 | relative humidity as a percentage |

* + 1. Hydrogen
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| H | yes | number | hydrogen in parts per million |

* + 1. Hydrogen Sulfide
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| H2S | yes | number | hydrogen sulfide in parts per million |

* + 1. Magnetometer
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| x | yes | number | magnetic field around the x axis in microtesla |
| y | yes | number | magnetic field around the y axis in microtesla |
| z | yes | number | magnetic field around the z axis in microtesla |

* + 1. Methane
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| CH4 | yes | number | methane in parts per million |

* + 1. Nitric Oxide
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| NO | yes | number | nitric oxide in parts per million |

* + 1. Nitric Dioxide
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| NO2 | yes | number | nitric dioxide in parts per million |

* + 1. Oxygen
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| O | yes | number | oxygen in parts per million |

* + 1. Particulate Matter
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| particulateMatter | yes | number | particulate matter levels in micrograms per cubic meter |

* + 1. Proximity
       1. sample *params*:

None

* + - 1. 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 |

* + 1. Soil Moisture
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| moisture | yes | number between 0 and 1 | relative soil moisture level |

* + 1. Temperature
       1. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| temperature | yes | number | temperature in degrees Celsius |

* + 1. Touch
       1. sample *params*:

None

* + - 1. sample *result*:

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

* + - 1. 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 |

* + - 1. Volatile Organic Compounds
      2. sample *params*:

None

* + - 1. sample *result*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Description** |
| tvoc | yes | number | total volatile organic compounds in parts per billion |

* 1. Display Class Pattern
     1. constructor(*options*)

1. Execute all steps of the Peripheral Class Pattern constructor
   * 1. adaptInvalid(*area*)
2. **CheckInternalFields**(this)
3. Throw if *area* is absent
4. If **HasProperty**(*area*, "x")
   1. Let *x* be **GetProperty**(*area*, "x")
5. Else
   1. Let *x* be 0
6. If **HasProperty**(*area*, "y")
   1. Let *y* be **GetProperty**(*area*, "y")
7. Else
   1. Let *y* be 0
8. If **HasProperty**(*area*, "width")
   1. Let *width* be **GetProperty**(*area*, "width")
9. Else
   1. Let *width* be the width of the frame buffer in pixels
10. If **HasProperty**(*area*, "height")
    1. Let *height* be **GetProperty**(*area*, "height")
11. Else
    1. Let *height* be the height of the frame buffer in pixels
12. Adjust *x*, *y*, *width*, *height* to define a valid area to update
13. **SetProperty**(*area*, "x", *x*)
14. **SetProperty**(*area*, "y", *y*)
15. **SetProperty**(*area*, "width", *width*)
16. **SetProperty**(*area*, "height", *height*)
    * 1. close()
17. Execute all steps of the Peripheral Class Pattern close method
    * 1. begin(*options*)
18. **CheckInternalFields**(this)
19. Let *status* be **GetInternalField**(this, "status")
20. Throw if *status* is null
21. Let *x* be 0
22. Let *y* be 0
23. Let *width* be the width of the frame buffer in pixels
24. Let *height* be the height of the frame buffer in pixels
25. Let *continue* be false
26. 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")
27. Throw if the area defined by *x*, *y*, *width*, and *height* is invalid.
28. If *status* is ready
    1. **SetInternalField**(this, "status", "updating")
29. Else
    1. Throw if *continue* is false
30. Use *x*, *y*, *width*, *height* to prepare the frame buffer to receive scanlines
    * 1. configure(*options*)
31. Execute all steps of the Peripheral Class Pattern configure method
    * 1. end()
32. **CheckInternalFields**(this)
33. Let *status* be **GetInternalField**(this, "status")
34. Throw if *status* is not "updating"
35. **SetInternalField**(this, "status", "finishing")
36. Make updated frame buffer visible
37. **SetInternalField**(this, "status", "ready")
    * 1. send(scanlines)
38. **CheckInternalFields**(this)
39. Let *status* be **GetInternalField**(this, "status")
40. Throw if *status* is not "updating"
41. Throw if *scanlines* is absent
42. Let *pointer* be **GetBytePointer**(*scanlines*)
43. Let *n* be **GetProperty**(*lines*, "byteLength")
44. Transfer *n* bytes from *pointer* to the frame buffer
    * 1. get width()
45. **CheckInternalFields**(this)
46. Return the width of the frame buffer in pixels
    * 1. get height()
47. **CheckInternalFields**(this)
48. Return the height of the frame buffer in pixels
    * 1. Notes

* 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.
  + 1. 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” |  |

* 1. Real-Time Clock Class Pattern
     1. constructor(*options*)

1. Execute step 1 of the Peripheral Class Pattern constructor
2. Let *interrupt* be **GetInternalField**(this, "interrupt")
3. Let *onAlarm* be **GetInternalField**(this, "onAlarm")
4. If *interrupt* is not null and *onAlarm* is not null
   1. Let *interruptParams* be **GetProperty**(*params*, "interrupt")
   2. Let *onReadable* be a function with the following steps:
      1. Queue a task that performs
         1. **Call**(this, *onAlarm*)
   3. **SetProperty**(*interruptParams*, "onReadable", *onReadable*)
5. Execute steps 2 to 4 of the Peripheral Class Pattern constructor
   * 1. close()
6. Execute all steps of the Peripheral Class Pattern close method
   * 1. configure(*options*)
7. Execute all steps of the Peripheral Class Pattern configure method
   * 1. get time()
8. **CheckInternalFields**(this)
9. Let *status* be **GetInternalField**(this, "status")
10. Throw if *status* is null
11. If the peripheral is readable
    1. Let *result* be the clock time as a JavaScript number
12. Else
    1. Let *result* be undefined
13. Return *result*.
    * 1. set time(*time*)
14. **CheckInternalFields**(this)
15. Let *status* be **GetInternalField**(this, "status")
16. Throw if *status* is null
17. If the peripheral is writable
    1. Convert *time* into a JavaScript number
    2. Set the clock time to *time*
       1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| clock | yes | Object |  |
| interrupt | no | null or Object | null |
| onAlarm | no | null or Function | null |

* + 1. configure *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| alarm | no | number | 0 |

* 1. Network Interface Class Pattern
     1. constructor(*options*)

1. Execute all steps of the Base Class Pattern constructor
   * 1. close()
2. Execute all steps of the Base Class Pattern close method
   * 1. connect(*options*)
3. **CheckInternalFields**(this)
4. Let *connection* be **GetInternalField**(this, "connection")
5. Throw if *connection* is not **0**
6. **SetInternalField**(this, "connection", 100)
7. Let *port* be **GetInternalField**(this, "port")
8. Let *onChanged* be **GetInternalField**(this, "onChanged")
9. Monitor the network interface specified by *port*
   1. When changed
      1. If *onChanged* is not null
         1. Queue a task that performs
            1. **Call**(this, *onChanged*)
      2. disconnect()
10. **CheckInternalFields**(this)
11. Let *connection* be **GetInternalField**(this, "connection")
12. If *connection* is not **0**
    1. Disconnect the network interface
       1. get MAC()
13. **CheckInternalFields**(this)
14. Let *connection* be **GetInternalField**(this, "connection")
15. If *connection* is more than **0**
    1. Let *result* be the MAC address of the network interface as a JavaScript string
16. Else
    1. Let *result* be undefined
17. Return *result*.
    * 1. get address()
18. **CheckInternalFields**(this)
19. Let *connection* be **GetInternalField**(this, "connection")
20. If *connection* is more than or equal to **500**
    1. Let *result* be the IP address of the network interface as a JavaScript string
21. Else
    1. Let *result* be undefined
22. Return *result*.
    * 1. get connection()
23. **CheckInternalFields**(this)
24. Let *connection* be **GetInternalField**(this, "connection")
25. return *connection*
    * 1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| onChanged | no | null or Function | null |
| port | no | string |  |

* 1. Ethernet Network Interface
     1. connect(*options*)

1. Execute steps 1 to 6 of the Network Interface Class Pattern connect method
2. Start connecting the network interface specified by *port*
3. Execute step 7 of the Network Interface Class Pattern connect method
   1. Wi-Fi Network Interface
      1. connect(*options*)
4. Execute steps 1 to 6 of the Network Interface Class Pattern connect method
5. Throw if *options* is not an object
6. If **HasProperty**(*options*, "SSID")
   1. Let *SSID* be **GetProperty**(*options*, "SSID")
   2. Convert *SSID* into a JavaScript string
7. Else
   1. Let *SSID* be undefined
8. If **HasProperty**(*options*, "BSSID")
   1. Let *BSSID* be **GetProperty**(*options*, "BSSID")
   2. Convert *BSSID* into a JavaScript string
9. Else
   1. Let *BSSID* be undefined
10. Throw if both *SSID* and *BSSID* are undefined
11. If **HasProperty**(*options*, "channel")
    1. Let *channel* be **GetProperty**(*options*, "channel")
    2. Convert *channel* into a JavaScript number
12. Else
    1. Let *channel* be undefined
13. If **HasProperty**(*options*, "secure")
    1. Let *secure* be **GetProperty**(*options*, "secure")
    2. Convert *secure* into a JavaScript boolean
14. Else
    1. Let *secure* be **false**
15. If **HasProperty**(*options*, "password")
    1. Let *password* be **GetProperty**(*options*, "password")
    2. Convert *password* into a JavaScript string
16. Else
    1. Let *password* be undefined
17. Start connecting the network interface specified by *port* to the access point specified by *SSID*, *BSSID*, *channel* and *secure* with *password*
18. Execute step 7 of the Network Interface Class Pattern connect method
    * 1. scan(options)
19. **CheckInternalFields**(this)
20. Let *scanning* be **GetInternalField**(this, "scanning")
21. Throw if *scanning* is **true**
22. Throw if *options* is not an object
23. Let *onFound* be **GetProperty**(*options*, "onFound")
24. Throw if not **IsCallable**(*onFound* )
25. If **HasProperty**(*options*, "onComplete")
    1. Let *onComplete* be **GetProperty**(*options*, "onComplete")
    2. Throw if not **IsCallable**(*onComplete*)
26. Else
    1. Let *onComplete* be undefined
27. If **HasProperty**(*options*, "channel")
    1. Let *channel* be **GetProperty**(*options*, "channel")
    2. Convert *channel* into a JavaScript number
28. Else
    1. Let *channel* be undefined
29. If **HasProperty**(*options*, "frequency")
    1. Let *frequency* be **GetProperty**(*options*, "frequency")
    2. Convert *frequency* into a JavaScript number
    3. Throw if *frequency* is neither **2.4** nor **5**
30. Else
    1. Let *frequency* be undefined
31. If **HasProperty**(*options*, "secure")
    1. Let *secure* be **GetProperty**(*options*, "secure")
    2. Convert *secure* into a JavaScript boolean
32. Else
    1. Let *secure* be **false**
33. **SetInternalField**(this, "scanning", **true**)
34. Start scanning for access points matching *channel*, *frequency* and *secure*
    1. When an access point is found
       1. Let *result* be an empty object
       2. Let *value* be the SSID of the access point as a JavaScript string
       3. **SetProperty**(*result*, "SSID", *value*)
       4. Let *value* be the BSSID of the access point as a MAC address JavaScript string
       5. **SetProperty**(*result*, "BSSID", *value*)
       6. Let *value* be the RSSI of the access point as a JavaScript number
       7. **SetProperty**(*result*, "RSSI", *value*)
       8. Let *value* be the channel of the access point as a JavaScript number
       9. **SetProperty**(*result*, "channel", *value*)
       10. Let *security* be the security mode of the access point as a JavaScript string
       11. **SetProperty**(*security*, "security", *value*)
       12. Queue a task that performs
           1. **Call**(this, *onFound*, null, *result*)
    2. When done
       1. **SetInternalField**(this, "scanning", **false**)
       2. If *onComplete* is not undefined
          1. Queue a task that performs
             1. **Call**(this, *onComplete*)
       3. get BSSID()
35. **CheckInternalFields**(this)
36. Let *connection* be **GetInternalField**(this, "connection")
37. If *connection* is more than or equal to **400**
    1. Let *result* be the BSSID of the access point as a JavaScript string
38. Else
    1. Let *result* be undefined
39. Return *result*.
    * 1. get RSSI()
40. **CheckInternalFields**(this)
41. Let *connection* be **GetInternalField**(this, "connection")
42. If *connection* is more than or equal to **400**
    1. Let *result* be the RSSI of the access point as a JavaScript string
43. Else
    1. Let *result* be undefined
44. Return *result*.
    * 1. get SSID()
45. **CheckInternalFields**(this)
46. Let *connection* be **GetInternalField**(this, "connection")
47. If *connection* is more than or equal to **400**
    1. Let *result* be the SSID of the access point as a JavaScript string
48. Else
    1. Let *result* be undefined
49. Return *result*.
    * 1. get channel()
50. **CheckInternalFields**(this)
51. Let *connection* be **GetInternalField**(this, "connection")
52. If *connection* is more than or equal to **400**
    1. Let *result* be the channel of the access point as a JavaScript number
53. Else
    1. Let *result* be undefined
54. Return *result*.
    1. Domain Name Resolver Class Pattern
       1. constructor(*options*)
55. Execute all steps of the Base Class Pattern constructor
    * 1. close()
56. Execute all steps of the Base Class Pattern close method
    * 1. resolve(options[, callback])
57. **CheckInternalFields**(this)
58. Throw if *options* is not an object
59. If **HasProperty**(*options*, "host")
    1. Let *name* be **GetProperty**(*options*, "host")
    2. Convert *name* to a JavaScript string
60. Else
    1. Throw
61. Throw if *callback* is not undefined and not **IsCallable**(*callback*)
62. If *name* matches an IP address
    1. If *callback* is not undefined
       1. Queue a task that performs
          1. **Call**(this, *callback*, null, *name*, *name*)
63. Else
    1. Start the resolution with *name*
       1. When the resolution succeeded
          1. If *callback* is not undefined
             1. Let *address* be the resolved address as a JavaScript string
             2. Queue a task that performs

**Call**(this, *callback*, null, *name*, *address*)

* + 1. When the resolution failed
       1. If *callback* is not undefined
          1. Let *error* be a JavaScript Error object describing the failure
          2. Queue a task that performs

**Call**(this, *callback*, *error*)

* 1. DNS over UDP
     1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| socket | yes | Object | N/A |
| servers | yes | Array of strings | N/A |

* + 1. Notes
* The resolution itself can be implemented in JavaScript. See the [sample code](https://github.com/Moddable-OpenSource/moddable/blob/public/examples/io/udp/dns/dns.js)
  1. DNS over HTTPS
     1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| http | yes | Object | N/A |
| servers | yes | Array of string | N/A |

* + 1. Notes
* The resolution itself can be implemented in JavaScript.
  1. NTP Client
     1. constructor(*options*)

1. Execute all steps of the Base Class Pattern constructor
   * 1. close()
2. Execute all steps of the Base Class Pattern close method
   * 1. getTime(*callback*)
3. **CheckInternalFields**(this)
4. Let *synchronizing* be **GetInternalField**(this, "synchronizing")
5. Throw if *synchronizing* is **true**
6. Throw if not **IsCallable**(*callback*)
7. **SetInternalField**(this, "synchronizing", **true**)
8. Start the synchronization
   1. When the synchronization succeeded
      1. Let *time* be the synchronized time as a JavaScript number
      2. Queue a task that performs
         1. **Call**(this, *callback*, null, *time*)
         2. **SetInternalField**(this, "synchronizing", **false**)
   2. When the synchronization failed
      1. Let *error* be a JavaScript Error object describing the failure
      2. Queue a task that performs
         1. **Call**(this, *callback*, *error*)
         2. **SetInternalField**(this, "synchronizing", **false**)
      3. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| socket | yes | Object | N/A |
| servers | yes | Array of string | N/A |

* + 1. Notes
* The synchronization itself can be implemented in JavaScript. See the [sample code](https://github.com/Moddable-OpenSource/moddable/blob/public/examples/io/udp/sntp/sntp.js)
  1. TCP Client Class Pattern
     1. constructor(*options*)

1. Execute all steps of the Base Class Pattern constructor
2. Let *dnsOptions* be **GetInternalField**(this, "dns")
3. Let *dnsConstructor* be **Get**(*dnsOptions*, "io")
4. Let *dnsParams* be a copy of *dnsOptions*
5. **Set**(*dnsParams*, "target", this)
6. Let *dnsResolver* be **New**(*dnsConstructor*, *dnsParams*)
7. **SetInternalField**(*target*, "dnsResolver", *dnsResolver*)
8. Let *resolve* be **Get**(*dnsResolver*, "resolve")
9. Let *resolveParams* be a new object
10. **Set**(*resolveParams*, "host", **GetInternalField**(this, "host"))
11. Let *resolveCallback* be **GetInternalField**(this, "resolveCallback")
12. **Call**(*dnsResolver*, *resolve*, *resolveParams*, *resolveCallback*)
    * 1. close()
13. Let *tcpSocket* be **GetInternalField**(this, "tcpSocket")
14. If *tcpSocket* is not null
    1. **Call**(*tcpSocket*, **Get**(*tcpSocket*, "close"))
15. Let *dnsResolver* be **GetInternalField**(this, "dnsResolver")
16. If *dnsResolver* is not null
    1. **Call**(*dnsResolver*, **Get**(*dnsResolver*, "close"))
17. Execute all steps of the Base Class Pattern close method
    * 1. #resolveCallback(*error*, *name*, *address*)
18. Let *target* be **Get**(this, "target")
19. **Call**(this, **Get**(this, "close"))
20. **SetInternalField**(*target*, "dnsResolver", null)
21. If *error* is null
    1. Let *tcpOptions* be **GetInternalField**(*target*, "socket")
    2. Let *tcpConstructor* be **Get**(*tcpOptions*, "io")
    3. Let *tcpParams* be a copy of *tcpOptions*
    4. **Set**(*tcpParams*, "address", *address*)
    5. **Set**(*tcpParams*, "port", **GetInternalField**(*target*, "port"))
    6. **Set**(*tcpParams*, "onError", **GetInternalField**(this, #tcpError))
    7. **Set**(*tcpParams*, "onReadable", **GetInternalField**(this, #tcpReadable))
    8. **Set**(*tcpParams*, "onWritable", **GetInternalField**(this, #tcpWritable))
    9. **Set**(*tcpParams*, "target", this)
    10. Let *tcpSocket* be **New**(*tcpConstructor*, *tcpParams*)
    11. **SetInternalField**(*target*, "tcpSocket", *tcpSocket*)
22. Else
    1. Let *onError* be **GetInternalField**(*target*, "onError")
    2. If *onError* is not null
       1. Queue a task that performs
          1. **Call**(*target*, *onError*, *error*)
       2. read(count)
       3. write(*data*[,options])
       4. #tcpError(*error*)
       5. #tcpReadable(*count*)
       6. #tcpWritable(*count*)
       7. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "buffer" | ArrayBuffer | byte buffer |

* + 1. Notes
* The read, write, #tcpError, #tcpReadable, #tcpWritable functions implement the network protocol, which usually requires a state machine, buffers, parsers, serializers, etc.
* Such methods can read and write from the TCP socket and can queue tasks to call the client callbacks.
* For each network protocol, the client has specific methods and callbacks, and the write method can have specific options.
  1. HTTP Client
     1. constructor(*options*)

1. Execute step 1 of the TCP Client Class Pattern constructor
2. Let *requests* be a new Array object
3. **SetInternalField**(this, "requests", *requests*)
4. Execute steps 2 to 12 of the TCP Client Class Pattern constructor
   * 1. close()
5. Let *requests* be **GetInternalField**(this, "requests")
6. For each *request* of *requests*
   1. Cancel *request*
7. Execute all steps TCP Client Class Pattern close method
   * 1. request(*options*)
8. Let *requests* be **GetInternalField**(this, "requests")
9. Let *requestConstructor* be the HTTP Client Request constructor
10. Let *requestParams* be a copy of *options*
11. **Set**(*requestParams*, "target", this)
12. Let *request* be **New**(*requestConstructor*, *requestParams*)
13. Add *request* to *request*
14. When this is ready
    1. Start the request
       1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| dns | yes | Object | N/A |
| host | yes | string | N/A |
| socket | yes | Object | N/A |
| port | no | number | 80 |
| onError | no | null or Function | null |

* + 1. Notes
* The HTTP Client Request constructor is available only to the HTTP Client class.
* The HTTP Client class conforms to the TCP Client Class Pattern here above except:
  + The read and write methods are provided by the HTTP Client Request instance.
  + The HTTP Client Request instance owns the network protocol specific callbacks.
* If the HTTP Client handles a single request at time, step 7 of the request method waits for the former request to complete.
* For details about the implementation of the HTTP Client, see the [sample code](https://github.com/Moddable-OpenSource/moddable/blob/public/examples/io/tcp/httpclient/httpclient.js)
  1. HTTP Client Request
     1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| method | no | string | GET |
| path | no | string | / |
| headers | no | Map | null |
| port | no | number | 80 |
| onHeaders | no | Function | null |
| onReadable | no | Function | null |
| onWritable | no | Function | null |
| onDone | no | Function | null |

* + 1. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "buffer" | ArrayBuffer | byte buffer |

* 1. MQTT Client
     1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| dns | yes | Object | N/A |
| host | yes | string | N/A |
| socket | yes | Object | N/A |
| port | no | number | 1883 |
| id | no | string |  |
| user | no | string |  |
| password | no | string or Byte Buffer |  |
| keepAlive | no | number | 0 |
| clean | no | boolean | true |
| will | no | Object\* | null |
| onReadable | no | Function | null |
| onWritable | no | Function | null |
| onError | no | Function | null |
| onControl | no | Function | null |

* The will object has:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| topic | yes | string | N/A |
| message | yes | string or Byte Buffer | N/A |
| QoS | no | 0, 1, or 2 | 0 |
| retain | no | boolean | false |

* + 1. write options

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| operation | no | number | MQTTCLient.PUBLISH |
| id | no | number |  |
| topic | yes\* | string | N/A |
| QoS | no\* | 0, 1, or 2 | 0 |
| retain | no\* | boolean | false |
| duplicate | no\* | boolean | false |
| byteLength | no\* | number | data.byteLength |
| items | yes\* | Array | N/A |

* topic is required when operation is MQTTCLient.PUBLISH
* QoS, retain, duplicate, byteLength are used when operation is MQTTCLient.PUBLISH
* items is required and used when operation is MQTTCLient.SUBSCRIBE or MQTTCLient.UNSUBSCRIBE.
* items is an array of objects that have:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| topic | yes | string | N/A |
| QoS | no\* | 0, 1, or 2 | 0 |

* QoS is used when operation is MQTTCLient.SUBSCRIBE
  + 1. Notes
* The MQTT Client class conforms to the TCP Client Class Pattern here above.
* For details about the implementation of the MQTT Client, see the [sample code](https://github.com/Moddable-OpenSource/moddable/blob/public/examples/io/tcp/mqttclient/mqttclient.js)
  1. WebSocket Client
     1. constructor(*options*)

1. Execute step 1 of the TCP Client Class Pattern constructor
2. Let *tcpSocket* be **GetInternalField**(this, "attach")
3. If *tcpSocket* is null
   1. Execute steps 2 to 12 of the TCP Client Class Pattern constructor
4. Else
   1. **SetInternalField**(this, "tcpSocket", *tcpSocket*)
      1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| attach | no | instance of TCP Socket | null |
| dns | yes\* | Object | N/A |
| host | yes\* | string | N/A |
| socket | yes\* | Object | N/A |
| port | no\* | number | 80 |
| protocol | no\* | string | “” |
| headers | no\* | Map | null |
| onReadable | no | Function | null |
| onWritable | no | Function | null |
| onError | no | Function | null |
| onControl | no | Function | null |
| onClose | no | Function | null |

* If attach is present, dns, host, socket, port, protocol and headers are neither required nor used.
  + 1. write options

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| binary | no | boolean | **true** |
| more | no | boolean | **false** |
| opcode | no | number |  |

* + 1. Notes
* The WebSocket Client class conforms to the TCP Client Class Pattern here above.
* For details about the implementation of the WebSocket Client, see the [sample code](https://github.com/Moddable-OpenSource/moddable/blob/public/examples/io/tcp/websocketclient/websocketclient.js)
  1. TCP Server Class Pattern
     1. constructor(*options*)

1. Execute all steps of the Base Class Pattern constructor
2. Let *connections* be a new Set object
3. **SetInternalField**(this, "connections", *connections*)
4. Let *tcpOptions* be **GetInternalField**(*target*, "listener")
5. Let *tcpConstructor* be **Get**(*tcpOptions*, "io")
6. Let *tcpParams* be a copy of *tcpOptions*
7. **Set**(*tcpParams*, "port", **GetInternalField**(*target*, "port"))
8. **Set**(*tcpParams*, "onReadable", **GetInternalField**(this, #tcpReadable))
9. **Set**(*tcpParams*, "target", this)
10. Let *tcpSocket* be **New**(*tcpConstructor*, *tcpParams*)
11. **SetInternalField**(this, "tcpSocket", *tcpSocket*)
    * 1. close()
12. Let *connections* be **GetInternalField**(this, "connections")
13. For each *connection* of *connections*
    1. **Call**(*connection*, "close")
14. Let *tcpSocket* be **GetInternalField**(this, "tcpSocket")
15. If *tcpSocket* is not null
    1. **Call**(*tcpSocket*, "close")
16. Execute all steps of the Base Class Pattern close method
    * 1. #tcpReadable(*count*)
17. Let *target* be **Get**(this, "target")
18. Let *connections* be **GetInternalField**(*target*, "connections")
19. Let *onConnect* be **GetInternalField**(*target*, "onConnect")
20. Let *connectionConstructor* be a class conforming to the TCP Server Connection Class Pattern
21. While *count* > 0
    1. Let *from* be **Call**(this, **Get**(this, "read"))
    2. Let *connection* be **New**(*connectionConstructor*, *target*, *from*)
    3. Add *connection* to *connections*
    4. Queue a task that performs
       1. **Call**(*target*, *onConnect*, *connection*)
    5. Let *count* be *count* - 1
       1. constructor *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| listener | yes | Object | N/A |
| port | no\* | number | 80 |
| onConnect | yes | Function | N/A |

* + 1. Notes
* The connection constructor is specific to each network protocol, and available only to the implementation: a static private field of the server class, a closure of the server module, etc.
  1. TCP Server Connection Class Pattern
     1. constructor(*server*, *from*)

1. **SetInternalField**(this, "server", *server*)
2. Let *tcpConstructor* be **Get**(*from*, "constructor")
3. Let *tcpParams* be **New**("Object")
4. **Set**(*tcpParams*, "from", *from*)
5. **Set**(*tcpParams*, "onError", **GetInternalField**(this, #tcpError))
6. **Set**(*tcpParams*, "onReadable", **GetInternalField**(this, #tcpReadable))
7. **Set**(*tcpParams*, "onWritable", **GetInternalField**(this, #tcpWritable))
8. **Set**(*tcpParams*, "target", this)
9. Let *tcpSocket* be **New**(*tcpConstructor*, *tcpParams*)
10. **SetInternalField**(this, "tcpSocket", *tcpSocket*)
    * 1. close()
11. Let *tcpSocket* be **GetInternalField**(this, "tcpSocket")
12. If *tcpSocket* is not null
    1. **Call**(*tcpSocket*, **Get**(*tcpSocket*, "close"))
13. Let *server* be **GetInternalField**(this, "server")
14. Let *connections* be **GetInternalField**(*server*, "connections")
15. Remove this from *connections*
    * 1. read(*count*)
      2. write(*data*[, options])
      3. #tcpError(*error*)
      4. #tcpReadable(*count*)
      5. #tcpWritable(*count*)
      6. read / write *data*

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Write** |
| "buffer" | ArrayBuffer | byte buffer |

* + 1. Notes
* The read, write, #tcpError, #tcpReadable, #tcpWritable functions implement the network protocol, which usually requires a state machine, buffers, parsers, serializers, etc.
* Such methods can read and write from the TCP socket and can queue tasks to call the server connection callbacks.
* For each network protocol, the server connection has specific methods and callbacks, and the write method can have specific options.
  1. HTTP Server
     1. Notes
* The HTTP Server class conforms to the TCP Server Class Pattern here above.
* The server connection constructor is the HTTP Server Connection class.
* For details about the implementation of the HTTP Server, see the [sample code](https://github.com/Moddable-OpenSource/moddable/blob/public/examples/io/listener/httpserver/httpserver.js)
  1. HTTP Server Connection
     1. detach()

1. Let *tcpSocket* be **GetInternalField**(this, "tcpSocket")
2. **SetInternalField**(this, "tcpSocket", null)
3. Let *server* be **GetInternalField**(this, "server")
4. Let *connections* be **GetInternalField**(*server*, "connections")
5. Remove this from *connections*
6. return *tcpSocket*
   * 1. accept(*options*)
7. Throw if options is not an object
8. For each supported callback
   1. Let *name* be the name of the supported callback
   2. Let *callback* be **GetProperty**(*options*, *name*)
   3. If *callback* is not undefined
      1. Throw if not **IsCallable**(*callback*)
      2. **SetInternalField**(this, *name*, *callback*)
9. Start receiving the request
   * 1. get route()
10. Let *result* be **GetInternalField**(this, "route")
11. Return result
    * 1. set route(*options*)
12. Execute steps 1 and 2 of the accept method
13. **SetInternalField**(this, "route", *options*)
    * + 1. accept and set route *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| onDone | no | Function | null |
| onError | no | Function | null |
| onReadable | no | Function | null |
| onRequest | no | Function | null |
| onWritable | no | Function | null |

* + 1. respond(*options*)

1. Throw if options is not an object
2. Let *status* be **GetProperty**(*options*, "status")
3. Convert *status* into a JavaScript number
4. Throw if *status* is no positive integer
5. Let *headers* be **GetProperty**(*options*, "headers")
6. Throw if *headers* is no Map instance
7. Start sending the response with *status* and *headers*
   * + 1. respond *options*

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Required** | **Range** | **Default** |
| status | yes | positive integer | N/A |
| headers | yes | Map | N/A |

* + 1. Notes
* The HTTP Server Connection class conforms to the TCP Server Connection Class Pattern here above.
* The HTTP Connection callbacks can be changed with the route setter, usually in the onRequest callback, when the HTTP method, path, and headers are available.
* For examples of routes, see the [sample code](https://github.com/Moddable-OpenSource/moddable/blob/public/examples/io/listener/httpserver/options)
  1. Provenance Sensor Class Pattern
     1. configure(*options*)

1. Execute all steps of the Sensor Class Pattern configure method
   * 1. sample([*params*])
2. Execute steps 1 to 6 of the Sensor Class Pattern sample method
3. 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*)
4. Execute steps 7 of the Sensor Class Pattern sample method
   * + 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.
  + - 1. sample *params*:

None

* + - 1. sample *result*:

In addition to the sample results defined in the [Sensor Class Pattern](#alg-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 |

* + 1. 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.
  1. IO Provider Class Pattern
     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
   * 1. close()

Execute all steps of the Peripheral Class Pattern close method

Bibliography

**IO**

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>

**W3C Sensor**

1. W3C Generic Sensor specification. <https://www.w3.org/TR/generic-sensor/>
2. W3C Accelerometer draft. <https://w3c.github.io/accelerometer/>
3. W3C Ambient Light Sensor draft. <https://www.w3.org/TR/ambient-light/>
4. W3C Proximity Sensor draft. <https://w3c.github.io/proximity/>

**Secure ECMAScript (SES)**

1. Ecma TC39 - Compartments Proposal. <https://github.com/tc39/proposal-compartments>
2. Ecma TC39 - SES Proposal. <https://github.com/tc39/proposal-ses>
3. Draft Specification for Standalone SES. <https://github.com/Agoric/SES-shim/blob/master/packages/ses/docs/source/draft-standalone-spec.md>

**ResizableArrayBuffer and GrowableSharedArrayBuffer**

1. In-Place Resizable and Growable ArrayBuffers. <https://github.com/tc39/proposal-resizablearraybuffer>

Ecma International  
Rue du Rhone 114  
CH-1204 Geneva  
Tel: +41 22 849 6000  
Fax: +41 22 849 6001  
Web: <https://ecma-international.org/>

**Software license**

All Software contained in this document (“Software”) is protected by copyright and is being made available under the “BSD License”, included below. This Software may be subject to third party rights (rights from parties other than Ecma International), including patent rights, and no licenses under such third party rights are granted under this license even if the third party concerned is a member of Ecma International. SEE THE ECMA CODE OF CONDUCT IN PATENT MATTERS AVAILABLE AT <https://ecma-international.org/memento/codeofconduct.htm> FOR INFORMATION REGARDING THE LICENSING OF PATENT CLAIMS THAT ARE REQUIRED TO IMPLEMENT ECMA INTERNATIONAL STANDARDS.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the authors nor Ecma International may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE ECMA INTERNATIONAL “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ECMA INTERNATIONAL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.