Programmer's Guide to Kinoma Element

About This Document

Kinoma Element is hardware and software for building Internet of Things products. It delivers cost-effective Wi-Fi connectivity in a palm-sized package, and its software is primarily ES6, the latest version of JavaScript. This document provides both guidance and reference details, organized into the following sections:

- An introduction to Kinoma Element's main features (next)
- Overview of Developing for Kinoma Element, containing essential information that you need to know to develop for Kinoma Element
- Kinoma Element API Reference, which describes in detail the modules, objects, and classes that make up the Kinoma Element API
- Kinoma Element CLI Reference, a detailed reference for the Kinoma Element command-line interface

About Kinoma Element

Kinoma Element is hardware and software for building Internet of Things (IoT) projects, prototypes, and products. Built around a single chip that combines an ARM CPU, Wi-Fi, and memory, Kinoma Element delivers cost-effective Wi-Fi connectivity in a palm-sized package. Unlike most IoT hardware, Kinoma Element's software is primarily JavaScript—specifically ES6, the latest version of the language. JavaScript is a natural fit for a world filled with web services that communicate using JSON, and scripting speeds development and eases delivery of updates.

The chip at the heart of Kinoma Element is the Marvell MW302 system-on-chip (SoC), which features:

- 200 MHz ARM Cortex-M4
- Wi-Fi B/G/N
- 512 KB RAM
- 4 MB flash memory
- USB
- 16 programmable pins

The JavaScript engine powering Kinoma Element is XS6, the JavaScript virtual machine created by Kinoma and optimized for situations with limited memory and CPU power. XS6 implements the ES6 standard (ECMAScript 2015), so the language is the same as that used in web browser and servers. The environment in which the JavaScript runs is different—a tiny fraction of the memory, CPU power, and storage—so the JavaScript code developers create is different too.

As a foundation for IoT projects, prototypes, and products, Kinoma Element has built-in support for these popular network protocols:

- HTTP client and server
- HTTPS (TLS versions 3.1, 3.2, and 3.3) client and server
- WebSocket client and server
- CoAP client and server
- MQTT client
- mDNS announcement and discovery
- SSDP announcement

Developers may implement other network protocols in JavaScript using the built-in **`Socket`** objects, which support both TCP and UDP.

IoT projects, prototypes, and products often augment Kinoma Element with additional components, such as sensors, lights, motors, and buttons. To connect to these components, Kinoma Element has 16 reconfigurable pins.

- Power (3.3 V)
- Ground
- Digital Input/Output
- Analog Input
- Pulse Width Modulation (PWM)
- Serial Input/Output
- I^2C

The Pins module in Kinoma Element is a high-level API for working with components attached to the hardware pins; it interacts with pins using URLs and callbacks, much like making requests to a web server. Details about the Pins module can be found in the document *Using the Pins Module to Interact with Sensors on Kinoma Element* (../element-pins-module).

Overview of Developing for Kinoma Element

This section provides essential information that you need to know to develop for Kinoma Element.

Application Project File

Applications for Kinoma Element (or Kinoma Create) can be developed in the Kinoma Code IDE, which enables you to configure Wi-Fi, reprogram hardware pins, update firmware, and scan network devices. Applications developed in Kinoma Code use a JSON project file that is always named **`project.json`**. The project file defines the application name, ID, and entry point.

```
{
    "id": "example.kinoma.marvell.com",
    "title": "Example",
    "Element": {
        "main": "main"
    }
}
```

The ID is a string that uniquely identifies the application; by convention, the application ID uses a dotted domain name style. The application title is a string containing the application name to display to users.

The **`Element`** object must be present to run the application on Kinoma Element. It tells the IDE that this application is compatible with Kinoma Element. The **`main`** entry inside the **`Element`** object specifies the name of the program file (without the file name extension) to run to launch the application. Most applications use **`main.js`**, so the program file root is **`main`**. If the program file name is, for example

to launch the application. Most applications use **main.js**, so the program file root is **main**. If the program file name is, for example, **starthere.js**, the **Element** object is as follows:

```
"Element": {
    "main": "starthere"
}
```

Application Structure

There is a difference between a program and an application for Kinoma Element:

- A *program* for Kinoma Element is a JavaScript module that runs without using events or callbacks and so does not export an object with event handlers.
- An *application* for Kinoma Element is a JavaScript module that exports an object with event handlers and so can use events and callbacks.

Most Kinoma Element projects are applications, but programs are useful for simple experiments. The following **main.js** is a simple example of a program; it writes system values to the console.

```
import console from "console";

console.log(`System.host: ${System.hostname}`);
console.log(`System.device: ${System.device}`);
console.log(`System.osVersion: ${System.osVersion}`);
console.log(`System.timezone: ${System.timezone}`);
```

An application is an object with event handlers, as shown in the following **main.js**. The application object is the default export of the module.

```
export default {
    onLaunch() {
    },
    onQuit() {
    }
};
```

The **onLaunch** event is called when the application begins executing, and the **onQuit** event is called when execution ends. The application is terminated after **onQuit** returns, immediately cancelling pending asynchronous operations. An application may omit the **onLaunch** and **onQuit** events if it does not use them.

```
import HTTPClientRequest from "HTTPClient";

let main = {
    onLaunch() {
        this.request = new HTTPClientRequest("http://www.kinoma.com/file.txt");
        this.request.onTransferComplete = success =>
            console.log(String.fromArrayBuffer(this.request.content));
        this.request.start();
    }
};
export default main;
```

Garbage Collector

The JavaScript language uses a garbage collector for memory management. The garbage collector determines which objects are no longer in use and automatically frees their memory. In Kinoma Element, the garbage collector follows the JavaScript specification. Programmers developing code for Kinoma Element need to be more aware of the garbage collector than those working on web browsers and web servers, for these reasons:

- Kinoma Element has considerably less memory, so the garage collector executes more frequently, causing unused objects to be collected sooner.
- Kinoma Element has a relatively lightweight application framework that delegates management of object lifetime to the application.

Applications on Kinoma Element need to be aware of the garbage collector when invoking asynchronous operations, such as making HTTP client requests or starting an HTTP server. In the following example, the HTTP client request may be garbage-collected before the request completes, because once **onLaunch** returns, the local variables (for instance, **request**) go out of scope, leaving no JavaScript reference to the request.

```
onLaunch() {
    let request = new HTTPClientRequest("http://www.kinoma.com/file.txt");
    request.onTransferComplete = success => console.log("done");
    request.start();
}
```

To ensure that the request is not garbage-collected before it completes, you can assign the request to another object that will not be garbage-collected. Because the application object is not collected until the application terminates, it is a convenient object to which to attach the HTTP request.

```
onLaunch() {
    let request = new HTTPClientRequest("http://www.kinoma.com/file.txt");
    this.request = request;
    request.onTransferComplete = success => console.log("done");
    request.start();
}
```

Single-Threaded Runtime

Kinoma Element uses a single-threaded runtime. This approach is consistent with the single-threaded model of the JavaScript language, helps conserve memory, and generally simplifies programming. Because the programming model is single-threaded, all APIs are expected to either execute quickly (that is, within a few milliseconds) or use a callback to indicate when an operation is complete.

```
Socket.resolv("kinoma.com", result => console.log("Socket.resolv complete"));
```

Some objects in Kinoma Element use an event-driven model to report notifications.

```
let tcp = new Socket({host: "www.kinoma.com", port: 80, proto: Socket.TCP});
tcp.onConnect = () => console.log("connected");
tcp.onData = buffer => console.log(`received ${buffer.byteLength} bytes")
```

Application scripts should follow these same principles—that is, execute quickly or use callbacks and events to break long operations into pieces.

Because Kinoma Element is single-threaded, certain operations block execution. Of particular note, the Telnet and USB console connections are blocked when the application stops at a breakpoint in the debugger. The console resumes normal operation when the debugger allows the application to continue running.

Kinoma Element has a watchdog timer for detecting when an application blocks for an extended period. If it detects this situation, the watchdog assumes that the application is in an infinite loop or has crashed, and restarts the system.

Telnet

The Kinoma Element command-line interface (CLI) is available over Telnet on port 2323. All diagnostic output is sent to the Telnet connection.

You can connect to Telnet from the terminal in Mac OS X using the IP address of Kinoma Element, as follows:

```
telnet 10.0.1.69 2323
```

Kinoma Element advertises the address of its Telnet service using Zeroconf (mDNS), enabling a connection to be established by name rather than by IP address. If you have not set your Kinoma Element's hostname, the advertised name is based on its MAC address (**xxxxxx** in the following example).

```
telnet "Kinoma Element-xxxxxx".local 2323
```

After you set the hostname, connections can be established using that hostname (**K5_25** in the following example).

```
telnet K5_25.local 2323
```

USB

The Kinoma Element CLI is also available over USB using a serial terminal application. All diagnostic output is sent to the USB console.

Many different serial terminal applications are available, and they vary by platform. Common serial terminal applications include **cu**, Minicom, and Screen.

Diagnostic Output

Kinoma Element operates a console for diagnostic output. Scripts can use **console.log** and **trace** to send information to the diagnostic output. JavaScript exceptions are logged, in addition to any diagnostic output by other modules.

Output is sent to the Telnet connection, the USB console connection, and the debugging console output.

TFTP

Kinoma Element implements the Trivial File Transfer Protocol (TFTP) to upload files to and download files from Kinoma Element. Most developers do not need to use TFTP directly when working with the Kinoma Code IDE. TFTP is useful for retrieving log files and interacting directly with the Kinoma Element file system.

> ***Important:*** Because TFTP is a low-level tool, it is possible to damage the file system on Kinoma Element by overwriting key files, so use TFTP with caution.

To communicate with Kinoma Element using TFTP, use a TFTP client. On Mac OS X, use the **tftp** command-line tool. Start by specifying the port and host, as follows:

```
tftp -e 10.0.1.10 6969
tftp -e <Element-hostname>.local 6969
```

The **-e** option indicates that the data transfer mode is binary, which is recommended for Kinoma Element.

The TFTP service on Kinoma Element always operates on the **k3** partition; it cannot access other partitions.

To upload a file, use the **put** command.

```
tftp > put hello.xsb
```

To update the firmware directly:

```
tftp > put xsr6_mc_k5.bin
```

To retrieve the console log file:

```
tftp > get log
```

Open Source

The majority of software in Kinoma Element is available under the Apache open source license. The Kinoma team is working to make all the software in Kinoma Element open source.

The software is in the KinomaJS repository (https://github.com/Kinoma/kinomajs/tree/master/xs6/sources/mc) on GitHub.

Instructions to build the open source code are in the README file (https://github.com/Kinoma/kinomajs/blob/master/README.md), in the section about Kinoma Element.

API Stability

The Kinoma Element API Reference section describes the modules, objects, and classes that the Kinoma team designed for development of applications and BLLs. (For more about BLLs, see *Programming with Hardware Pins for Kinoma Element* (../element-bll/).) These public APIs are not expected to change significantly; if changes are made, they will be documented.

Kinoma Element contains other software which is not documented. This software is often used to implement the public APIs, or contains capabilities that have not yet been fully developed. You can learn about these APIs by inspecting JavaScript objects in the debugger or

reviewing the Kinoma Element open source code. These undocumented APIs may change, or even be removed, without notice. You are free to use them, of course, but they may not be supported by the Kinoma engineering team.

JavaScript Implementation

Kinoma Element uses the XS6 JavaScript virtual machine. XS6 implements the ES6 specification, also known as ECMAScript 2015. XS6 has excellent compatibility with the JavaScript standard, enabling developers to use the same language they already know from web browsers and servers.

Details of the JavaScript implementation to be aware of are provided below. Resource-constrained hardware motivates the differences. Kinoma Element has significantly less memory and CPU performance than is common in web browsers and servers. These differences may be relevant for large applications and applications that import code modules from other environments.

String.fromArrayBuffer(buffer)

Kinoma Element applications work with sensor inputs and network protocols that provide data in ArrayBuffers, to accommodate binary data. That binary data may contain string data. The ES6 specification does not provide a function to directly convert binary data to a string. The operation can be implemented in JavaScript but will be relatively inefficient on a resource-constrained device like Kinoma Element. Because this operation is common in Kinoma Element applications, the **String** object is extended with the built-in function **fromArrayBuffer**.

```
let text = String.fromArrayBuffer(buffer);
```

The buffer must contain a valid UTF-8 string.

To extract a string from part of an ArrayBuffer, extract the data into a new ArrayBuffer first.

```
let substring = buffer.slice(10, 20);
let string = String.fromArrayBuffer(substring);
```

function Object

The **function** object differs from the ES6 specification in three ways, each of which reduces runtime memory use.

- The **length** property of a function, which indicates the expected number of arguments to the function, returns **undefined**.
- The **name** property of some functions returns **undefined**. The **name** property is present in the ES6 specification for debugging purposes, and its value is defined as implementation-dependent in the standard.
- The **toString** property of a function returns only a stub text representation of the function. (The function source code or a decompiled version of the code is unavailable.)

Executing JavaScript Source Code

Both the **function** object constructor and the **eval** function enable scripts to parse and execute JavaScript source code directly. This capability is implemented on Kinoma Element; however, it is recommended that scripts use the **function** object constructor and **eval** rarely, if at all. The parsing of the ES6 JavaScript and subsequent byte code generation requires considerable memory and so may fail when memory is low or the script is large.

require(module) and require.weak(module)

In ES6 JavaScript, modules are accessed using the **import** statement.

```
import console from "console";
console.log("using import statement");
```

The **import** statement creates a variable named **console**, visible to all code in the loading module, to use for accessing the properties of the imported module.

In versions prior to ES6, the **import** statement is unavailable. The CommonJS module specification, often used by pre-ES6 code, defines the **require** function for accessing a module.

```
console = require("console");
console.log("using require function");
```

Although the functionality is similar, the runtime behavior is different: the **import** statement completes execution before the script containing it begins execution, while the **require** function executes only at the time it is called. This difference means that modules referenced using **import** are always loaded, whereas **require** loads modules only when invoked.

Kinoma Element implements the global **require** function for loading ES6 modules. This function returns the default export of the module.

```
if (loggingEnabled) {
    let console = require("console");
    console.log("log entry");
}
```

CommonJS defines all calls to **require** for a given module to return the same exported object. That limits the ability of the garbage collector to collect the module. This behavior is convenient for developers but tends to increase overall memory use. Kinoma Element implements the **require.weak** function to load a module and leave it eligible for garbage collection. If the module is already in memory, **require.weak** returns the same exported object, so there is never more than one copy of the module loaded.

```
if (loggingEnabled) {
    let console = require.weak("console");
    console.log("log entry");
}
// Console is out of scope here, so may now be garbage-collected
```

Applications typically use the ES6 **import** statement. If you find that memory is tight, it may be worthwhile to consider using **require** or **require.weak** to reduce runtime memory.

> **Note:** Not all modules are designed to be garbage-collected. Scripts should use **require.weak** only with modules that support unloading.

trace(string)

The **trace** global function is similar to **console.log**, but unlike **console.log**, **trace** does not add a line feed at the end of the output. The **trace** function takes a single string argument.

```
trace("1");
trace("2");
trace("3\n");
```

The output in this case would be:

```
123
```

Note: The output of **trace** may not be visible until a line feed (**\n** above) is processed as part of the output.

Relationship to Kinoma Create

Kinoma Element and Kinoma Create are both combinations of hardware and software for building IoT projects, prototypes, and products. Kinoma Element is a considerably smaller, lighter-weight product, suitable for embedding digital intelligence and Wi-Fi connectivity into just about anything. Kinoma Create is built around a much more powerful CPU that has more memory (approximately 200 times more) and runs the Linux operating system, all to support the needs of the built-in touch screen, audio speaker, and microphone.

Despite their significant hardware differences, Kinoma Element and Kinoma Create share a great deal of software, making it easier for developers to move back and forth between them. These are some of the ways in which Kinoma Element and Kinoma Create have software compatibility:

- Both are powered by the XS6 virtual machine, and so provide the same JavaScript language implementation.
- Both use the Pins module for applications to communicate with hardware pins.
- Both support the BLL model of JavaScript drivers for hardware components. The same BLL often works on both.
- Both support the WebSocket API.
- Both use the Kinoma Code IDE for software development.

In addition, Kinoma Create supports the KinomaJS Files API (../filesAPI/) and Kinoma Element implements a **Files** class with a subset of that functionality.

However, while Kinoma Element and Kinoma Create offer many of the same software APIs, it is not intended that the same applications will work for both. The different form factor and power demand a different application for each. Developers often will be able to reuse their code that communicates with pins, as well as many modules that they create.

Turning Off Kinoma Element

The recommended way to turn off Kinoma Element is to press the power button. When the light goes out, Kinoma Element has powered down. This method of powering down gives Kinoma Element an opportunity to cleanly exit the current application (by calling its **onQuit** event) and to announce to the network that the device is going away.

If Kinoma Element is turned off by removing power, other devices on the network will not receive a notification that Kinoma Element has turned off. They may continue to show Kinoma Element services as available in their user interface. For example, Kinoma Code may continue to show a powered-down Kinoma Element in its device list for several minutes before the mDNS announcement times out.

Kinoma Element API Reference

This section describes in detail the modules, objects, and classes that make up the Kinoma Element API.

The following conventions are used in this section:

- **Default values** — Unless noted otherwise, the default values for optional value properties or parameters are 0, **false**, and **undefined** for types **number**, **boolean**, and **string**, respectively.
- **"Read only"** — All value properties can be read, but only some can be set; properties that cannot be set are designated as "Read only."

System Object

The **System** object contains functions that report and configure fundamental device capabilities.

```
import System from "system";
```

Functions

addPath(path)

Adds one or more directories to the search path used when loading JavaScript modules. Most applications will not need to add additional paths, because the module search path is configured by the Kinoma Code IDE.

```
System.addPath("/k2/myModules/", "/k3/moreModules/");
```

gc()

Runs the JavaScript garbage collector. Applications do not usually need to call this function, because the garbage collector runs automatically when needed.

reboot(force)

Restarts the device. If the optional **force** parameter is set to **true**, the device is immediately restarted, without cleanly exiting any applications and services.

```
System.reboot();
```

shutdown(force)

Shuts down the device. If the optional **force** parameter is set to **true**, the device is immediately shut down, without cleanly exiting any applications and services.

```
System.shutdown(true);
```

Values

device

A string indicating the model of the device running the script (**K5** for Kinoma Element). *(Read only)*

```
console.log(`Device is ${System.device}`);
```

hostname

A string indicating the name of the device.

```
console.log(`Hostname is ${System.hostname}`);
System.hostname = "khin";
```

osVersion

A string indicating the version of the operating system the device is running. *(Read only)*

```
console.log(`osVersion is ${System.osVersion}`);
```

The format of the string is, for example, **WM/3001016**.

platform

A string indicating the platform software name (**mc** for Kinoma Element). *(Read only)*

```
console.log(`platform is ${System.platform}`);
```

time

A number indicating UNIX time in seconds.

```
console.log(`time is ${System.time}`);
System.time += 10;        // Jump 10 seconds into the future
```

> **Note:** The **System.time** property is initialized from a network source when Kinoma Element first boots. The real-time clock in Kinoma Element is not maintained when power is turned off.

timestamp

A number indicating the UNIX time when the operating system software was built. *(Read only)*

```
console.log(`timestamp is ${System.timestamp}`);
```

timezone

An object containing a number specifying the difference in seconds between the current device location and UTC, and a number flag indicating whether daylight savings time is active.

```
console.log(`Time zone offset is ${System.timezone.timedifference}`);
console.log(`Daylight savings is ${System.timezone.dst}`);

System.timezone = {timedifference: 0, dst: 1};    // London with daylight savings time active
```

Environment Class

The **Environment** class is a simple way for applications to store small pieces of data.

```
import Environment from "env";
```

There is a default system environment store, and applications can create additional environment stores as necessary. An environment store can be encrypted, making it suitable for storing passwords.

Constructor

new Environment(path, autosave, encrypt)

Creates an instance for accessing a single environment store. The parameters are:

- **path** — *(Optional)* Indicates the path to the environment store to use. If the environment store does not already exist, it is created. If the **path** argument is not present, the default system environment store is used.
- **autosave** — *(Optional)* Indicates whether the environment store should be saved after each change. If **autosave** is **false** (the default), the environment store is saved only when the **save** and **close** functions are called.

> **Note:** Saving the environment store takes some time, and a value of **false** for **autosave** enables the application to decide when to update the environment store. For applications that only occasionally update the environment store, setting **autosave** to **true** is recommended in order to help ensure the integrity of the environment file.

- **encrypt** — *(Optional)* If set to **true**, causes the values in the environment store to be encrypted in storage. The values are automatically decrypted when accessed. Encrypted values take more time to get and set. The default system environment store is not encrypted; the environment store used to maintain the passwords for Wi-Fi access points is encrypted.

```
let env = new Environment("myEnv");
```

Functions

close()

`close()`

Closes, and if necessary saves, the environment store. After **close** is called, no additional calls should be made to this **Environment** instance.

`get(name)`

Returns the value in the environment store associated with the key specified by the **name** string. If the key is not present in the environment store, **get** returns **null**.

```
let env = new Environment();
let firmwareVersion = env.get("FW_VERS");
```

`save()`

Stores the content of the environment store. A save is automatically performed by **close**, or by **set** if **autosave** is enabled.

`set(name, value)`

Stores the **value** parameter (converted to a string, if needed) to the environment store under the key specified by **name**. If there is already a value associated with **name**, it is replaced by the specified value.

If **value** is not passed or is set as **undefined**, **set** removes the key associated with **name** from the environment store.

```
let env = new Environment("myEnv");
env.set("foo", "test");
let foo = env.get("foo");    // returns "test"
env.set("foo");
foo = env.get("foo");        // returns null
```

Iterators

The **Environment** class implements an iterator to enable applications to retrieve all the keys of an environment store.

```
let env = new Environment();
for (let name of env)
    console.log(`${name} = ${env.get(name)}\n`);
```

uuid Object

The **uuid** object creates, and optionally caches, universally unique identifier (UUID) values. UUID values are strings.

```
import uuid from "uuid";
```

Functions

`create()`

Generates a new UUID value from a random number, the current time, and the network connection's MAC address.

```
for (let i = 0; i < 5; i++)
    console.log(`UUID ${i}: ${uuid.create()}`);
```

`get(name)`

Checks the UUID cache for a UUID of the **name** parameter. If found, the UUID is returned; otherwise, a new UUID is generated, cached, and returned.

```
let deviceUUID = uuid.get("device");
```

TimeInterval Class

The **TimeInterval** class implements core timer functionality. Higher-level timing functions, including **setTimeout** and **setInterval**, are implemented using **TimeInterval**.

```
import TimeInterval from "timeinterval";
```

Only a limited number of **TimeInterval** instances may be allocated in the system. The limit is 24 in the default configuration.

Constructor

`new TimeInterval(callback, interval)`

Creates a **TimeInterval** instance. When the time interval fires, the function specified by the **callback** parameter is called. The initial time interval is specified by the **interval** parameter in milliseconds.

The newly created **TimeInterval** instance is not active. The **start** function must be called to activate it.

```
let timer = new TimeInterval(() => console.log(`another second ${Date.now()}`), 1000);
timer.start();
```

Functions

`close()`

Closes the **TimeInterval** instance, so no further callbacks fire. After **close** is called, no additional calls should be made to this **TimeInterval** instance.

start(interval)

Activates the **TimeInterval** instance, so that the callback fires after the interval has elapsed. The optional **interval** parameter specifies the interval in milliseconds; if it is not present, the most recent interval passed to the constructor or the **start** function is used.

```
let timer = new TimeInterval(() => console.log(`timer fired`), 1000);
timer.start(500); //will fire in 500ms, not 1000ms
```

stop()

Deactivates the **TimeInterval** instance. The callback does not fire until **start** is called.

Values

interval

The current interval of the **TimeInterval** instance, in milliseconds. *(Read only)*

```
let timer = new TimeInterval(() => console.log(`timer fired`), 1000);
timer.start();
console.log(`current interval is: ${timer.interval}`); //1000
```

Timer Module

The Timer module contains implementations of four functions commonly used by HTML5 applications for time callbacks.

```
import {setInterval, clearInterval, setTimeout, clearTimeout} from "timer";
```

The Timer module is implemented using the **TimeInterval** class. The Timer module guarantees that pending timeouts and intervals cannot be garbage-collected, whereas **TimeInterval** instances can be garbage-collected.

For the convenience of developers familiar with HTML5, the Kinoma Element application runtime makes **setInterval**, **clearInterval**, **setTimeout**, and **clearTimeout** available as global functions.

Functions

clearInterval(interval)

Cancels a repeating callback created using **setInterval**.

```
let repeater = setInterval(() => console.log(`Tick $(Date.now()`), 1000);
// ... some time later
clearInterval(repeater);
```

clearTimeout(timeout)

Cancels a one-time callback created using **setTimeout**.

```
let oneshot = setTimeout(() => console.log("Two seconds later"), 2000);
// ... less than two seconds later
clearTimeout(oneshot);
```

setInterval(callback, delay)

Schedules a repeating callback at an interval specified in milliseconds by the **delay** parameter.

```
setInterval(() => console.log(`Tick $(Date.now()`), 1000);
```

setInterval returns a reference to the new interval, which can be used to cancel the interval using **clearInterval**.

setTimeout(callback, delay)

Schedules a one-time callback to be called after a time specified in milliseconds by the **delay** parameter.

```
setTimeout(() => console.log("Two seconds later"), 2000);
```

setTimeout returns a reference to the new timeout, which can be used to cancel the interval using **clearTimeout**.

wdt Object

The **wdt** (watchdog timer) object monitors device activity. If it detects that the device has been blocked for an extended period of time, the watchdog timer assumes the application has crashed, and restarts the device.

```
import wdt from "watchdogtimer";
```

The length of time the watchdog timer waits before deciding the application has crashed depends on the mode of operation. It is always at least 5 seconds.

Most applications do not need to interact with the watchdog timer directly, since they do not typically perform operations that block for an extended period. If an application performs a blocking operation that it expects to take more than 5 seconds, it has two options to prevent the watchdog timer from restarting the device:

- Periodically call the watchdog timer to tell it that the application is operating normally, by calling **strobe**. This resets the watchdog timer.
- Disable the watchdog timer when starting the operation, and restart it when complete.

In general, applications are advised to use the **strobe** approach.

> **Note:** Certain operations temporarily disable the watchdog timer. In particular, it is disabled during a debugging session so that

breakpoints do not cause the watchdog to time out and restart the device.

Functions

`resume()`

Reenables the watchdog timer after it has been disabled by the **`stop`** function.

`stop()`

Disables the watchdog timer. Applications should use **`stop`** only when absolutely necessary, as it prevents the system from detecting when an application is stuck in an infinite loop.

Use the **`resume`** function to reenable the watchdog timer.

`strobe()`

Tells the watchdog timer that the application is properly functioning during a long task. Calling **`strobe`** resets the watchdog timer.

Files Class

The **`Files`** class provides tools for working with the file system.

```
import Files from "files";
```

The embedded file system implementation has some limitations:

- File paths can be up to 128 bytes, including the file name, file extension, and all parent directory names.
- Path names are encoded with UTF-8 encoding, so non-ASCII characters use more than one byte.
- The file system uses a slash (**`/`**) as the directory separator. The file system does not ignore multiple slashes in a row, so **`/k1/data/foo.txt`** and **`/k1/data//foo.txt`** refer to different files.
- The path to a directory or volume always ends with **`/`**. The path to a file never ends with **`/`**.
- Empty directories are unsupported. If a directory contains no files or directories, it is automatically deleted.

Note: The **`Files`** class is designed for upward compatibility with the KinomaJS Files API, although they are not identical.

Iterators

`Files.Iterator(path);`

Retrieves all the files and directories contained in the directory specified by the **`path`** parameter.

```
for (let item of Files.Iterator(path))
    console.log(item.name);
```

`Files.VolumeIterator();`

Retrieves all the volumes on the device.

```
for (let item of Files.VolumeIterator())
    console.log(`Volume name ${item.name} at path ${item.path}`);
```

Static Functions

`deleteDirectory(path)`

Deletes the directory specified by **`path`**. This operation is recursive, so all files and directories contained in the directory are also deleted.

```
Files.deleteDirectory("/k1/data/");
```

`deleteFile(path)`

Deletes the file with the specified path.

```
Files.deleteFile("/k1/data/foo.txt");
```

`deleteVolume(name)`

Erases the volume with the specified name.

```
Files.deleteVolume("k1");
```

`getInfo(path)`

Retrieves information about the file at the specified path. If no file exists at the specified path, this function returns **`undefined`**.

```
let path = "/k1/data/foo.txt";
let info = Files.getInfo(path);
console.log(`File ${path} has ${info.size} bytes.`);
```

`getSpecialDirectory(name)`

Retrieves the path to a special system-designated directory associated with the **`name`** parameter.

```
Files.getSpecialDirectory("documentsDirectory");
```

The following special directory names are available:

- `applicationDirectory`
- `preferencesDirectory`
- `documentsDirectory`
- `variableDirectory`
- `nativeApplicationDirectory` (simulator only)

`getVolumeInfo(path)`

Retrieves information about the volume at the specified path. If no volume exists at the specified path, this function returns `undefined`.

```
let path = "/k1/";
let info = Files.getVolumeInfo(path);
console.log(`Volume ${path} has ${info.size} bytes with removable ${info.removable}.`);
```

`read(path)`

Reads the entire contents of the file at the specified path into an ArrayBuffer.

```
let buffer = Files.read("/k1/data/foo.txt");
console.log(String.fromArrayBuffer(buffer));
```

`renameFile(from, to)`

Renames the file at the path specified by the `from` parameter to the name given in the `to` parameter.

```
Files.renameFile("/k1/data/foo.txt", "bar.txt");
```

`write(path, buffer)`

Replaces the content of the file at the specified path with the content of the `buffer` parameter. The `buffer` argument may be an ArrayBuffer or a string.

```
Files.write("/k1/data/foo.bin", new ArrayBuffer(10));
Files.write("/k1/data/foo.txt", "Hello, world.");
```

File Class

The `File` class provides tools for reading and writing data to an individual file.

```
import File from "file";
```

Constructor

`new File(path, mode)`

Opens the file specified by `path`. (See Files Class for information about paths.) If the optional `mode` parameter is 0 or unspecified, the file is opened for read-only access; if `mode` is 1, the file is opened for write access. If `mode` requests write access and the file does not exist, the file is created and then opened.

```
let file = new File("/k1/data/foo.txt");
```

Functions

`close()`

Closes the file. After `close` is called, no additional calls should be made to this `File` instance.

`read(type, count, buffer)`

Reads data from the file. The `count` parameter specifies the number of bytes to read. The `type` parameter indicates which JavaScript object to read the data into, as either `String` or `ArrayBuffer`.

```
let str = file.read(String, 5);
let bytes = file.read(ArrayBuffer, 10);
let all = file.read(ArrayBuffer, file.bytesAvailable);
```

If `type` is `ArrayBuffer`, an ArrayBuffer can be passed in the `buffer` parameter as an optimization to minimize buffer allocations.

```
let buffer = new ArrayBuffer(10);
file.read(ArrayBuffer, buffer.byteLength, buffer);
file.read(ArrayBuffer, buffer.byteLength, buffer);
```

`readChar()`

Reads one byte from the file, returning the value as a number between 0 and 255.

```
let byte = file.readChar();
```

`write(item...)`

Stores data to the file. Each argument is one of the following: an integer representing the byte value to write; a string; an ArrayBuffer; or an array containing integers, strings, ArrayBuffers, and arrays.

```
file.write(42);
file.write("a string", [13, 10]);
file.write((buffer.length >> 8) & 0xff, buffer.length & 0xff, buffer);
```

Values

length

The total number of bytes in the file. Read this property to retrieve the size, and set it to change the size.

```
console.log(`This file contains ${file.length} bytes.`);
file.length = 0;    // Empty file contents
```

position

The current read/write position in the file. When the file is first opened, the position is 0. The **read**, **readChar**, and **write** functions update the position.

```
file.position = file.length;    // Set position to end of file
```

HTTPClientRequest Class

The **HTTPClientRequest** class implements support for making a request to an HTTP server.

```
import HTTPClientRequest from "HTTPClient";
```

HTTPClientRequest supports HTTP and HTTPS connections.

Examples

Get JSON

```
let request = new HTTPClientRequest("https://www.kinoma.com/example.json");
request.onTransferComplete = success => {
    let body = String.fromArrayBuffer(request.content);
    let message = JSON.parse(body);
    // etc.
}
request.start();
```

Post JSON

```
let request = new HTTPClientRequest("https://www.kinoma.com/post");
request.method = "POST";
request.setHeader("Content-Type", "application/JSON");
request.start(JSON.stringify({
    command: "something",
    value: 12,
    option: "do the right thing"
}));
```

Download to File

```
let request = new HTTPClientRequest("http://www.kinoma.com/file.txt");
let file = new Files("/k1/file.txt");
request.onDataReady = buffer => file.write(buffer);
request.onTransferComplete = success => file.close();
request.start();
```

Constructor

new HTTPClientRequest(url)

Initializes a new HTTP request. Call **start** to initiate the request to the server.

```
let request = new HTTPClientRequest("http://www.kinoma.com/index.html");
```

The new HTTP request is initialized with a request method of **GET**; set the **method** value property to change it.

Functions

getHeader(name)

Retrieves the header specified by **name** from the HTTP response headers. If no header is found with the specified name, **undefined** is returned.

```
let when = request.getHeader("date");
```

HTTP response headers are available only after the **onHeaders** event is invoked.

> **Note:** HTTP header names are case-insensitive, so **getHeader("FOO")** and **getHeader("Foo")** refer to the same header.

setHeader(name, value)

Adds an header name and value to the HTTP request headers.

```
request.setHeader("Content-Type", "text/plain");
```

start(body)

Begins the HTTP request to the server. The optional **body** parameter is a string or ArrayBuffer for the request body.

```
request.start(JSON.stringify({
    command: "something",
    value: 12,
    option: "do the right thing";
}));
```

Values

content

The complete HTTP response body after the **onTransferComplete** event is called. If an application replaces the **onDataReady** event, the content is undefined.

```
let bodyText = String.fromArrayBuffer(request.content);
```

method

The request method of the request. The default method is **GET**.

```
request.method = "HEAD";
```

statusCode

The HTTP status code in the response headers. It is available after the **onHeaders** event has fired.

```
if (request.statusCode == 404)
    console.log("resource not found");
```

Events

onDataReady(buffer)

Called when data arrives for the response body. The **buffer** parameter is an ArrayBuffer containing the data. **onDataReady** may be called multiple times for a single request.

```
request.onDataReady = buffer => {
    console.log("Received ${buffer.byteLength} bytes");
    super.onDataReady(buffer);
}
```

Applications are not required to implement **onDataReady**. The default implementation of **onDataReady** concatenates all the data into a single ArrayBuffer that is available when the **onTransferComplete** event is called. The **onDataReady** event is useful for applications that require immediate processing of the incoming data.

onDataSent()

Called after the request body has been completely transmitted to the server. If there is no request body, **onDataSent** is not called.

```
request.onDataSent = () => {
    console.log("HTTP request body sent");
}
```

onHeaders()

Called when the HTTP response headers are available.

```
request.onHeaders = () => {
    this.mime = request.getHeader("Content-Type");
}
```

onTransferComplete(success)

Called when the entire HTTP response body has been received. The **success** parameter is a boolean indicating whether the request completed successfully. In some cases the HTTP status code may not be known—for example, when the server is unreachable.

The message body is stored in the **content** property of the **HTTPClientRequest** instance, unless the application has replaced the default implementation of **onDataReady**.

```
request.onTransferComplete = success => {
    if (success && (200 == request.statusCode)) {
        let bodyText = String.fromArrayBuffer(request.content);
        console.log(bodyText);
    }
    else
        console.log(`request failed with error ${request.statusCode}`);
}
```

HTTPServer Class

The **HTTPServer** class provides support for applications to implement HTTP and HTTPS servers.

```
import HTTPServer from "HTTPServer";
```

When the server receives a request, it calls the **onRequest** event to respond, passing the **HTTPServerRequest** instance that represents the request.

Examples

Echo Server

This HTTP server responds to all requests with the requested URL in a text document.

```
let server = new HTTPServer({port: 80});
server.onRequest = request => {
    request.response("text/plain", `Your URL is ${request.url}.`);
}
```

File Server

This HTTP server returns files based on the URL path. The form of the URL is

```
http://<IP-address>/file/<path>
```

(for example, **http://10.0.1.2/file/k1/foo.txt**).

```
let server = new HTTPServer({port: 80});
server.onRequest = request => {
    if (request.url.startsWith("/file/")) {
        let data = Files.read(request.url.substring("/file/".length));
        request.response("text/plain", data);
    }
    else
        request.errorResponse(403, "Forbidden");
}
```

REST JSON Server

This HTTPS server responds to JSON requests with a portion of their JSON data converted to lowercase.

```
let server = new HTTPServer({port: 443, ssl: true});
server.onRequest = request => {
    let json = JSON.parse(request.content);
    if (json.name)
        json.name = json.name.toLowerCase();
    request.response("application/JSON", JSON.stringify(json));
}
```

Constructor

new HTTPServer(params)

Takes a single argument, a dictionary of initialization properties.

```
let server = new HTTPServer({port: 80});
let secureServer = new HTTPServer({port: 443, ssl: true});
```

There are three properties in the **params** dictionary:

- **port** — The port number that the HTTP server will listen on for connection requests.
- **socket** — A listening socket to use for listening for connection requests. If a socket is not provided, the HTTP server will allocate one.
- **ssl** — A boolean indicating whether this connection should be secure.

The server is active immediately after being created.

Functions

close()

Terminates the **HTTPServer** instance. All active connections are closed immediately, as is the listening socket.

Note: The server closes the listening socket only if it allocated it.

Events

onRequest(request)

Called when the HTTP server receives a new connection. When the **onRequest** event is received, the request headers and request body, if any, are available.

```
server.onRequest = request => {
    console.log(`URL: ${request.url);
    console.log(`Date header: ${request.getHeader("date")});
    if (request.content)
        console.log(`Request body: ${String.fromArrayBuffer(request.content)}`);
    request.response("text/plain", "Hello");
}
```

Multiple HTTP server requests may be active at the same time.

HTTPServerRequest Class

An **HTTPServerRequest** object is created by the HTTP server for each incoming request. (Applications do not create **HTTPServerRequest** objects directly.) The **request** object is passed to the application by **onRequest** even on the HTTP server.

Functions

`errorResponse(statusCode, reason, close)`

Sets the response for a request that was not successfully handled.

- **`statusCode`** — The HTTP status code
- **`reason`** — The text reason for failure
- **`close`** — *(Optional)* If set to **`true`**, the socket associated with the request is closed immediately after the response is sent

```
request.errorResponse(404, "Not Found");
request.errorResponse(403, "Forbidden", true);    // Close immediately
```

`getHeader(name)`

Retrieves the header specified by **`name`** from the HTTP request headers. If no header is found with the specified name, **`undefined`** is returned.

```
let when = request.getHeader("date");
```

Note: HTTP header names are case-insensitive, so **`getHeader("FOO")`** and **`getHeader("Foo")`** refer to the same header.

`putChunk(buffer)`

Sends part of the data in a chunked response. See **`responseWithChunk`** for an example.

`response(contentType, buffer, close)`

Sets the response for a request that was successfully handled.

- **`contentType`** — The MIME type of the data
- **`buffer`** — A string or ArrayBuffer containing the response body
- **`close`** — *(Optional)* If set to **`true`**, the socket associated with the request is closed immediately after the response is sent

```
request.response();        // no response body
request.response("text/plain", "Hello");
request.response("application/JSON", JSON.stringify({foo: 1});
```

`responseWithChunk(contentType)`

Sends a chunked response—for example, a response using **`Transfer-Encoding: chunked`**. This function is used together with **`putChunk`** and **`terminateChunk`**. The optional **`contentType`** parameter sets the **`Content-Type`** header in the response.

```
server.onRequest = request => {
    this.request = request;
    request.responseWithChunk();
}

// ... some time after onRequest is received
this.request.putChunk("some text");

// ... more time passes
this.request.putChunk(new ArrayBuffer(5));
this.request.terminateChunk();
```

`setHeader(name, value)`

Adds a header name and value to the HTTP response headers.

```
request.setHeader("Content-Type", "text/plain");
```

`terminateChunk(footer)`

Finishes sending a chunked response. See **`responseWithChunk`** for a complete example.

The optional **`footer`** parameter is a dictionary sent at the end of the message.

```
request.terminateChunk({status: 3});
```

Values

`content`

The complete HTTP request body, as an ArrayBuffer.

```
let json = JSON.parse(request.content);
```

`method`

The HTTP method of the request, as a string.

```
console.log(`Method is: ${request.method}`);
```

`url`

The **`path`**, **`query`**, and **`fragment`** parts of the URL of the request made to the HTTP server.

```
let data = Files.read(request.url.substring("/file/".length));
```

WebSocket Module

The WebSocket module implements two objects: a **WebSocket** client constructor and a **WebSocketServer** constructor.

```
import {WebSocket, WebSocketServer} from "websocket";
```

The **WebSocket** constructor is a compatible subset of the **WebSocket** object in HTML5. The **WebSocketServer** constructor implements an API based on the WebSocket client.

The WebSocket module supports WS and WSS connections.

Examples

WebSocket Client

This simple WebSocket client connects to an echo server and sends a single message; when the response is received, it closes the connection. All events are logged to the console.

```
let client = new WebSocket("ws://echo.websocket.org");
client.onopen = () => {
    console.log(`ws client open`);
    client.send("HELLO");
}
client.onmessage = msg => {
    console.log(`ws client message ${msg.data}`);
    client.close();
}
client.onclose = () => console.log(`ws client close`);
client.onerror = () => console.log(`ws client error`);
```

WebSocket Server

This simple WebSocket server listens for connections on port 10000. The server echoes back to the client each message received.

```
let server = new WebSocketServer(10000);
server.onStart = function(client) {
    client.onopen = () => console.log(`ws server client.open`);
    client.onmessage = msg => {
        console.log(`ws server client.onmessage ${msg.data}`);
        client.send(msg.data);
    };
    client.onclose = () => console.log(`ws server client.close`);
    client.onerror = () => console.log(`ws server client.error`);
};
server.onClose = () => console.log(`ws server close`);
```

The WebSocket server invokes the **onStart** event for each new connection, passing a client instance. The client instance implements the same API as the WebSocket client.

Constructors

new WebSocket(url)

Creates a WebSocket client and initiates a connection to the specified URL. If the connection is established successfully, the **onopen** event is invoked. If the connection attempt fails, the **onerror** event is invoked.

```
let client = new WebSocket("ws://echo.websocket.org");
```

new WebSocketServer(port)

Opens a new WebSocket server, listening on the specified port. When a new connection is received, the **onStart** event is invoked.

```
let server = new WebSocketServer(10000);
```

WebSocket Client Functions

close()

Closes the WebSocket client, terminating the connection to the server. No calls to the WebSocket client should be made after **close** is called.

send(data)

Transmits the data, which can be a string or an ArrayBuffer. WebSocket clients always transmit strings as UTF-8 data.

```
client.send("Hello");
client.send(new ArrayBuffer(8));
```

WebSocket Client Events

onclose()

Called when the remote endpoint closes the connection.

onerror()

Called when an error is detected, such as a dropped connection.

`onmessage(msg)`

Called when a message is received from the remote endpoint. The message data is available in **`msg.data`**. The data can be a string or an ArrayBuffer.

```
function onmessage(msg) {
    if ("String" === typeof msg.data)
        console.log("received String");
    else
        console.log("received ArrayBuffer");
}
```

`onopen()`

Called when the connection is successfully established to the remote endpoint.

WebSocket Server Functions

`close()`

Closes the WebSocket server, terminating active client connections.

WebSocket Server Events

`onStart(client)`

Called each time a new client connects to the server. The new client instance is passed to the event, which should set the event handlers needed to interact with the client.

The following trivial **`onStart`** handler waits for the first message from the client, sends a **`goodbye`** message, and then closes the connection.

```
server.onStart = function(client) {
    client.onmessage = msg => {
        client.send("goodbye");
        client.close();
    };
};
```

---

mdns Object

The **`mdns`** object implements both the client and the server functions of the Zero Configuration Networking (Zeroconf) protocol, also known as Bonjour.

```
import mdns from "mdns";
```

Examples

Announce Service

This example announces a Telnet service available on port 2323.

```
mdns.add("_telnet._tcp", System.hostname, 2323);
```

The following ends the announcement of the Telnet service.

```
mdns.remove("_telnet._tcp");
```

Monitor for Service

This example monitors the local network for available Telnet services.

```
mdns.query("_telnet._tcp" service => {
    console.log(`${service.status} "${service.keys.name}" ${service.service} at ${service.addr}.${service.port}`
});
```

The output looks like this:

```
found "Bob's Element" _telnet._tcp at 10.0.0.14:2323
lost "Bob's Element" _telnet._tcp at 10.0.0.14:2323
```

The following ends monitoring the local network for Telnet services.

```
mdns.query("_telnet._tcp");
```

Functions

`add(service, name, port, txt, ttl)`

Begins announcing availability of a new network service.

```
mdns.add("_telnet._tcp", "Bob's Kinoma Element", 2323);
```

The parameters are:

- **`service`** — A string specifying the type of service, in the format defined by the mDNS specification.
- **`name`** — A human-readable string that typically identifies the device running the service.

- **port** — The port number on which the server implementing the service is available.
- **txt** — *(Optional)* An object whose properties are key-value pairs to include in the mDNS service TXT record.

```
mdns.add("_telnet._tcp", "Bob's Kinoma Element", 2323, {status: "online", launched: Date.now()});
```

  The mDNS protocol limits the size of the TXT record. Use the **update** function to modify the TXT record after adding the service.
- **ttl** — *(Optional)* The "time to live" in seconds. The default value is 255.

query(service, callback)

Registers a callback function to be invoked when instances of the specified service are discovered or lost.

```
mdns.query("_telnet._tcp", service => {
    console.log(`${service.status} "${service.keys.name}" ${service.service}
        at ${service.addr}.${service.port}`
});
```

The **service** object contains the following properties:

- **status** — A string indicating the status:
  - **found** when the device is initially discovered
  - **update** when values in the TXT record change
  - **lost** when the device is no longer available
- **service** — A string specifying the type of the service (for example, **_kinoma_pins._tcp**)
- **name** — The human-readable name of the service
- **addr** — The IP address of the service (for example, **10.0.1.14**)
- **port** — The port number on which the service is available
- **keys.txt** — The key-value pairs provided by the device

The following is a complete service record provided by the Kinoma Element Pins sharing service.

```
{
    status: "found"
    service: "_kinoma_pins._tcp",
    name: "fakeBLL test",
    addr: "10.0.1.14",
    port: 9999,
    keys: {
        txt: {
            bll: "fakeSensor",
            uuid: "000167C3-67C3-1001-E494-00504302fe01",
            _ws: "ws://*:8900/"
        }
    }
}
```

To end notifications, call **query** without the **callback** parameter.

```
mdns.query("_telnet._tcp");
```

remove(service)

Ends the announcement of the service named. The **service** parameter string must exactly match the string used when calling **add**.

```
mdns.remove("_telnet._tcp");
```

resolv(name, callback)

Performs a one-time search for the specified device name.

```
let name = "Bob's Kinoma Element";
mdns.resolv(name, address => {
    if (address)
        console.log(`Found ${name} at IP address ${address}`);
    else
        console.log(`Unable to find ${name}`);
});
```

update(service, txt)

Changes the content of the TXT record included in the mDNS service record.

```
mdns.update("_telnet._tcp", {status: "door open"});
```

To clear the TXT record, omit the **txt** parameter.

```
mdns.update("_telnet._tcp");
```

SSDP Object

The **SSDP** object implements the server side of the Simple Service Discovery Protocol, which is used by the UPnP Device Architecture and others.

```
import SSDP from "ssdp";
```

The **SSDP** object provides the announcement of a service to the network. The service itself is provided by an HTTP server, which the application needs to implement separately.

Functions

`add(description)`

Adds an SSDP service description to the list of descriptions the SSDP object broadcasts for discovery.

```
SSDP.add({
    DEVICE_TYPE: "shell",
    DEVICE_VERSION: 1,
    DEVICE_SCHEMA: "urn:schemas-kinoma-com",
    HTTP_PORT: 10000,
    LOCATION: "/",
});
```

The **`HTTP_PORT`** and **`LOCATION`** properties define the HTTP server for clients to communicate with this service.

> **Note:** The **`SSDP`** object supports announcing multiple services at the same time.

`remove(description)`

Removes an SSDP service description from the list of descriptions the SSDP broadcasts for discovery.

```
SSDP.remove({
    DEVICE_TYPE: "shell",
    DEVICE_VERSION: 1,
    DEVICE_SCHEMA: "urn:schemas-kinoma-com",
    HTTP_PORT: 10000,
    LOCATION: "/",
});
```

All five fields in the service description must match between **`add`** and **`remove`** in order for the service to be removed.

Net Object

The **`Net`** object provides functions for working with URLs and network addresses.

```
import Net from "net";
```

Functions

`isDottedAddress(address)`

Examines the **`address`** argument to determine whether it is a valid IP address in the form **`ww.xx.yy.zz`**, such as **`10.0.1.4`**.

```
isDottedAddress("www.kinoma.com");  // false
isDottedAddress("10.0.1.2")  // true
isDottedAddress("10.0.1");  // false
isDottedAddress("10.0.1.256");  // false
```

`parseUrl(url)`

Breaks a URL into its constituent parts—**`scheme`**, **`user`**, **`password`**, **`authority`** (**`host`** and **`port`**), **`path`**, **`name`**, **`query`**, and **`fragment`**—and returns them in an object

```
let parse = Net.parseUrl("http://www.kinoma.com:123/where?command=2#frag");
// parse.scheme == "http"
// parse.port == 123
// parse.path == "where"
// parse.query == "command=2"
// parse.fragment = "frag"
```

Socket Class

The **`Socket`** class implements a non-blocking network socket API, with support for both TCP streams and UDP (IPv4 only).

```
import Socket from "socket";
```

Constructor

`new Socket(params)`

The **`params`** argument contains a dictionary of configuration values for the new socket.

```
let tcp = new Socket({host: "www.kinoma.com", port: 80, proto: Socket.TCP});
let udp = new Socket({host: BROADCAST_ADDR, port: BROADCAST_PORT,
     proto: Socket.UDP, multicast: addr, ttl: 255});
```

The following properties are supported by the dictionary:

- **`host`** — The hostname, or the host IP address in dot notation
- **`port`** — The port number to connect to
- **`proto`** — The protocol for the socket, as **`Socket.TCP`** or **`Socket.UDP`**
- **`multicast`** — *(UDP only; optional)* The interface address in dot notation

- **ttl** — *(UDP only; optional)* The maximum hop count beyond the local network

Functions

close()

Immediately disposes of the socket, releasing any associated resources.

flush()

Forces all pending data to be sent. This function is primarily for internal use; most applications do not need to flush data.

read(type, count, buffer)

Performs a non-blocking read of data from the socket. The **count** parameter specifies the number of bytes to read. The **type** parameter indicates which JavaScript object to read the data into, as either **String** or **ArrayBuffer**.

```
let str = socket.read(String, 5);
let bytes = socket.read(ArrayBuffer, 10);
let all = socket.read(ArrayBuffer, socket.bytesAvailable);
```

If **type** specifies an ArrayBuffer, an ArrayBuffer can be passed in the **buffer** argument as an optimization to minimize buffer allocations.

```
let buffer = new ArrayBuffer(10);
socket.read(ArrayBuffer, buffer.byteLength, buffer);
socket.read(ArrayBuffer, buffer.byteLength, buffer);
```

recv(bytesAvailable, buffer)

Performs a non-blocking read of data from the socket into an ArrayBuffer. **recv** is the low-level function used to implement the high-level **read** function; most applications use **read** instead.

bytesAvailable indicates the number of bytes to read. It is limited by the number of bytes to be read as indicated by **socket.bytesAvailable**.

An existing ArrayBuffer can be passed in the optional **buffer** argument as an optimization to minimize buffer allocations.

```
let buffer = socket.recv(8);
```

send(data, address)

Performs a non-blocking transmission of data on the socket. **send** is the low-level function used to transmit UDP packet data and to implement the high-level **write** function; most applications use **write** instead.

The parameters are:

- **data** — The data to transmit, in one of the following formats: an integer representing a byte value; a string; an ArrayBuffer; or an array containing integers, strings, ArrayBuffers, and arrays
- **address** — *(UDP only; optional)* A string indicating the IP address and port to send the data to

```
socket.send("a string");
socket.send(13);
socket.send(["a string", 13, 10]);
socket.send(arrayBuffer, "10.0.0.3:2882");
```

If the data cannot be completely transmitted without blocking, **send** returns 0 and no data is transmitted. If all data is successfully transmitted, **send** returns 1. If an error occurs, an exception is thrown.

Applications can check the **socket.bytesWritable** property for how many bytes the socket is able to accept for transmission.

write(item...)

Performs a non-blocking transmission of data on the socket. Each argument specifies the data to transmit, in one of the following formats: an integer representing a byte value; a string; an ArrayBuffer; or an array containing integers, strings, ArrayBuffers, and arrays

```
socket.write("a string", [13, 10]);
socket.write((buffer.length >> 8) & 0xff, buffer.length & 0xff, buffer);
```

The **write** function transmits only complete arguments. The return value indicates how many arguments were transmitted. If the data can be completely transmitted without blocking, **write** returns the number of arguments. If, for example, there are three arguments and only the first can be transmitted, **write** returns 1.

Applications can check the **socket.bytesWritable** property for how many bytes the socket is able to accept for transmission.

Values

addr

The IP address of the socket. *(Read only)*

bytesAvailable

The number of bytes ready to be read on the socket. *(Read only)*

```
let all = socket.read(ArrayBuffer, socket.bytesAvailable);
```

bytesWritable

The number of bytes that can be written to the socket. *(Read only)*

peer

The IP address and port this socket is connected to, as a string in the format **<peerAddr>:<peerPort>**—for example, **10.0.1.12:80**.

peerAddr

The IP address this socket is connected to.

peerPort

The port number this socket is connected to.

port

The port number of the socket. *(Read only)*

Events

onClose()

Called when the socket is about to be closed. This can happen as the result of an error—for example, when the device disconnects from the network.

The **onClose** function is responsible for calling **close** on the socket.

```
function onClose() {
    // Cleanup
    ...
    this.close();
}
```

onConnect()

Called when the socket successfully connects to the host specified in the dictionary passed to the constructor.

onData(buffer)

Called with bytes that have been received by the socket. The **buffer** argument contains the bytes that have been read from the socket into an ArrayBuffer.

onError()

Called when the socket encounters an error, such as loss of network connection. The application can perform any necessary cleanup.

The **onError** function is responsible for calling **close** on the socket.

```
function onError() {
    // Cleanup
    ...
    this.close();
}
```

onMessage(bytesAvailable)

Called when data is ready to be received by the socket. Most applications use **onData** instead of **onMessage**. **onMessage** is useful for applications that require more control over how the data is read.

If the **onMessage** event is not overridden, the default implementation reads the data available on the socket and generates the **onData** event. The **bytesAvailable** parameter indicates the number of bytes ready to be read from the socket.

Static Functions

static resolv(name, callback)

Performs an asynchronous name resolution. The **name** parameter is the name to be resolved. The **callback** parameter is the function to call when the name resolution is complete. The callback function takes a single argument, which is a string with the IP address the name resolved to, or **null** if the name could not be resolved.

```
Socket.resolv("www.kinoma.com" result => console.log(`resolved to ${result ? result : "ERROR"}));
```

Static Constants

Socket.TCP = "tcp"

Socket.UDP = "udp"

ListeningSocket Class

The **ListeningSocket** class extends the **Socket** class; it implements a network listener for use by network server protocols implementations that accept incoming connections, such as HTTP and WebSocket.

```
import ListeningSocket from "socket";
```

Constructor

new ListeningSocket(params)

```
sock = new ListeningSocket({port: 5151, proto: Socket.UDP});
sock = new ListeningSocket({addr: "239.255.255.250", port: 1900, proto: Socket.UDP,
    membership: ifc.addr, ttl: 2});
```

Functions

accept(socket)

Accepts an incoming connection request and returns a socket for the connection

```
let incoming = socket.accept();
```

Events

onConnect()

Called when an incoming connection request is received. Use **socket.accept** to accept the connection request and create a new socket for the connection.

SecureSocket Class

The **SecureSocket** class extends the **Socket** class; it implements the TLS protocol using a **Socket** object. **SecureSocket** is a building block for secure communication, including the HTTPS and WSS protocols.

```
import SecureSocket from "SecureSocket";
```

Constructor

new SecureSocket(params)

Extends the initialization dictionary of the **Socket** object with an **options** property to configure the TLS connection.

```
let socket = new SecureSocket({host: addr, port: port, proto: Socket.TCP,
    options: {extensions: {server_name: adr, max_fragment_length: 1024}}});
```

SecureSocket adds the following properties to the **params** dictionary:

- **protocolVersion** — The version of the TLS protocol to implement; currently supports 3.1 (0x3031), 3.2 (0x3032), and 3.3 (0x3033). Note that 3.3 is supported only on Kinoma Element. The default is 3.1.
- **cache** — If **true** (the default), enable the session cache; if **false**, disable.
- **verifyHost** — If **true**, verify the hostname in certificates. The default is **false**.
- **extensions** — An object that contains additional options:
  - **extensions.server_name** — The name of the server the client is connecting to.
  - **extensions.max_fragment_length** — The maximum fragment length of a TLS packet on this socket. The default is 16384.

Note: In the future, the **SecureSocket** object will support specifying a certification.

Connection Object

The **Connection** object contains functions related to the network and Wi-Fi configuration of the device.

```
import Connection from "wifi";
```

The Wi-Fi connection is always in one of three modes:

- *Station mode* — In this mode, the device is a Wi-Fi client, connecting to a Wi-Fi access point. This is the most common mode.
- *Access point mode* — In this mode, the device is a Wi-Fi access point for other devices to connect to. This mode is most commonly used for configuring a new device.
- Off

Functions

connect(config)

Used to select the operating mode of the Wi-Fi—station mode or access point mode—and to configure the Wi-Fi connection used in station mode.

If **config** is a number, the **connect** function selects the operating mode, as follows:

```
Connection.connect(Connection.FALLBACK);   // Access point mode
Connection.connect(Connection.NORMAL);     // Station mode
```

If **config** is a dictionary, it is used as the configuration for station mode. It also sets the operating mode to station mode.

```
Connection.connect({ssid: "myAccessPoint",
                    security: "wpa2",
                    password: "frogs",
                    hidden: false,
                    save: false});
```

The configuration properties for the station mode configuration are as follows:

- **ssid** — The name of the access point

- **bssid** — The BSSID of the access point, as a string
- **security** — The security mode to use to connect, as a string: **wpa**, **wpa2**, or **none**
- **password** — The password to use when connecting; not required if **security** is **none**
- **hidden** — A boolean indicating whether the access point is hidden
- **save** — A boolean indicating whether this connection should be remembered in the Wi-Fi environment store following a successful connection

disconnect()

If the device is in station mode, disconnects from the current Wi-Fi access point, if any; if the device is in access point mode, disables the access point.

getInterfaces()

Returns an object describing all network connections.

```
let interfaces = Connection.getInterfaces();
for (let name in interfaces) {
    let interface = interfaces[name];
    console.log(`Interface name ${name}`);
    console.log(` Connected: ${interface.UP}`);
    console.log(` Multicast supported: ${interface.MULTICAST}`);
    console.log(` Loopback: ${interface.LOOPBACK}`);
    console.log(` IP Address: ${interface.addr}`);
    console.log(` MAC address: ${interface.mac}`);
    console.log(` DNS server: ${interface.dns}`);
}
```

Kinoma Element connects to the network exclusively using Wi-Fi, so **getInterfaces** returns either 0 or 1 active network connections. When the Kinoma Element simulator is run on the desktop, more than one network interface may be included.

scan(rescan)

Returns an array of Wi-Fi access points that are visible to the device. Scan results may be cached. To force a new scan, set the optional **rescan** parameter to **true**.

```
let aps = Connection.scan();
aps.forEach(ap => {
    console.log(`Name: ${ap.ssid}`);          // "my access point"
    console.log(`bssid: ${ap.bssid}`);        // "A512F37E93BC"
    console.log(`Security: ${ap.security}`);  // "wpa", "wpa2", or "none"
});
```

stats()

Writes information (like the following) about active sockets to the console; for debugging purposes only.

```
Type   l_port    l_ipaddr    r_port       r_ipaddr    State
UDP    9999      127.0.0.1     0          0.0.0.0
UDP    6969      0.0.0.0       0          0.0.0.0
UDP    5353      0.0.0.0       0          0.0.0.0
UDP    12345     127.0.0.1     0          0.0.0.0
TCP    2323      0.0.0.0       1          0.0.0.0     Listen
TCP    10000     0.0.0.0       1          0.0.0.0     Listen
UDP    1900      0.0.0.0       0          0.0.0.0
TCP    8081      0.0.0.0       1          0.0.0.0     Listen

Total sockets:16 ==> Unused:8, TCP:3, UDP:5, RAW:0
```

Values

ip

The IP address of the active connection, as a string. *(Read only)*

```
console.log(`IP Address is ${Connection.ip}`);    // "10.0.1.3"
```

mac

The MAC address of the Wi-Fi connection, as a string. *(Read only)*

```
console.log(`MAC Address is ${Connection.mac}`);    // "A512F37E93BD"
```

rssi

The "received signal strength indicator" of the current Wi-Fi connection in dB, as a number. The range of values is -120 to 0, where a larger number indicates a stronger signal. If there is no active Wi-Fi connection, **rssi** has a value of **undefined**. *(Read only)*

```
console.log(`RSSI is ${Connection.rssi}`);     // -20
```

ssid

The name of the access point broadcast in access point mode, as a string. *(Read only)*

```
console.log(`Access point name is ${Connection.SSID}`);    // "Kinoma Element-A512F37E93BD"
```

status

The current status of the Wi-Fi connection. Although the **status** property can be written as well as read, it is not recommended that applications set this property.

```
switch (Connection.status) {
```

```
    case Connection.CONNECTED: console.log("Connected"); break;
    case Connection.DISCONNECTED: console.log("Disconnected"); break;
    case Connection.ERROR: console.log("Error"); break;
    case Connection.INITIALIZED: console.log("Initialized"); break;
    case Connection.UNINITIALIZED: console.log("Uninitialized"); break;
    case Connection.SCANNED: console.log("Scanned"); break;
    case Connection.PENDING: console.log("Pending"); break;
    default: console.log("unknown"); break;
}
```

Debug Object

The **Debug** object contains functions to assist in debugging JavaScript code.

```
import Debug from "debug";
```

Most applications do not need to use the **Debug** object directly. (IDEs, such as Kinoma Code, use the **Debug** object functions.)

Functions

gc(flag)

Turns the JavaScript garbage collector on if **flag** is **true** (the default) or off if **flag** is **false**.

```
Debug.gc(false);
...
Debug.gc(true);
```

Disabling the garbage collector in a memory-constrained device like Kinoma Element is not recommended; it may increase JavaScript memory use, leading to system instability.

login(host, name)

Establishes a debugging connection to the **xsbug** debugger running at the address specified by the **host** parameter. The optional **name** parameter specifies the name of the application being debugged.

```
let success = Debug.login("10.0.1.2", "my app");
let success = Debug.login("10.0.1.3:5002", "my app");
```

Only one debugging connection can be active at a time.

logout()

Terminates the active **xsbug** debugging session, if one is active.

report(silent)

Provides information about the current memory use of the JavaScript virtual machine.

Calling **report** with **silent** set to **false** (the default) writes a memory report like the following to the console.

```
=== heap info ===
malloc: 18672 free, 338 allocations, 15552 biggest block, 40752 by Fsk
heap2: 0x123340, 0x14fa80, 182080 remains
heap3: 0x2001f7f4, 12 remains
===
# Chunk allocation: reserved 26432 used 19768 peak 26416 bytes, 1 blocks
# Slot allocation: reserved 97520 used 95648 peak 95984 bytes, 0 free
```

Calling **report** with **silent** set to **true** returns an object with information about the current memory.

```
let info = Debug.report(true);
console.log(`Slot memory: ${info.slot}, chunk memory: ${info.chunk}`);
```

setBreakpoint(file, line)

Adds a debugging breakpoint for the specified file at the specified line number.

```
Debug.setBreakpoint("/k1/main.jsb", 10);
```

console Object

The **console** object provides functions to control the output of diagnostic information.

```
import console from "console";
```

Console output is sent to the following output locations, when active:

- Telnet
- USB serial connection
- Log file at the path **/k2/log**
- JavaScript debugging connection (Kinoma Code, Kinoma Studio, **xsbug**)

> **Note:** The console is blocked during certain operations; for example, when the application is stopped at a breakpoint in the debugger, the console will not respond.

Functions

log(params...)

Accepts one or more parameters, converts them to human-readable form, and writes them to the console output locations that are active.

The **log** function converts the following JavaScript objects to human-readable strings for output: **undefined**, **null**, **Boolean**, **Number**,

`String`, `Symbol`, `Function`, `Array`, and `Object`.

```
console.log("one", 2, {three: 3}, ["four"]);
```

Values

`enable`

If **`true`**, activates output to the console log file; if **`false`**, output to the log file is inactive.

```
console.enable = true;
```

CLI Object

The command-line interface (CLI) implements the parsing and execution of commands; the **`CLI`** object is used by Telnet and USB to execute commands.

```
import CLI from "CLI";
```

Applications do not usually invoke the **`CLI`** object directly. However, for debugging purposes it can be useful to execute a command programmatically—for example, to display system state at a specific point in execution.

Functions

`evaluate(line)`

Takes a single command line as input in the **`line`** parameter. If the command line is successfully parsed, it is executed and any output is sent to the console.

```
CLI.evaluate("modules");  // loaded modules
CLI.evaluate("scan");     // visible Wi-Fi access points
CLI.evaluate("printenv"); // default environment store
```

CoAP module

The CoAP module implements the Constrained Application Protocol (https://tools.ietf.org/html/rfc7252), a lightweight alternative to HTTP. CoAP runs over UDP, so it uses less network bandwidth and can be implemented efficiently on low cost hardware like Kinoma Element.

Currently there is both Client and Server implemented. However, in both cases only single packet delivery is implemented.

```
import CoAP from 'coap'
```

Examples

A simple CoAP client that connects to the public test server at eclipse.org, and gets a response.

Example client GET

```
import { CoAP } from 'coap';

const main = {
    onLaunch(){
    let client = new CoAP.Client();
    this.client = client;
    client.get("coap://californium.eclipse.org:5683", response => {
        trace(`response code ${response.code}, payload ${response.payload.byteLength} bytes\n`);
    });

    }
}

export default main;
```

Another simple CoAP client that connects to the public test server at eclipse.org. This one posts some data, and then logs the response. It also handles the acknoledgement from the server, as well as an error response.

Example client POST

```
...
    const client = new CoAP.Client();
    this.client = client;
    let payload = ArrayBuffer.fromString("Hello");

    client.onAck = request => trace(`ack for request ${request.url}\n`);
    client.onError = (error, request) => trace(`error ${error} for request ${request.url}\n`);

    client.post("coap://californium.eclipse.org:5683", payload, response => {
        trace(`response code ${response.code} \n`);
    });
...
```

In this example the acknowledgement and error callbacks are set to log their responses. The payload for a POST request must be an arrayBuffer.

Example Server request callback

```
...
    const server = new CoAP.Server({port: 5683});
    this.server = server;
```

```
    server.bind('/test', session => {
        // handle request
        const response = session.createResponse();
        response.setPayload('HELLO', 'text/plain');
        session.send(response);
    });

    server.start();
...
```

This example shows a simple returned message, when it's resource path is requested. The format of the request would be `'coap://DEVICE-IP:5683/test'`.

Constructors

`new CoAP.Client()`

```
const client = new CoAP.Client();
```

Creates a new CoAP Client with no outstanding requests. Make requests with the client using `get`, `post`, `put`, `delete_`, `observe`, and `send` functions.

`new CoAP.Server()`

```
const server = new CoAP.Server({ port: 5683 });
```

Creates a new CoAP Server with the reccomended port number. Use the `.start()` method to run the server and the `.bind` method to handle resource requests.

Functions - CoAP Client

`get(url, callback)`

Initiates a CoAP request using the GET method.

```
client.get("coap://californium.eclipse.org:5683", response => {
    trace(`response code ${response.code}, payload ${response.payload.byteLength} bytes\n`);
});
```

The get function implementation calls the send function. See the description of the send function for details.

`post(url, payload, callback)`

Initiates a CoAP request using the POST method with the payload as the message body.

```
let payload = ArrayBuffer.fromString("Hello");
client.post("coap://californium.eclipse.org:5683", payload, response => {
    trace(`response code ${response.code} \n`);
});
```

The post function implementation calls the send function. See the description of the send function for details.

`put(url, payload, callback)`

Initiates a CoAP request using the PUT method with the payload as the message body.

```
let payload = ArrayBuffer.fromString("Hello PUT");
client.put("coap://californium.eclipse.org:5683", payload, response => {
    trace(`response code ${response.code}, payload ${response.payload.byteLength} bytes\n`);
});
```

The put function implementation calls the send function. See the description of the send function for details.

`observe(url, callback)`

Initiates a CoAP request using the GET method with the observe option enabled. The object is returned which will be required to cancel the observation after that.

```
let observation = client.observe("coap://californium.eclipse.org:5683", response =>
    trace(`observe code ${response.code}, payload ${response.payload.byteLength} bytes`);
});
```

`cancel(observation)`

Cancel the observe request. You need to keep the reference of the return value of `observe()` to be cancelled, then pass that value to cancel().

```
client.cancel(observation);
```

`send(url, payload, callback)`

The url parameter is either a string indicating the URL of the address to send the CoAP request, or an object with properties that includes the URL together with other settings used to construct the CoAP message. The CoAP Client implements the following properties.

- `confirmable` - a boolean that indicates if this request must be confirmed. The default is true. The confirmable property may instead be set using the type parameter with values of Type.Con and Type.Non.
- `method` - the method of the request as a value from the `CoAP.method` enumeration. The default method is `GET`.
- `observe` - a boolean that indicates if this is a an observe request. The default value is false.
- `options` - an array of array which is pair of an option, from the `CoAP.Option` constants and its value. i.e.

```
[[Option.ContentFormat, ContentFormat.Json], [Option.Observe, Obverve.Register], ...]
```

- **type** - a value from the CoAP.Type constants indicating the type of CoAP packet to send
- **url** - the address to send the CoAP request. This is the only require property.

The following examples show using the url parameter object to configure message parameters.

```
client.send({url: "coap://10.0.1.32:5683/colors", observe: true}, undefined, response => console.log("colors replied"));
client.send({url: "coap://10.0.1.32:5683/message", confirmable: false, method: CoAP.Method.DELETE}, new ArrayBuffer(1));
client.send({url: "coap://10.0.1.32:5683/message", options: [[CoAP.Option.ContentFormat, CoAP.ContentFormat.Json], [CoAP.Opti
on.Observe, CoAP.Obverve.Register]]}, ArrayBuffer.fromString(JSON.stringify({hello: "world"})));
```

The payload is an ArrayBuffer sent as the message body. Pass undefined for the payload when there is no message body.

The callback parameter, when present, is a function to call when a message reply is received. The callback is called once for each reply received, which may be multiple times for an observable request. If no callback is provided, the onResponse event on the CoAP client instance is invoked.

The callback receives the response as its parameter. The response is an object containing properties corresponding to the content of the response.

```
client.send({url: "coap://10.0.1.32:5683/test"}, undefined,
    response => trace(`test replied with code ${response.code}\n`));
```

The following is a list of the properties contained in the response object.

- **code** - the CoAP message response code (https://tools.ietf.org/html/rfc7252#section-12.1.2) indicating the success or failure of the message
- **messageId** - the ID of the CoAP message
- **options** - the options specified in the message
- **payload** - the payload of the response as an ArrayBuffer
- **token** - the token generated by the CoAP client when sending the message
- **type** - the type of the CoAP message, from the CoAP.Type enumeration
- **version** - the version of the CoAP message, from 0 to 3 inclusive. The Kinoma Element implementation expects the version value to be 1.

The send function returns the newly created request object.

```
let request = client.send("coap://10.0.1.32:5683/test");
console.log(`request.messageId: ${request.messageId}`);
console.log(`request.token: ${request.token}`);
console.log(`request.url: ${request.url}`);
console.log(`request.host: ${request.host}`);
console.log(`request.port: ${request.port}`);
// etc.
```

The request object contains properties representing the complete content of the CoAP message. The application should not modify the request object.

> **Note:** Applications typically use delete_, get, send, put, post, and observe rather than send. Use send to configure requests in specialized scenarios which require more control over the message format.

Events - CoAP Client

onAck

The onAck event is invoked when the CoAP client determines that a request marked as confirmable has been successfully delivered.

```
client.onAck = request => console.log(`confirmed deilvery of message with token ${request.token}`);
client.get("coap://10.0.1.32:5683/test");
```

> **Note:** The acknowledgement that a confirmable request has been successfully delivered is not the response to the message. The message response is delivered to the request callback or the client onResponse event.

onError(error, request)

The onError event is invoked when an error occurs processing a client request. The error parameter is a string describing the error, for example "resolve error". The request parameter is the request object returned by send.

```
client.onError = (error, request) => console.log(`onError "${error}" on message with token ${request.token}`);
```

onResponse(response, request)

The onResponse event is invoked when a reply is received by a request issued on the CoAP client for which no callback function was provided.

The request parameter is the request object returned by send.

```
client.onResponse = (response, request) =>     console.log(`received response to message with token ${request.token}`);
let request = client.send("coap://10.0.1.32:5683/test");
console.log(`sent message, token ${request.token}`);
request = client.send("coap://10.0.1.32:5683/test");
console.log(`sent another message, token ${request.token}`);
```

Values - CoAP Client

CoAP.Type

The object request/response type. Con - Confirmable request. Non - non-confrimable request. Ack - Acknowledgement response. Rst - Reset reponse.

```
const Type = {
    Con: 0,
    Non: 1,
    Ack: 2,
    Rst: 3
```

```
};
```

CoAP.Method

The object request type.

```
const Method = {
    GET: 1,
    POST: 2,
    PUT: 3,
    DELETE: 4
};
```

CoAP.Option

CoAP reqeust options.

```
const Option = {
    // RFC 7252 core
    IfMatch: 1,
    UriHost: 3,
    ETag: 4,
    IfNoneMatch: 5,
    UriPort: 7,
    LocationPath: 8,
    UriPath: 11,
    ContentFormat: 12,
    MaxAge: 14,
    UriQuery: 15,
    Accept: 17,
    LocationQuery: 20,
    ProxyUri: 35,
    ProxyScheme: 39,
    Size1: 60,

    // draft-ietf-core-block-15
    Block2: 23,
    Block1: 27,
    Size2: 28,

    // draft-ietf-core-observe-14
    Observe: 6
};
```

CoAP.ContentFormat

The format of the reponse object.

```
const ContentFormat = {
    PlainText: 0,
    LinkFormat: 40,
    Xml: 41,
    OctetStream: 42,
    Exi: 47,
    Json: 50
};
```

CoAP.Observe

Set observe flag to send/enable observation packets.

```
const Observe = {
    Register: 0,
    Deregister: 1
};
```

MQTT module

The MQTT module exports the MQTT object. Currently implemented is the Client object, and its behaviors.

```
import MQTT from 'mqtt';
```

MQTT follows the publish/subscription model of communication. A server acts as a gateway for the communication, where the clients connect, sub/unsubscribe, publish, and receive messages from one another, through a common topic. There is no required mechanism for creating a new topic; any connection will create the specified topic if it does not exist.

Quality of service or **qos** is reffered to several times, the meaning is descirbed in this table.

Value	Description
0	At most one delivery
1	At least one delivery
2	Exactly one delivery

Examples

MQTT Client

The following creates and connectes an MQTT Client. The **onMQTTClientConnect** behavior is defined below as well. This will help us determine whether the connection was made. Here, we are using the IoT eclipse MQTT testing web service.

```
let client = new MQTT.Client();
this.client = client;
client.behavior = {
    onMQTTClientConnect(client, returnCode, sessionPersist){
        trace(`client: ${JSON.stringify(client)}, returnCode: ${returnCode}, sessionPersist: ${sessionPersist}\n`);
    }
}
```

```
client.connect('iot.eclipse.org', 1883, {
    keepAlive: 20,
        will: {
            topic: "kinoma/status", // required
            qos: 1,
            retain: true,
        }
});
```

> **Note:** `this` is used to keep the client referenced by the application, and prevent it from be taken away by the element's aggressive garbage collector.

Constructor

`new MQTT.Client(clientIdentifier, cleanSession)`

```
this.client = new MQTT.Client();
```

The parameters for client are optional. The **`clientIdentifier`** string will be used by the server to distinguish the connection. If supplied along with the boolean **`cleanSession`** being false, the previous connection will be restored. If not supplied, a randomly generated identifier will be used and a fresh session will be started.

Functions

`connect(host,port,options)`

Invokes an MQTT connection from your client to a server. **`host`** is the target url, **`port`** is typically 1883, or 8883 for a TLS encypted connection. **`options`** is an object which includes these parameters :

- **`keepAlive`** - The interval number in seconds that the client uses to send PING message to let the server know the client is alive. The default is 60.
- **`username`** - The username string for authentication.
- **`password`** - The password string for authentication.
- **`will`** - The information about the **last will**. If supplied, the last will will be set.
- **`will.topic`** - The topic string to which the last will may be published. This is required in the last will.
- **`will.data`** - The message which will be published as a last will. It is a string or arrayBuffer.
- **`will.qos`** - The quality of service number to be used when the last will message is published.
- **`will.retain`** The boolean retain flag to be used when the last will message is published.

Calling this method will invoke **`onMQTTClientConnect`**.

`disconnect()`

Close the connection to the MQTT server. Calling this method will invoke **`onMQTTClientDisconnect`**.

`publish(topic, data, qos, retain)`

Sends a message to all topic subscribers. **`topic`** is the topic string. **`data`** is either a string or arrayBuffer message. **`retain`** by default is false. A retained message will be stored by the server, and is sent to all new client connections. Calling this method will invoke **`onMQTTClientPublish`**.

`subscribe(topicFilter, qos)`

Links the client to a specified topic. **`topicFilter`** is a string or string array, specifying topic(s) and/or filter(s). Topics are specified via path strings (i.e. 'topic/example/thing'). The '#' (i.e. 'topic/#') and '+' (i.e. 'topic/+/example) symbols are multiple, and single level wildcards, respectively. Calling this method will invoke **`onMQTTClientSubscribe`**.

`unsubscribe(topicFilter)`

Unlinks the client to the specified topic(s) and/or filter(s). **`topicFilter`** is a string or string array. Calling this method will invoke **`onMQTTClientUnsubscribe`**.

Behaviors

The following behaviors can be custom defined by adding them to the client behavior object. In all cases **`client`** is the client data object and **`packetID`** is the ID of the received MQTT packet.

`onMQTTClientConnect(client, returnCode, sessionPersist)`

Called when the connection is made. **`returnCode`** specifies the server response, 0 being accepted, else refused. **`sessionPersist`** is a boolean which indicates whether the server already has a previous persistant session from the client and remebers it's subscriptions.

```
onMQTTClientConnect(client, returnCode, sessionPersist){
    ...
}
```

`onMQTTClientSubscribe(client, packetId, result)`

Called when a subscription is made.

`result` is a boolean of whether the subscription was successful.

```
onMQTTClientSubscribe(client, packetId, result) {
    ...
},
```

`onMQTTClientUnsubscribe(client, packetId)`

Called when an unsubscription is made.

```
onMQTTClientUnsubscribe(client, packetId) {
```

```
    ...
},
```

onMQTTClientPublish(client, packetId)

Called when a message is published.

```
onMQTTClientPublish(client, packetId) {
    ...
},
```

onMQTTClientMessage(client, topic, payload, qos, retain, dup)

Called when a message is received. **topic** is the channel through which the message was recieved. **payload** is the message. The **retain** flag denotes whether it was a retained message. The **dup** flag indicates if the messgae was sent more than once.

```
onMQTTClientMessage(client, topic, payload, qos, retain, dup) {
    ...
},
```

onMQTTClientDisconnect(client, cleanClose)

Called on disconnection. **cleanClose** flag dnotes whether the connection was exited gracefully or not.

```
onMQTTClientDisconnect(client, cleanClose) {
    ...
},
```

onMQTTClientError(client, err, reason)

Called when receiving an error message. **err** is the error object. **reason** is a string message that attempts ot explain the error.

```
onMQTTClientError(client, err, reason) {
    ...
}
```

Kinoma Element CLI Reference

The Kinoma Element command-line interface (CLI) is available over Telnet and USB.

Command-line arguments are separated by spaces. Arguments that contain spaces are enclosed in double quotes.

```
cd k1
cat "a file.txt"
```

In syntax shown in this section, angle brackets indicate required parts of the syntax and square brackets indicate optional parts.

cat <path>

Displays the contents of the file at the location specified by **path**.

```
[jphAir.local]$ cat /k1/mc.env
.ver\01\0FW_VER\00.99.7\0ELEMENT_SHELL\01\0UUID
\025FB6A35-95CB-1A04-82EE-000000000000\0XSBUG_HOST\010.0.1.69:5003\0\0
```

cd [path]

Changes the working directory to **path**, or to the root directory if **path** is not provided. The path can be relative or absolute.

```
[ajcElement]$ pwd
/
[ajcElement]$ cd k0
[ajcElement]$ pwd
/k0
```

connect <ssid> <security> <password> <hidden> <save>

Connects to the specified Wi-Fi access point. The arguments are:

- **ssid** — The name of the Wi-Fi access point
- **security** — **wpa2** for access points using the WPA2 protocol or **none** for access points with no security
- **password** — The access point's password, or **undefined** for open access points
- **hidden** — **true** if the access point is hidden
- **save** — **true** to save this connection to the **wifi** environment variables group so that it will be used for automatic connections in the future

```
[jphAir.local]$ connect "KinomaWPA" wpa2 cube-able-stub false false
[jphAir.local]$ connect "Marvell Cafe" none "" false true
```

date

Displays the current date and time as returned by **Date()**.

```
[ajcElement]$ date
Tue, 19 Apr 2016 22:07:20
```

eval <expression>

Runs the JavaScript **eval** function on the specified expression.

```
[jphAir.local]$ eval 12+13
```

```
[jphAir.local]$ eval 12+13
25
[jphAir.local]$ eval "console.log('Hello, world.')"
Hello, world.
```

gc

Runs the JavaScript garbage collector to free any unreferenced objects, including modules.

getenv <name>

Displays the environment variable **name** from the default environment variables group.

hexdump <path>

Displays the contents of the file at the location specified by **path** in hexadecimal format.

```
[jphAir.local]$ hexdump /k1/mc.env
2e 76 65 72 00 31 00 46 57 5f 56 45 52 00 30 2e
39 39 2e 37 00 45 4c 45 4d 45 4e 54 5f 53 48 45
4c 4c 00 31 00 55 55 49 44 00 32 35 46 42 36 41
```

hostname

Displays the hostname of the Kinoma Element.

```
[ajcElement]$ hostname
ajcElement
```

ip

Displays the IP address of the Kinoma Element.

```
[ajcElement]$ ip
10.85.20.157
```

launch <path>

Launches the application at the specified path.

load <module>

Loads the specified module using **require.weak()**.

ls [path]

Lists the files and directories in **path**. If **path** is not specified, the working directory is used.

```
[jphAir.local]$ ls /k1
mc.env
wifi
```

mac

Displays the MAC address of the Kinoma Element.

modules [prefix]

Displays the currently loaded modules. Use (for example) **modules te** to display only loaded modules that begin with **te**.

```
[jphAir.local]$ modules
 1: application
 2: board_led
 3: CLI
 4: console
 5: inetd
 6: mdns
 7: pinmux
```

netstat

Displays details about the current network status.

```
[jphAir.local]$ netstat
Type    l_port        l_ipaddr    r_port        r_ipadd   State
CP        8081        0.0.0.0            1        0.0.0.0   Listen
        969         0.0.0.0            0        0.0.0.0
UDP        5353        0.0.0.0            0        0.0.0.0
UDP        12345        127.0.0.1          0        0.0.0.0
UDP        12346        127.0.0.1          0        0.0.0.0
UDP        5353        0.0.0.0            0        0.0.0.0
TCP        2323        0.0.0.0            1        0.0.0.0   Listen
TCP        10000        0.0.0.0            1        0.0.0.0   Listen
UDP        1900        239.255.255.250    0        0.0.0.0
UDP        49158        0.0.0.0            0        0.0.0.0
TCP        8081        0.0.0.0            1        0.0.0.0   Listen
```

printenv [group] [encrypted]

Displays all the environment variables in the specified environment variables group. If no environment variables group is specified, the environment variables of the default group are displayed.

Set **encrypted** to **true** if the environment variables are encrypted.

```
[jphAir.local]$ printenv
FW_VER=0.99.7
ELEMENT_SHELL=1
UUID=25FB6A35-95CB-1A04-82EE-000000000000
```

pwd

Displays the working directory.

```
[jphAir.local]$ pwd
/k1
```

quit

Quits all currently running applications.

reboot [mode]

Restarts Kinoma Element.

If **mode** is **true**, the device reboots immediately without tearing down the current running services.

reconnect <mode>

If **mode** is **1**, Kinoma Element attempts to reconnect to the current Wi-Fi access point. If **mode** is **2**, Kinoma Element disconnects from the Wi-Fi access point and enters UAP (Micro Access Point) mode.

rename <from> <to>

Renames the file specified by **from** to the name specified by **to** in the working directory.

report

Displays memory usage statistics of the JavaScript virtual machine.

```
[jphAir.local]$ report
=== heap info ===
malloc: 29360 free, 365 allocations, 26240 biggest block, 40928 by Fsk
heap2: 0x126740, 0x151a80, 176960 remains
heap3: 0x2001f7f4, 12 remains
===
# Chunk allocation: reserved 26432 used 24304 peak 26416 bytes, 1 blocks
# Slot allocation: reserved 78832 used 72672 peak 78416 bytes, 0 free
```

Note: The Kinoma Element simulator displays only the last two lines (chunk and slot allocation).

rm <path>

Deletes the file specified by **path**.

rmdir <path>

Deletes the directory specified by **path**.

saveenv

Writes the default environment variables group to storage.

scan [flush]

Performs a scan for Wi-Fi access points visible to Kinoma Element and displays results. If the **flush** argument is missing, the scan results are from the Kinoma Element Wi-Fi access point cache; if it is set to **true**, the cache is emptied and a full rescan is performed.

setenv <name> <value>

Sets the environment variable **name** to **value** in the default environment variables group.

shutdown [force]

Turns off Kinoma Element. If **force** is present and set to **true**, Kinoma Element is immediately restarted without first terminating any active applications or network services.

timestamp

Display the timestamp of the Kinoma Element firmware build.

unsetenv <name>

Removes the environment variable **name** from the default environment variables group.

update [target] [do-not-update]

Begins the Kinoma Element firmware update process. This command does not check to see if an update is needed, so it always performs an update to the latest firmware.

- **target** — **ELEMENT_FIRMWARE_RELEASE** (the default, and the recommended value for most developers), **ELEMENT_FIRMWARE_SMOKE**, or **ELEMENT_FIRMWARE_QA** .
- **do-not-update** — If set to **true**, the firmware files are downloaded but not installed. This is primarily useful for debugging the firmware update process.

version

Displays the firmware version number and the timestamp of the Kinoma Element firmware build.

```
[ajcElement]$ version
1.2.0 (Sun Mar 27 2016 05:25:34 GMT ())
```

xsbug [host]

Connect to the standalone **xsbug** JavaScript debugger. This command is useful in advanced debugging scenarios but is not necessary when working with the Kinoma Code IDE.

If **host** is not provided, the environment variable **XSBUG_HOST** from the default environment variables group is used.

```
[jphAir.local]$ xsbug 10.0.1.69:5003
```
