Accessors Tutorial

By Christopher Brooks, Edward A. Lee, and Beth Osyk
Version 1.0, Dec. 5, 2017

Accessors are a technology for making the Internet of Things accessible to a broader community of citizens, inventors, and service providers through open interfaces, an open community of developers, and an open repository of technology. Accessors enable composing heterogeneous devices and services in the Internet of Things (IoT).

An **actor** is a software component that reacts to streaming input events and produces streaming output events. An **accessor** is an actor that wraps a (possibly remote) device or service in an actor interface. An **accessor host** is a service running in the network or on a client platform that hosts applications built as a composition of accessors that stream data to each other. The host is like a browser for things. Such applications are called **swarmlets** in this document.

Accessors embrace **concurrency**, **atomicity**, and **asynchrony**. The actor model, which governs interaction between accessors, permits accessors to execute concurrently with segregated private data and a message-passing interface for interaction. Internally, many accessors use **asynchronous atomic callbacks** (**AAC**) to invoke remote services and handle responses asynchronously and atomically.

Accessors are defined in a JavaScript file that includes a specification of the **interface** (inputs, outputs, and parameters) and an implementation of the **functionality** (reactions to inputs and/or production of outputs). Any JavaScript file that conforms with the **accessor specification** defines an **accessor class**.

The TerraSwarm accessor library provides a collection of example accessors. This library is maintained via an SVN repository that permits many contributors to add accessors to the library.

An **instance** of an accessor is created by a **swarmlet host** that evaluates the JavaScript in the accessor definition. At this time, there are at least three accessor hosts compatible with **accessor specification 1.0**:

- A browser host, which allows inspection of the accessor, and if the accessor is suitable for execution in a browser, interactive invocation of the accessor.
- A Node.js host, an interactive program that runs in Node.js that allows instantiation and execution of accessors.
- CapeCode, which supports composition of accessors with visual block diagrams and provides a large library of actors that the accessors can interact with.

1. Table of Contents

2. Minimal Accessor Example

A minimal accessor that takes a numeric input, doubles it, and sends the result to an output is given below:

```
exports.setup = function () {
    this.input('input');
    this.output('output');
};
exports.initialize = function () {
    var self = this;
    this.addInputHandler('input', function () {
        self.send('output', self.get('input') * 2);
    });
};
```

This accessor has two parts, an interface definition in the `setup()` function, and a specification of the function to be performed. The function to be performed is given by an input handler function.

Notice the commonly used JavaScript idiom, where the value of the variable `this` is captured in a variable called `self` (this name is arbitrary), and then within the callback function, `self` is used instead of `this`. In JavaScript, when a function is invoked, the value of the variable `this` is the object on which the function is invoked. The `setup` and `initialize` functions are invoked by the host on the accessor, which has functions `input`, `output`, etc., defined. But the input handler function is not assured of being invoked on the accessor. Any callback function that the accessor defines, such as the function passed to `addInputHandler` above, should be written in such a way that it will work no matter what object the host invokes it on. A browser host, for example, may invoke it with `this` equal to the window object. To understand the `this` keyword in Java, see

- http://www.quirksmode.org/js/this.html
- http://javascriptissexy.com/understand-javascripts-this-with-clarity-and-master-it/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

More interesting examples are found on the TerraSwarm accessor repository.

3. Interface Definition

An **accessor interface** specifies what it requires of an accessor host and what its inputs, outputs, and parameters are. An accessor interface definition may include the following elements in the accessor specification file:

- require: Modules required by the accessor.
- An `exports.setup()` function that can invoke the following functions:
    - input: Specify an input to the accessor.
    - output: Specify an output from the accessor.
    - parameter: Specify a parameter for the accessor.

- implement: Declare that the accessor implements another interface.
- extend: Declare that the accessor extends another accessor.

Inputs, outputs, and parameters may optionally have specified data types. The `exports.setup()` function may also invoke `instantiate()` and `connect()` as explained below in the composition section. All of the above except `require()` are functions of the accessor, so they should be invoked with the prefix `this` as illustrated above. The `require()` function, on the other hand, is a top-level function that can be invoked from anywhere. See Top-Level Functions below for a complete list of such functions.

3.1. Inputs

An accessor may define any number of inputs in the body of its setup() function using:

```
input(<string> name, [options]):
Create an input with the specified name and options.
```

For example:

```
exports.setup = function() {
    this.input('bridgeIPAddress', {
        'type':'string',
        'value':'localhost'
    });
}
```

The **`input()`** function takes one or two arguments, a name (a required string, recommended to be camelCase with a leading lower-case character) and an object giving any of the options:

type	The data type of the input. If this is not specified, then any valid JavaScript value may be provided as an input. If it is specified, it must be one of the valid data types shown below.
value	A default value for the input. Specifying a default value ensures that **get**() will not return null (see below). In the Cape Code Host, an input will be a Ptolemy PortParameter if the input has a **value** option.
options	An array of possible values for this input. A user interface may list these possible values, and a host may restrict the values to be one of these values.
visibility	A hint to the host of whether this input should be made visible to casual users. This is a string that is one of "full", "none", "expert", or "notEditable", meaning "full visibility", "no visibility", "expert visibility", or "visible, but not editable". This option is useful for accessors that extend other accessors but do not want to make the inputs of the base accessor visible in the derived accessor.

The name is required to be distinct from other inputs, outputs, and parameters. It may match the name of an input in a base accessor (specified by either extend() or implement()), in which case the options will override those specified in the base.

Getting Inputs

The current value of an input may be retrieved in the accessor's JavaScript code using the **get**() function, as in the following example:

```
var address = this.get('bridgeIPAddress');
```

The **get**() function takes one argument, the input name (as a string). If the input definition includes a **value** option, then **get**() will not return null unless the accessor has specifically been sent a null. If you call **get**() before the accessor host has provided any input value, then the returned value will be that given in the **value** option. Subsequently, the returned value will be the *most recent* value provided by the accessor host.

If no **value** option is given, then **get**() will return null if called before the host has provided an input. In addition, **get**() may return null if there is no *current* input value provided by the host. This enables the host to fire the accessor while providing some and not all of the inputs. Thus, it is good programming practice to test the return value of **get**() for null if no **value** attribute is provided.

Sending to Inputs

An accessor may use the **send**() function to send values to its own inputs. Typically, this would be done in a callback function that is triggered by some external event. It could also be done in accessor that extends another accessor and wishes to set the value of an input in the base accessor. A side effect of sending a value to its own input port is to trigger an input handler given using **addInputHandler**().

For example, suppose that the following line is executed in a callback:

```
this.send('bridgeIPAddress', '127.0.0.1');
```

This will send the string "127.0.0.1" to the bridgeIPAddress input. After the callback function has completed execution, the accessor host will call any input handler that has been registered for this port using **addInputHandler**(). In the body of the handler, the accessor can call **get**() to retrieve the value of the input.

If an accessor sends a value to its own input in an input handler or in the fire() function, the updated value of that input will not be available until the *next* reaction. An input value remains constant throughout a reaction.

If you send a null value to an input, then **get**() on that input will return null on the next invocation, even if a **value** attribute is given for that input.

Setting the Default Value of an Input

As shown above, if a 'value' option is given when defining an input, then the input will have a default value. You can change that default value (or add one to an input that did not previously have one) using the **setDefault**() function. For example:

```
this.setDefault('bridgeIPAddress', '127.0.0.1');
```

Unlike using **send**(), however, this will not trigger execution of any input handler. Also unlike **send**(), the host may treat this new default value as a persistent value, to become a permanent default, even in future executions of the same swarmlet.

Setting the default in this way can be useful in an accessors that extends another, but wishes to change the default value provided by the base accessor.

Input Handlers

- *<handle>* **addInputHandler**(*<string> input*, *function*, [*arguments*]): Specify a function to invoke when the input with name *input* (a string) receives a new input value. If additional arguments are specified, those will be passed to the function when it is invoked. A function may be added more than once, possibly with different arguments, in which case it will be invoked more than once. If the first argument is null, or if no name argument is given and the first argument is a function, then the specified function will be invoked when *any* new input arrives. This function returns a handle that can be used to call **removeInputHandler**(). Note that the handler function is not limited to observing the input that triggers its execution. It can call get() on any input. See Input.
- **removeInputHandler**(*handle*): Remove the callback function with the specified handle (returned by **addInputHandler**()).

Example:

```
exports.initialize = function() {
  // Specify a function to invoke when a new input arrives on myInput.
  this.addInputHandler("myInput", function () {
    ...
  });
  // Specify a function to invoke when a new input arrives on any
input.
  this.addInputHandler(function() {
    ...
  });
};
```

Note that when the function is invoked, it will be invoked in a context where 'this' will be some arbitrary object as determined by the host. If you need to invoke accessor functions (such as `get` or `send`) in your handler, then you will need to capture the value of 'this' using the common JavaScript idiom, as illustrated in this example:

```
exports.initialize = function() {
  // Capture the value of 'this':
  var self = this;
  this.addInputHandler("myInput", function () {
    self.send('outputName', 'hello world');
  });
};
```

Alternatively, you can bind the function to 'this' as follows:

```
exports.initialize = function() {
  this.addInputHandler("myInput", myFunction.bind(this));
};
function myFunction() {
  this.send('outputName', 'hello world');
}
```

Binding the function ensures that whenever it is invoked, from any context, the value of 'this' will the value specified in the binding.

3.2. Outputs

An accessor may define any number of outputs in the body of its `setup()` function using

- **output**(*<string> name*, [*options*]): Create an output with the specified name and options.

For example:

```
exports.setup = function() {
    this.output('price', {
        'type':'number'
    });
}
```

The **output**() function takes one or two arguments, a name (a required string, recommended to be camelCase with a leading lower-case character) and an object with any of the following options:

type	The data type of the output. If this is not specified, then any valid JavaScript value may be sent to the output. If it is specified, it must be one of the valid data types.
spontaneous	true or false. A spontaneous output is produced by an asynchronous callback rather than as a response to an input. Every directed cycle in a connectivity graph must contain at least one spontaneous output or there will be a deadlock due to a causality loop. A spontaneous output port must both be declared and have input handler invoke setTimeout(). See Action Functions and Spontaneous.

The name is required to be distinct from other inputs, outputs, and parameters. It may match the name of an output in a base accessor (specified by either extend() or implement()), in which case the options will override those specified in the base.

Sending to Outputs

An accessor may use the **send**() function to send values via its output ports. This may be done in a callback function, such as a function passed to **setTimeout** (see Top-Level Java Script Functions) or **addInputHandler**(see). For example, suppose that the following line is executed in a callback:

```
this.send('price', 42);
```

This will send the number 42 to the price output. If you send a null value to an output, and that output is connected to an input *x* of a downstream accessor, then that accessor's input handler will get null when it invokes **get**('x').

Note that if the output data type is JSON, then the JavaScript value will be converted to a JSON string before sending. This can be confusing. Specifically, if you want to send, say, `{"foo": 42}`, then you should do

```
this.send('output', {"foo": 42});
```

and not

```
this.send('output', '{"foo": 42}');
```

In other words, do not convert your object to a JSON string yourself.

If you call **send**() multiple times before the receiving accessor has a chance to react to the data, then the data will be queued and will cause multiple reactions of the downstream accessor, one for each **send**().

3.3. Parameters

An accessor may specify parameters in the body of the `setup()` function by invoking:

- **parameter**(*<string> name*, [*options*]): Create the parameter. Options are optional.

A parameter, unlike an input, is not expected to change value during the execution of a swarmlet.

For example, the following accessor will send the value of the *foo* parameter to its output *bar* once:

```
exports.setup = function() {
  this.parameter('foo', {'value':'y', 'type':'string'});
  this.output('bar');
}
exports.initialize = function() {
  this.send('bar', this.getParameter('foo'));
}
```

The **parameter**() function takes one or two arguments, a name (a required string, recommended to be camelCase with a leading lower-case character) and an object with any of the following options:

type	The data type of the parameter. If this is not specified, then any valid JavaScript value may be provided as a value. If it is specified, it must be one of the valid data types.
value	The default value for the parameter. Specifying a default value ensures that **getParameter**() will not return null (see below).
options	An array of possible values for this parameter. A user interface may list these possible values, and a host may restrict the values to be one of these values.
visibility	A hint to the host of whether this parameter should be made visible to casual users. This is a string that is one of "full", "none", "expert", or "notEditable", meaning "full visibility", "no visibility", "expert visibility", or "visible, but not editable". This option is useful for accessors that extend other accessors but do not want to make the parameters of the base accessor visible in the derived accessor.

The name is required to be distinct from other inputs, outputs, and parameters. It may match the name of a parameter in a base accessor (specified by either extend() or implement()), in which case the options will override those specified in the base.

Getting Parameters

The current value of a parameter may be retrieved in the accessor's JavaScript code using the **getParameter**() function, as in the following example:

```
var address = this.getParameter('bridgeIPAddress');
```

The **getParameter**() function takes one argument, the parameter name (as a string).

If no **value** option is given, then **getParameter**() will return null if the host has not provided a parameter value.

Setting Parameters

The current value of a parameter may be set in the accessor's JavaScript code using the **setParameter**() function, as in the following example:

```
this.setParameter('bridgeIPAddress', '128.32.12.0');
```

This might be done, for example, in an accessor that extends another to change the default value of the parameter.

3.4. Implement

An accessor may specify that it implements an interface or the interface of another accessor in its `setup()` function by invoking:

- **implement**(*<string> name*): Implement an interface with the specified name.

This has the effect of executing the `setup()` function of the specified interface or accessor, so any inputs, outputs, and parameters defined by the specified interface or accessor will also be inputs, outputs, and parameters of this accessor.

For example, suppose an interface `MyInterface` has the following:

```
exports.setup = function() {
  this.parameter('foo', {'value':42});
  this.output('bar');
}
```

Then if an accessor specifies the following:

```
exports.setup = function() {
  this.implement('MyInterface');
}
```

then it will also have a parameter named 'foo' and an output named 'bar'. A derived accessor may modify the options of the interface specification. For example,

```
exports.setup = function() {
  this.implement('MyInterface');
  this.parameter('foo', {'value':43});
```

```
}
```

modifies the default value of 'foo' to 43.

3.5. Extend

An accessor may specify that it extends another accessor in its `setup()` function by invoking:

- **extend**(*<string> name*): Extend an accessor with the specified name.

This has the effect of implementing the interface of the other accessor, as if implement had been invoked, but in addition, it inherits all exported fields and functions of the specified accessor. Specifically, the exports property of the base accessor becomes the prototype of the exports property of the extending accessor.

For example, suppose a base accessor `MyBase` has the following:

```
exports.initialize = function() {
  console.log('Initialize base accessor.');
}
```

Then if an accessor specifies the following:

```
exports.setup = function() {
  this.extend("MyBase");
}
```

then it will inherit the `exports.initialize()` function of the base accessor.

Functions

An accessor may override any exported function of the base accessor by simply defining a function with the same name. Moreover, the override function can invoke the overridden function as in the following example:

```
exports.initialize = function() {
  exports.ssuper.initialize.call(this);
  console.log('Initialize extending accessor.');
};
```

When the `initialize()` function of the extended accessor is invoked, then the following will appear on the console:

```
Initialize base accessor.   Initialize extending accessor.
```

Notice that the exports property has a ssuper property that refers to the exports property of the base accessor, which is also the prototype of the exports property of the extending accessor. Thus, ssuper refers to the prototype.

Notice further that the initialize function is invoked using `initialize.call(this)`. This ensures that while the base accessors initialize function is executing, the variable "`this`" refers to the extending accessor.

2. Input Handlers

An accessor can also override input handler functions. Suppose that a Base accessor has the following:

```
exports.inputHandler = function() {
   // Send true to output.
   this.send('output', true);
}
exports.initialize = function() {
   this.addInputHandler('input', this.exports.inputHandler.bind(this));
}
```

Notice that `this.exports.inputHandler` not `exports.inputHandler` is added as an input handler. Hence, a derived accessor that extends this base can override the input handler with the following code:

```
exports.inputHandler = function() {
   // Send false to output instead.
   this.send('output', false);
}
```

When the initialize function of the base accessor is invoked, it will actually add the derived accessor's handler rather than its own because "`this`" will refer to the derived accessor. In an instance of Derived, when an input arrives on 'input', the accessor will send `false` to 'output', not `true` as in the Base accessor.

If instead we had written in the base accessor

```
this.addInputHandler('input', exports.inputHandler.bind(this));
```

then the override would not succeed. As with all JavaScript functions, `exports.inputHandler` is resolved in the scope in which `initialize()` is *defined*, not in the scope in which it *executes*. Therefore, `exports.inputHandler` refers to the 'inputHandler' property of the Base accessor. This latter pattern can be used to prevent extended accessors from overriding certain behavior.

Inputs, Outputs and Parameters

A derived accessor may modify the options of the inputs, outputs, and parameters of the extended accessor. For example, if `MyBase` has a parameter named `foo`, then

```
exports.setup = function() {
  this.extend('MyBase');
  this.parameter('foo', {'value':43});
}
```

modifies the default value of 'foo' from whatever it is in the base to 43.

Accessing Base Variables in a Derived Accessor

A derived accessor may read and write the values of variables in the parent accessor if the parent accessor declares the variables using `exports`. For example if the `MyBase` accessor has:

```
exports.socket = null;
```

Then derived accessors may refer to that variable with

```
exports.ssuper.socket
```

For example UDPSocketListener is extended by Moto360GestureListener and Moto360GestureListener uses `exports.ssuper.socket`.

3.6. Data Types

Accessor parameters, inputs, and outputs follow a principle called *gradual typing*. If the XML definition of the input or output includes no type attribute, then any JavaScript type is acceptable. At this extreme, types are only checked at run time. But an accessor may constrain the type of an input, output, or parameter by including a type field.

The available types are still under discussion, but they will include at least:

- **boolean**: true or false
- **int**: an integer number
- **number**: a JavaScript number (integer or floating point)
- **string**: a string
- **JSON** : a JavaScript value or object that has a JSON representation. The object will be converted to a JSON string by the send() function, and converted back to a native JavaScript value or object by the get() or getParameter() function.

For any of these data types except boolean, you can define a finite number of possible values (an enumeration), as in the following example:

```
parameter('name',{'type':'string','value':'a','options':['a','b','c']
});
```

Giving an **options** option defines the allowed values.

4. Accessor Documentation

Documentation and metadata such as author, version, etc., are given using JSDoc tags in comments in the JavaScript accessor specification file. For example, the above minimal accessor might be documented as follows:

```
/** Double the value provided at *input* and send to *output*.
 *  @accessor Minimal
 *  @author Edward A. Lee (eal@eecs.berkeley.edu)
 *  @input {number} input A numeric input.
 *  @output {number} output The output for the doubled value.
 */
```

The main body of the documentation is given after the first `/**` in the accessor specification file. It can include any formatting commands supported by Markdown. For example, above, `*input*` will be rendered in italics. The documentation may include any of the following tags, in alphabetic order:

- @accessor *Name*: The name of the accessor. This should be capitalized and should match the name of the accessor file (without the .js extension). This is an accessor-specific tag used by the JSDoc Ptdoc Plugin
- @author *Author*: The author(s) of the accessor. This is arbitrary text and may optionally include contact information. See the JSDoc @author documentation.
- @input {type} *name*: A description of the input with the specified name and an optional type. If the input has a default value, that should be explained in the text. See JSDoc @type tag documentation for formatting of the type. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin.
- @module *Name*: The name of the accessor again, if the accessor is also available as a module. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin.
- @output {type} *name*: A description of the output with the specified name and an optional type. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin.
- @parameter {type} *name*: A description of the parameter with the specified name and an optional type. The default value should be explained in the text, if given. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin.
- @version *version*: A version designator for the accessor. For example, if the accessor is stored under an SVN version control system, this might specify `$$Id:$$`, which SVN will automatically replace with the current version number. This is an accessor-specific tag used by the JSDoc Ptdoc Plugin. Note that to avoid issues with expanding $, we use `$$Id$$` instead of `$Id$`.

5. Functionality

In addition to an interface, an accessor may have some functionality given in JavaScript. The script can define local state variables used by the accessor and functions to be invoked by the **swarmlet host**. The accessor's script will be executed in an **accessor context** provided by the host. The context provides some built-in functions and modules that are available for accessors to use. A swarmlet host *must* provide all the built-in functions and modules and *may* provide some or all of the optional modules. The set of required functions and modules is deliberately small, so that implementing a basic accessor host is easy.

Accessors run in a JavaScript host that provides a standard set of functions and modules that enable the accessor to interact with devices and services.

- Index of functions for quick reference.
- Action Functions: A set of functions that an accessor may define that are invoked by the host to execute a swarmlet. The swarmlet host will always

invoke these in such a way that the variable `this` is the accessor itself, which defines the functions such as `input`

- Top-Level JavaScript Functions: These function enable the accessor to get inputs and produce outputs. They also provide a small set of basic mechanisms that are commonly found in a JavaScript environment.
- Built-In JavaScript Modules: All accessor hosts are required to support all built-in modules. Modules conform with the CommonJS Module Specification.
- Optional JavaScript Modules: More sophisticated capabilities that may or may not be supported by a particular accessor host. These modules must also conform with the CommonJS Module Specification specification.
- JSDoc, auto-generated code documentation.

5.1. Action Functions

An accessor may define some number of action functions that are invoked by the host as described below. The only required function is the setup() function. An accessor may implement any subset of the rest. These functions are made available to the host by setting them as fields of the **exports** object in the accessor script with the corresponding name. For instance, the `initialize()` function is exposed in the accessor source code by setting the `exports.initialize` field equal to the desired function in the script. For example,

```
exports.initialize = function() {
    console.log('Hello World!');
}
```

An execution of an accessor consists of:

- Loading the accessor script, executing any top-level statements, and invoking the setup() function.
- Invoking the `initialize()` function, if it is defined.
- Executing the accessor, which performs the following actions repeatedly:
    - Invoke any timeout callbacks whose timeout matches current time of the host;
    - Invoke any input handlers that have been added to inputs that have new data;
    - If *any* input has new data, invoke any generic input handlers that have been added;
    - Invoke the `fire()` function, if it has been defined.
- Invoking the `wrapup()` function of the accessor.

fire()

If provided, the host will invoke this function when any new input is provided, or if there are no inputs at all, then whenever the host chooses to invoke it.

initialize()

This function, if provided, will be invoked once by the host when the application using the accessor starts up, and possibly again each time the swarmlet is re-initialized. The accessor may get any inputs that have default values, read parameters, and initialize state variables. It may also send outputs, but note that downstream accessors will not see this data in their initialize function. They will see it in an input handler, if they have registered one.

Input handlers

An input handler is a function that is invoked when a new input arrives. See Input Handlers.

latestOutput(*name*)

Return the most recent output sent via the named output.

provideInput(*name*, *value*)

Provide the specified input with the specified value.

react()

React to any provided inputs by invoking any associated input handlers and also invoking the fire() function, if there is one.

setParameter(*name*, *value*)

Set the specified parameter to have the specified value.

setup()

Set up the actor interface, defining inputs, outputs, and parameters.

wrapup()

If provided, the host will invoke this function once when the application shuts down. This function should not send outputs.

5.2. Discussion

With these action functions, there is a variety of patterns of accessor behavior:

- Event reactor: An accessor provides input handlers for one or more inputs. In these handlers, it may read inputs using get(), update state variables, and/or send outputs.
- Spontaneous accessor: The accessor sends outputs at times of its choosing, not in reaction to inputs. For example, it may send an output in

a function invoked after a timeout. The accessor can do this by calling `setTimeout(function,time)` in `initialize()`, for example. It may also send outputs in any callback function, for example one that handles a response to an asynchronous HTTP request or to a timeout. See Output.

Note that it is an error to send outputs before the host has invoked initialize or after it has invoked `wrapup()` (before the next `initialize()`). Therefore, an accessor should use its `wrapup()` function to, for example, cancel any pending timeouts or event handlers.

Note also that action functions are called "functions" even though they are not mathematical functions, because JavaScript calls them functions. They may have side effects.

5.3. Top-Level JavaScript Functions

This section describes the top-level JavaScript functions required to be provided by a swarmlet host. These are available to be invoked in any context by an accessor. Their invocation is prepended with `this`.

Module Dependencies

- **require**(*<string> @accessors-modules/module-name*): Load the specified module by name (a string) and return a reference to the module. The reference can be used to invoke any functions, constructors, or variables exported by the module.

Delaying Execution

- *<handle>* **setInterval**(*function*, *milliseconds*): Set the specified *function* to execute after specified time in milliseconds and again at multiples of that time, and return a handle.
- *<handle>* **setTimeout**(*function*, *milliseconds*): Set the specified *function* to execute after specified time in milliseconds and return a handle. If the interval is zero, then the specified function will be invoked in the next reaction, which will occur immediately after the conclusion of the current reaction. If several setTimeout requests have an interval of zero, then they will be invoked in successive reactions in the order in which they have been requested.
- **clearInterval**(*handle*): Clear a timer interval action with the specified *handle* (see **setInterval**()).
- **clearTimeout**(*handle*): Clear a timeout with the specified *handle* (see **setTimeout**()).

For both setInterval() and setTimeout(), if additional arguments are provided beyond the first two, then those arguments are passed to the function when it is called.

Also for both, if you wish for the specified function to send data to outputs or inputs or to get data from inputs, then you will need to bind the function to the accessor instance. For example,

```
var handle;
exports.initialize = function () {
    handle = setInterval(produce.bind(this), 1000);
};
function produce() {
    this.send('hello');
};
```

When `initialize()` is called, 'this' is the accessor instance. Binding the callback function to 'this' ensures that when it is called, 'this' will again be the accessor instance in `this.send('hello')`. You can alternatively use the following common JavaScript idiom:

```
var handle;
exports.initialize = function () {
    var self = this;
    handle = setInterval(function() {
        self.send('hello');
    }, 1000);
};
```

Here, the variable `self` captures the value of 'this' when `initialize()` called, making it available within the anonymous callback function.

After the specified function is called, the **fire**() function of the accessor will be called, if it exists. If the specified function is null, then only the **fire**() function of the accessor will be called.

Error Handling

- **error(string)**: Report an error with the specified message. This should be used by an accessor to report non-fatal errors, where it is OK to continue executing. For fatal errors, throw an exception.

Getting Resources

- **getResource(uri, options, timeout)**: Get a resource, which may be a relative file name or a URL, and return the value of the resource as a string. Implementations of this function will likely restrict the locations from which resources can be retrieved. A recommended policy for swarmlet hosts is to at least permit http and https accesses. Local files may be allowed, if for example they are given as relative file names relative to be in the same directory where the swarmlet model is located or in a subdirectory.

The options parameter may have the following values:

- If the type of the options parameter is a Number, then it is assumed to be the timeout in milliseconds.
- If the type of the options parameter is a String, then it is assumed to be the encoding, for example "UTF-8". If the value is "Raw" or "raw" then the data

is returned as an unsigned array of bytes. The default encoding is the default encoding of the system. In CapeCode, the default encoding is returned by Charset.defaultCharset().

- If the type of the options parameter is an Object, then it may have the following fields:
  - encoding {string} The encoding of the file, see above for values.
  - returnURI {string} If true, then return the URI of the resource instead of the contents. The default is false.
  - timeout {number} The timeout in milliseconds

If the callback parameter is not present, then getResource() will be synchronous read like Node.js's fs.readFileSync().

If the callback argument is present, then getResource() will be asynchronous like fs.readFile().

5.4. Built-In JavaScript Modules

The following objects provide bundles of functions and should be built in to any accessor host.

- console: Provides various utilities for formatting and displaying data.
- events: Provides an event emitter design pattern (requires util).
- util: Provides various utility functions.

Console

The console module is a JavaScript module for outputting messages. This implementation is designed to be compatible with the console module in Node.js. It requires the util module. Because this is a built-in module for accessors that subclass JSAccessor, there is no need to specify a *requires* tag in the interface specification. A simple use of the module in an accessor might look like this:

```
exports.fire = function() {
   var value = get(input);
   console.log('Input value is: %d', value);
}
```

The functions provided in this module are:

- **console.log(...)**: Print a message. This may go to stdout or to any other output appropriate for the particular accessor host. The first argument can be a printf-style formatting string, followed by arguments to insert into the output, as in the example above. If the first string does not contain any formatting elements, then util.inspect() is applied to all arguments to convert them to strings.
- console.info(...): Same as log().
- **console.error(...)**: Same as log(), but the message is prefixed with "ERROR: " and sent to stderr (or some other suitable destination).

- **console.warn(...)**: Same as log(), but the message is prefixed with "WARNING: " and sent to stderr (or some other suitable destination).
- **console.dir(object, options)**: Apply util.inspect() to the specified object (possibly with options) and then report as done by log(). The optional options argument is an object that may contain the following fields:
    - showHidden - if true then non-enumerable properties will be shown as well. Defaults to false.
    - depth - tells inspect how many times to recurse while formatting the object. Defaults to 2. Use null to get unbounded depth.
    - colors - if true, then the output will be styled with ANSI color codes. Defaults to false.
    - customInspect - if false, then custom inspect() functions defined on the objects being inspected won't be called. Defaults to true.
- **console.time(label)**: Record the current time using the specified label for use by a later call to timeEnd().
- **console.timeEnd(label)**: Log the time elapsed since the last call to time(label) that gave the same label (using the log() function to report the time).
- **console.trace(...)**: Send a stack trace to stderr or some other suitable output prefixed by "TRACE: " and any supplied message formatted as with the log() function.
- **console.assert(assertion, message)**: If the first argument is not "truthy", then throw an error that includes a (formatted) message given by the remaining arguments.

events

This module, which is borrowed from Node.js, contains a single class definition, EventEmitter. A typical usage pattern is to inherit it. For example:

```
var events = require('events');
var util = require('util');
function MyClass() {
   // Invoke the event emitter constructor.
   events.EventEmitter.call(this);
   util.log('intantiated');
}
util.inherits(MyClass, events.EventEmitter);
var instance = new MyClass();
instance.on('ping', function() {
    util.log('ping');
});
instance.emit('ping');
```

This code defines a class that subclasses EventEmitter. As a consequence, the class inherits the functions **on**() and **emit**(). The first of these functions defines what action to take when a named event is emitted. The last two lines of the code above

define the reaction to an event named 'ping' and then emit the event. The result of executing this code is, for example:

```
2 May 14:16:33 - intantiated   2 May 14:16:33 - ping
```

The events module is defined more fully at https://nodejs.org/api/events.html. It includes the following:

- Class: events.EventEmitter
  - emitter.addListener(event, listener)
  - emitter.on(event, listener)
  - emitter.once(event, listener)
  - emitter.removeListener(event, listener)
  - emitter.removeAllListeners([event])
  - emitter.setMaxListeners(n)
  - EventEmitter.defaultMaxListeners
  - emitter.listeners(event)
  - emitter.emit(event[, arg1][, arg2][, ...])
  - Class Method: EventEmitter.listenerCount(emitter, event)
  - Event: 'newListener'
  - Event: 'removeListener'

The util module is a JavaScript module containing various utility functions. This is intended to be compatible with the util module in Node.js. Because this is a built-in module for accessors that subclass JSAccessor, there is no need to specify a *requires* tag in the interface specification. A simple use of the module in an accessor might look like this:

```
exports.fire = function() {
   var value = get(input);
   util.log('Received an input.');
}
```

The functions provided in this module are:

- util.format(format, [...])
- util.debug(string)
- util.error([...])
- util.puts([...])
- util.print([...])
- util.log(string): Output a message with timestamp on stdout.
- util.inspect(object, [options])
- util.isArray(object)
- util.isRegExp(object)

- util.isDate(object)
- util.isError(object)
- util.pump(readableStream, writableStream, [callback])
- util.inherits(constructor, superConstructor)

6. Composition

An accessor may contain other accessors. To instantiate an accessor within another, use instantiate, as illustrated in the following example:

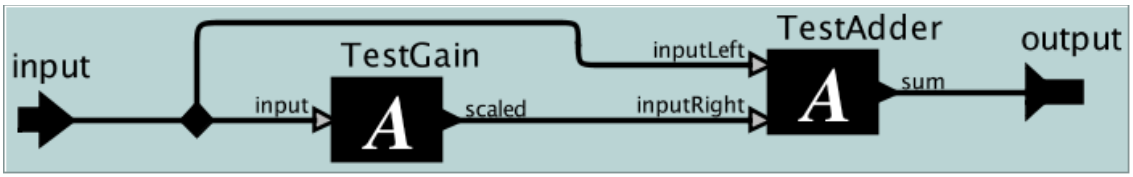
```
exports.setup = function() {
    var gain = this.instantiate('TestGain', 'test/TestGain');
    gain.setParameter('gain', 4);
}
```

This example instantiates a contained accessor `test/TestGain`, which is a test accessor from the TerraSwarm library that simply multiplies its input by a constant. It has a *gain* parameter, which in this case is being set to the constant 4. It could also be set to the parameter value of the composite using `getParameter()` (see parameter).

Any contained accessor will be initialized and wrapped up whenever its container is initialized and wrapped up (see execution below). But contained accessors are most useful if their inputs and outputs are connected to other contained accessors or to inputs and outputs of the container accessor. Connections are made using the connect function, as illustrated by the following example:

```
exports.setup = function() {
    this.input('input', {'type':'number', 'value':0});
    this.output('output', {'type':'number'});
    var gain = this.instantiate('TestGain', 'test/TestGain');
    gain.setParameter('gain', 4);
    var adder = this.instantiate('TestAdder', 'test/TestAdder');
    this.connect('input', adder, 'inputLeft');
    this.connect('input', gain, 'input');
    this.connect(gain, 'scaled', adder, 'inputRight');
    this.connect(adder, 'sum', 'output');
}
```

This connects two contained accessors (an instance of `test/TestGain` and one of `test/TestAdder`) as shown here:



7. Execution

An accessor is executed by a **swarmlet host**, a program that includes a JavaScript interpreter and provides the script with a collection of library functions and modules. All interaction between the accessor script and the network, devices, the file system, and other accessors is mediated by these libraries. Hence, the host controls what the accessor script has access to. Not all hosts can execute all accessors. The browser host, for example, limits network and file system access according to a browser security policy. When an accessor is instantiated by a host, the host checks compatibility and reports if the accessor is incompatible with that host.

The life of an accessor has four phases:

- instantiation: The host reads and executes the JavaScript file defining the accessor. If in that file an `exports.setup` function is defined, then that function will be executed in this phase.
- initialization: If the accessor defines an `exports.initialize` function, then that function is invoked in the initialization phase. This marks the beginning of the execution of a swarmlet.
- reaction: The accessor reacts to inputs and callbacks, as explained in detail below.
- wrapup: If the accessor defines an `exports.wrapup()` function, then that function is invoked in the wrapup phase. This marks the end of the execution of the swarmlet using the accessor.

The host instantiates an accessor exactly once. After instantiation, execution of the accessor consists of initialization, zero or more reactions, followed by wrapup. This execution pattern may be repeated multiple times.

A **reaction** of an accessor is triggered by the swarmlet host. When the host does this depends on the host. The Browser Host for example, will trigger a reaction when the user clicks a button labeled 'react to inputs' on a web page displaying the accessor. The Node Host will trigger a reaction when the script being executed invokes the `react()` function on the accessor instance. The Cape Code Host will trigger a reaction when an input to the accessor arrives.

A reaction performs exactly the following actions, in order:

- Invocation of any timeout callbacks, if the time is appropriate.
- Invocation of any registered handlers for any input that has received new data. These handlers will be invoked first in the order in which inputs are defined, and then, if an input has multiple handlers, in the order in which the handlers were added.
- Invocation of any registered handlers for new input data that the accessor has sent to itself. An accessor will continue checking for self-provided new input data until none is available.
- Invocation of any handlers that have been registered to be invoked when *any* new input arrives. These handlers will be invoked in the order in which they were added.
- Reaction of any contained accessors that have been provided with inputs (see below).
- If the accessor defines an `exports.fire` function, then that function is invoked now.

In the above, "new input" means input that has arrived from an external source since the last reaction. Inputs sent to the accessor by itself are handled in the same reaction.

A host is free to invoke a reaction whenever it chooses after it has invoked `initialize()` and before it has invoked `wrapup()`, even if there are no new inputs. Hence, anything the accessor does in its `fire()` function should make sense regardless of whether new inputs have been provided.

When invoking reactions of contained accessors, the swarmlet host follows a specific deterministic model of computation. First, contained accessors are assigned a **priority** based on their position in the graph of connected accessors. An accessor *A* with an output connected to accessor *B* will have a higher priority than *B*, unless the output of *A* is declared to be **spontaneous** (see output). The host will invoke reactions in priority order, ensuring that any accessor that feeds data to another accessor will react first.

An accessor may invoke `stop` which will invoke `wrapup()` on any contained accessors and then stop the host.

8. http-client Module

A host may provide some or all of a suite of optional modules. Here, we describe just one of those as an illustrative example, the http-client module. This is a library for making HTTP requests.

8.1. API

- **get**(*url | options*, *responseCallback*): Make an HTTP GET request and call the callback when the request has been satisfied.
- **post**(*url | options*, *responseCallback*): Make an HTTP POST request and call the callback when the request has been satisfied.
- **put**(*url | options*, *responseCallback*): Make an HTTP PUT request and call the callback when the request has been satisfied.
- **request**(*url | options*, *responseCallback*): Make an HTTP request and call the callback when the server has responded.

In all cases, the first argument is either a string with a URL or a JavaScript object with a number of fields defined below. Also, in all cases, the *responseCallback* function, if provided, will be called when the request has been fully satisfied.
The *responseCallback* function will be passed one argument, an instance of the **IncomingMessage** class, defined below. The difference between **request**() and the other functions is that the former only begins the interaction with the server, whereas the rest are complete, self-contained requests. Specifically, if you use **request**, you can use the returned object to send additional information to the server, for example to PUT a file on the server. Both functions return an instance of the **ClientRequest** class, defined below.

8.2. Usage

A simple use to read the contents of a web page looks like this:

```
var http = require('@accessors-modules/http-client');
http.get('http://ptolemy.eecs.berkeley.edu', function(response) {
   console.log('Got response: ' + response.body);
});
```

To view all returned parameters from http.get():

```
var http = require('@accessors-modules/http-client');
http.get('http://ptolemy.eecs.berkeley.edu', function(response) {
   console.log('Got response: ' + JSON.stringify(response));
});
```

POST and PUT requests typically specify a body, using options.body. (This URL will give an error since it doesn't allow posting).

```
var http = require('@accessors-modules/http-client');
var options = {
    body : {test : 'this is a test'},
    url : 'http://ptolemy.eecs.berkeley.edu'
};
http.post(options, function(response) {
   console.log('Got response: ' + JSON.stringify(response));
});
```

8.3. Options

The *options* argument to the above functions can be a string URL or a JavaScript object with the following (optional) fields:

- **body**: An object containing the request body. Images are supported in CapeCode only.
- **headers**: An object containing request headers. By default this is an empty object. Items may have a value that is an array of values, for headers with more than one value.
- **keepAlive**: A boolean that specified whether to keep sockets around in a pool to be used by other requests in the future. This defaults to false.
- **method**: A string specifying the HTTP request method. This defaults to 'GET', but can also be 'PUT', 'POST', 'DELETE', etc.
- **outputCompleteResponseOnly**: A boolean specifying if only a complete response is allowed, or if multi-part responses are acceptable.
- **timeout**: An integer specifying the maximum time to wait for a response, in milliseconds. Defaults to 5000 ms.
- **url**: A string that can be parsed as a URL, or an object containing the following fields:
    - **host**: A string giving the domain name or IP address of the server to issue the request to. This defaults to 'localhost'.
    - **path**: Request path as a string. This defaults to '/'. This can include a query string, e.g. '/index.html?page=12', or the query string can be specified as a separate field (see below). An exception is thrown if the request path contains illegal characters.
    - **protocol**: The protocol. This is a string that defaults to 'http'.
    - **port**: Port of remote server. This defaults to 80.
    - **query**: A query string to be appended to the path, such as '?page=12'.

8.4. IncomingMessage

The *responseCallback* function, if provided, will be passed an instance of **IncomingMessage**, which has these fields:

- **statusCode**: an integer indicating the status of the response. A description of HTTP response codes can be found here: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. Codes in the 300 range indicate redirects, and codes 400 or above indicate errors.
- **statusMessage**: a string with the status message of the response.
- **body**: a string with the body of the response.

8.5. ClientRequest

The *request* and *get* functions both return an instance of ClientRequest, which has the following functions:

- **end**(): Call this to end the request (no more data to send to the server). The **get**() function automatically calls this for you, but **request**() does not.
- **write**(*data*, *encoding*): Write data (e.g. for a POST request) using the specified encoding (e.g. 'UTF-8').
- The ClientRequest class is an event emitter that emits the following events:
- 'error': If an error occurs. The message is passed as an argument.
- 'response': A response is received from the server. This event is automatically handled by calling responseCallback, if responseCallback is not null, but you can also listen for this event.

8.6. Browser Considerations

Cross-Origin Requests

For security reasons, scripts in a browser are generally prohibited from certain types of data transfer in cross-origin requests. The same-origin policy states that two pages have the same origin if the protocol, port (if given), and host are the same for both pages.

Client-server coordination is required to achieve cross-origin communication. Two techniques are cross-origin resource sharing (CORS) and JSON with padding (JSONP).

Cross-Origin Resource Sharing (CORS)

A server may stipulate that cross-origin requests are allowed. This is called cross-origin resource sharing (CORS).

- For simple requests, a CORS-aware browser automatically adds an `Origin` request header specifying the host. For example:

```
Origin: http://www.terraswarm.org
```

A server sets the response header `Access-Control-Allow-Origin` to the list of hosts allowed access, using * for all hosts. For example:

```
Access-Control-Allow-Origin : *
```

or

```
Access-Control-Allow-Origin : http://www.terraswarm.org
```

The client checks this header. If access is allowed, the content is loaded.

- For non-simple preflighted requests, the client first sends an HTTP OPTIONS request with additional information.

JSON with padding (JSONP)

The JSON with padding technique takes advantage of the fact that `<script>` tags in HTML are exempt from cross-origin restrictions. For example, this is allowed:

```
<script src="http://www.otherdomain.com/thescript.js"> </script>
```

In a JSONP request, the client specifies some text to pre-pend to a JSON response. The server replies with the text plus the JSON data enclosed in parenthesis. If the

client defines a function locally and sends the function name as the text, this function is then executed with the JSON data as an argument.

For example, calling http://ip.jsontest.com/ will return your IP address in JSON form. Example:

```
{"ip": "8.8.8.8"}
```

A client could define a function that displays data:

```
function showIP(data) {   alert(data); }
```

Next, the client asks the server to prepend this function name to the returned JSON by appending `?callback=[function_name]` to the request URL. For example, calling http://ip.jsontest.com/?callback=showIP will return this response:

```
showIP({"ip": "8.8.8.8"});
```

The `showIP` function then executes with the returned data as an argument.

jQuery's `getJSON()` function will issue a JSONP request if the URL ends in `?callback?`. For example:

```
jquery.getJSON(
   "http://jsonplaceholder.typicode.com/posts/1?callback=?",
   function(data) {
       console.log(JSON.stringify(data));
   });
```

APIs and Libraries

XMLHttpRequest is an API that allows a client to fetch and send data without having to do a full page refresh. All modern browsers (Chrome, IE7+, Firefox, Safari, and Opera) include a built-in XMLHttpRequest implementation. XMLHttpRequest supports:

- Both asynchronous and synchronous requests
- Both text and binary data
- Event listeners
- Form submission and file upload

Please see Mozilla's XMLHttpRequest for a complete list of methods and browser support, and Using XMLHttpRequest for examples.

jQuery is a "fast, small, and feature-rich JavaScript library" that is supported by modern browsers. jQuery defines an easy-to-use API for common HTML manipulation and data transfer tasks.

jQuery includes an AJAX (Asynchronous Javascript and XML) method that wraps XMLHttpRequest for issuing requests. There is also a shorthand GET method.

Browserify is a bundling system and set of modules that replicates most node.js APIs and functionality in a browser. Browserify allows one to use the "requires" syntax in a browser and creates a bundle of all modules needed for a particular piece of code by collecting all of the "require" dependencies. For example:

```
browserify main.js > bundle.js
```

will recursively locate all dependencies starting in the `main.js` file and output flattened Javascript to `bundle.js`. The HTML page then includes this bundle using a script tag:

```
<script src="bundle.js"> </script>
```

Browserify includes both HTTP and HTTPS modules. These modules depend on many others, so accessors that use browserify would need to either include bundles or have the browserify source code available in the accessor repository or other location.

There is a browserify version of jQuery, **jQuery-browserify**. Unfortunately jQuery-browserify has no license, so the terms of use are unknown.