JavaScript Plugin

Tony Voss

Contents Page

Introduction and summary

This document is a user guide and reference manual for the JavaScript plugin for OpenCPN.

The plugin allows you to run JavaScript and to interact with OpenCPN. You can use it to build your own enhancements to standard OpenCPN functionality.

There is a separate technical guide covering the inner workings of the plugin and instructions for building it from sources.

The basics

Once the plugin has been enabled, its icon appears in the control strip.

Click on the icon to open the plugin console. The console comprises a script pane, an output pane and various buttons. You can write your JavaScript in the script pane and click on the **Run** button. The script will be compiled and executed. Any output is displayed in the output pane.

As a trivial example, enter

```
(4+8)/3
```

and the result 4 is displayed. But you could also enter, say

```
function fibonacci(n) {
      function fib(n) {
            if (n == 0) return 0;
            if (n == 1) return 1;
            return fib(n-1) + fib(n-2);
            }
      var res = [];
      for (i = 0; i < n; i++) res.push(fib(i));
      return(res.join(' '));
      }
print("Fibonacci says: ", fibonacci(20), "\n");
```

which displays

```
Fibonacci says: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181
```

This illustrates how functions can be defined and called, including recursively. So we have a super calculator!

You could paste the above into the script window and run it.

Note that the script pane displays line numbers - useful for understanding any syntax error messages. It also supports indent tabbing, which you should use to indent your script as in the above example.

You can use the `Load` button to load JavaScript from a .js file. The file name string is displayed above the script. The `Save` button saves the script back to the named file and the `Save As` button allows you to save it to a different chosen file.

You can also paste a script in from somewhere else. You might choose to prepare a non-trivial script in a JavaScript-aware editor. I use BBEdit on my Mac.

In this alpha release there are two test buttons **Test A** and **Test B**. Please ignore them.

JavaScript and the embedded engine

A useful guide/tutorial on JavaScript can be found here. The engine fully supports ECMAScript E5.1 with partial support for E6 and E7. (See the Duktape page here for details, but note that not all features discussed on that site are available.)

Note that the embedded engine does not support:
- for/of loops
- classes
- arrow functions

The tutorial also covers JavaScript's use in web pages which is not relevant at this time.

JavaScript plugin extensions

As the JavaScript engine is intended for embedding, it does not contain any input/output functionality, which is inevitably environment dependent.

I have implemented a few extensions to provide some output capability.

print(arg1, arg2…)

The print function displays a series of arguments in the results pane. Each argument can be of type *string*, a *number* or *boolean* value. Example:

```
print("Hello world ", 10*10, " times over\n");
```

Displays `Hello world 100 times over.` It is often useful to include the string "\n" as the last character to deliver a newline. If the console has been hidden, it will be shown so that the output can be seen.

alert(arg1, arg2…)

This is similar to `print` but the output is displayed in an alert box. The final newline is less necessary here. Be aware that OpenCPN processing is held up until an alert box is dismissed.

An argument cannot be an array or structure.. To display either of these, first turn it into a JSON string.

```
var boats = ["Yacht", "Dinghy", "Tender"];
print(boats, "\n");    // this will fail as boats is an array
print(JSON.stringify(boats), "\n");  // prints the JSON string of boats
```

readTextFile(fileString)

Reads the text file `fileString` and returns the text as a string. Example:

```
input = readTextFile("/Users/Tony/myfile");
print("File contains: ", input, "\n");
```

require(moduleName)

Loads and compiles the given module. See the Modules section.

Understanding the result

After a script has run, the *result* is displayed in the output window after any other output, such as from print statements. The result is the last executed statement, so for

```
"Hello";
print("Hi there!\n");
```

this will display “Hi there!” Followed by the result, which is `Hello` - this being the last statement returning a result.

In this example

```
3+4;
3 == 4;
```

The result is `false`. The `3+4` is not the last statement. The last statement has a boolean result of false.

If there is no statement returning a result, it is undefined, which is very often the case.

OpenCPN APIs

I have developed a number of APIs - interfaces to the functionality of OpenCPN.

These are all functions with names starting with *OCPN*.

Often it is necessary to set up a response to an OpenCPN event. These functions set up a callback to a function you supply and have names starting with *OCPNon.* The first argument is the name of the function to be called on the event. This sets up one call-back only. If you want the function to be called repeatedly, it needs set up the next call within it. When the OCPNon function is executed, a check is made that the nominated function exists within your code. Usually any error will be reported on compilation. However, where a call is made within a called-back function, the error can only be discovered at that time.

Herewith the currently implemented APIs.

OCPNpushNMEA(sentence)

Sentence is an NMEA sentence. It is truncated at the first * character, thus dropping any existing checksum. A new checksum is appended and the sentence pushed out over the OpenCPN connections. Example:

```
OCPNpushNMEA(“$OCRMB,A,0.000,L,,Yarmouth,5030.530,N,00120.030,W,15.386,82
.924,0.000,5030.530,S,00120.030,E,V");
```

OCPNgetMessageNames()

Returns a list of the message names seen since plugin activation. The list is one name per line. If a call-back is outstanding for that message, the name of the function is also displayed. Example:

```
print(OCPNgetMessageNames());
```

This is primarily used to determine what messages are being received and their precise names.

OCPNsendMessage(messageName[, message])

Sends an OpenCPN message. messageName is a text string being the name of the message to be sent, as reported by OCPNgetMessageNames. Optionally, you may include a second parameter as the JSON string of the message to be sent. Example:

```
routeGUID = “48cf3bc5-3abb-4f73-8ad2-994e796289eb”;
OCPNsendMessage(“OCPN_ROUTE_REQUEST”,JSON.stringify({“GUID”:routeGUID}));
```

Before making this call, you should have set up a call-back function using `OCPNonMessageName`.

OCPNonSeconds(functionName, seconds[, parameter])

Sets up a call to `funcionName` after a time of `seconds` have elapsed. Optionally, you may include a third parameter, which will be used as the argument for the call-back. Example:

```
OCPNonSecs(timesUp, 15, “15 seconds gone”);

function timesUp(what){
      print(what, “\n”);
      }
```

This would display the message `15 seconds gone` after 15 seconds.

Unlike other call-backs, you may set up any number of timed call-backs to different functions or the same function but each call-back is fulfilled only once. If multiple call-backs are due at the same time, they are fulfilled in reverse order with the most recently set up obeyed first.

OCPNonNMEAsentence(functionName)

Sets up a function to process the next NMEA sentence received by the plugin. The function is passed a structure containing `OK` -a boolean value concerning the validity of the checksum - and `value` - the sentence itself. Example:

```
OCPNonNMEAsentence(processNMEA);

function processNMEA(result){
      if (result.OK) print("Sentence received: ", result.value, "\n");
      else print("Got bad NMEA checksum\n");
      }
```

OCPNonMessageName(functionName, messageName)

Sets up a call-back to `functionName` next time a message with the name `messageName` is received. The function is passed the message, which is in JSON format. Example:

```
routeGUID = "48cf3bc5-3abb-4f73-8ad2-994e796289eb";
OCPNonMessageName(handleRT, "OCPN_ROUTE_RESPONSE");
OCPNsendMessage("OCPN_ROUTE_REQUEST",JSON.stringify({"GUID":routeGUID}));

function handleRT(routeJS){
      route = JSON.parse(routeJS);
      print("RouteGUID ", routeGUID, " has the name ", route.name, "\n");
      };
```

Note that here I have set up the call-back before sending the request to be sure the call-back is in place when the message arrives.

OCPNgetNavigation(functionName)

This function returns the latest OpenCPN navigation data as a structure. Example:

```
here = OCPNgetNavigation();
// print("JS-control:", JSON.stringify(here), "\n");
fromHere = here.position.value;   // saves the lat and long
variation = here.variation.value; //save magnetic variation
```

The structure created is in the style set by SignalK. Note in the above example the commented-out print statement which would print the entire message as a JSON string. This could be used to examine the structure and determine how to access it.

OCPNgetARPgpx()

This function returns the active route point as a GPX string or an empty string if there is no active route point. You need to parse the GPX string as required. Example:

```
APRgpx = OCPNgetARPgpx();  // get Active Route Point as GPX
if (APRgpx.length > 0){
      waypointPart  = /<name>.*<\/name>/.exec(APRgpx);
      waypointName = waypointPart[0].slice(6, -7);
      print("Active waypoint is ", waypointName, "\n");
      }
else print("No active waypoint\n");
```

OCPNcancelAll()

Cancels all outstanding call-backs and cleans up.

Call-backs are obeyed only once and are, therefore, self-cancelling unless renewed. But sometimes a call-back may never be fulfilled. This function cancels any outstanding call-backs and cleans up ensuring the Stop button returns to Run.

Modules

You can load modules which extend the standard JavaScript functionality from a library of pre-defined functions or methods. This is achieved with the `require` function.

The argument of `require` is the name of the function/object to be loaded. If that argument is a simple name without a suffix or file path separator, `require` looks in the library included with the plugin. Otherwise it looks for a matching file relative to your home directory. You cannot use an absolute path starting with '/' but you can use `..` notation to move up from your home directory. (This is true for MacOS and Unix-based systems - others to be confirmed.)

Functions

A function can be loaded as shown below. In this example, the fibonacci function built into the plugin library is loaded

```
fibonacci = require("fibonacci");
// defined fibonacci to be the function defined in the plugin library
print("Fibonacci says: ", fibonacci(15), "\n");
// prints Fibonacci says: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

The next script loads a similar function from the user's file space:

```
myfibby = require("myjavascripts/fibonacci.js");  // load my own version
print("My version says: ", myfibby(10), "\n");
// prints My version says: 0 1 1 2 3 5 8 13 21 34
```

Structures and methods

JavaScript supports the use of structures. The `require` function can load an object constructor which can include methods. By convention, constructors have an initial capital letter as a reminder that they are constructors.

You use the require function to load the constructor. You then need to construct one or more objects using the `new` statement, as shown in the following example.

`Position(lat, lon)`

This constructs a position structure with `value.latitude` and `value.longitude` set with the supplied arguments, which are optional.

This structure has the following properties and methods:

1. `.formatted` returns the position formatted for the human eye
2. `.nmea` returns the position formatted as used in NMEA sentences. You need to add the comma before and after, if required.
3. `.NMEAdecode(sentence, n)` decodes the NMEA sentence and sets `the` position to the n^{th} position in the string

Example 1:

```
Position = require("Position");
myPosition = new Position(58.5, -1.5);
myPosition.value.longitude = 0.5;      // change the longitude
print(myPosition.NMEA, "\n");    // displays 5830.000,N,0030.000,E
```

Example 2: Decode an NMEA string and print the second position for the human eye:

```
Position = require("Position");
```

```
thisPos = new Position();
sentence = "$OCRMB,A,0.000,L,,UK-
S:Y,5030.530,N,00121.020,W,0021.506,82.924,0.000,5030.530,S,00120.030,E,V
,A*69";
thisPos.NMEAdecode(sentence,2);
print(thisPos.formatted, "\n"); // displays 50º 30.530'N 001º 20.030'W
```

Loading your own functions

As an example of how to write your own functions to load with `require`, here is the fibonacci function.

```
function fibonacci(n) {
      function fib(n) {
            if (n == 0) return 0;
            if (n == 1) return 1;
            return fib(n-1) + fib(n-2);
            };
      var res = [];
      for (i = 0; i < n; i++) res.push(fib(i));
      return(res.join(' '));
      };
```

Note: the function fib is defined within the main function fibonacci. This is allowed in most JavaScript engines but is contrary to the ECMAScript standard.

Loading your own object constructors

Here is a trivial constructor for an object which includes a method

```
function Boat(_name, _make, _model, _length){
      this.name = _name;
      this.make = _make
      this.model = _model
      this.length = _length;
      this.summary = function(){return(this.name + " is an " + this.make
+ " " + this.model + " of length " + this.length +"m\n");}
      }
```

Note how the properties are set when the constructor is called. As this is a constructor, an object must be created from it, once loaded. Example:

```
Boat = require("myjavascripts/Boat.js");
myBoat = new Boat("Antipole", "Ovni", 395, 12);
myBoat.length = 12.2; // correction
print(myBoat.summary(), "\n");
// prints Antipole is an Ovni 395 of length 12.2m
```

Demonstration script

This script counts down for 30 seconds and then lists the OpenCPN messages and NMEA sentences it has seen. The NMEA sentences are sorted by count and then alphabetically.

The script demonstrates several aspects of JavaScript and the OpenCPN APIs.

You can copy it and paste it into the script window.

```
// listens for OpenCPN sentences and NMEA messages over configured time
and reports
// sorts NMEA messages by count and then alphabetically

var timeSeconds = 30; // time period over which to count
var progressSeconds = 5;  // count down this often
var secondsLeft = timeSeconds;
var log = [];   // will be array of entries

OCPNonSeconds(progress, 1); // count down every progressSeconds secs, but
first almost immediately
OCPNonSeconds(report, timeSeconds);
OCPNonNMEAsentence(logit);

function progress(){
      if (secondsLeft == timeSeconds){
            // this is first time in
            print("Seconds remaining");
            }
      if (secondsLeft >= progressSeconds){
            print(" ", secondsLeft);
            if (secondsLeft > progressSeconds) OCPNonSeconds(progress,
progressSeconds);
            secondsLeft -= progressSeconds;
            }
      }

function logit(returned){
      var entry = {type: "", count: 0};
      if (secondsLeft > 0) OCPNonNMEAsentence(logit);
      if (returned.OK){
            thisType = returned.value.slice(1, 6);
            count = log.length
            if (count > 0){
                  for (i in log){
                        if (log[i].type == thisType){
                              log[i].count += 1;
                              return;
                              }
                        }
                  }
            // no match or first entry - create new one
            entry.type = thisType;
            entry.count = 1;
            log.push(entry);
```

```
            }
      }


function report(){
      secondsLeft = 0;
      print("\n\nMessages seen:\n", OCPNgetMessageNames());
      entryCount = log.length;
      if (entryCount > 0){
            log.sort(function(a,b){ // sort log first by count then by
type alphabetically
                  if (a.count != b.count) return (b.count - a.count);
                  else return ((a.type < b.type) ? -1 : 1);
                  });
            print("\nNMEA sentences seen:\n");
            for (i in log){
                  print(log[i].type, " ", log[i].count, "\n");
                  }
            }
      else print("\nNo NMEA to report\n");
      OCPNcancelAll();
      }
```

Plugin version history

Version	Date	
0.1		Initial alpha release for feedback