JavaScript Plugin

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.

This document also has an appendix describing the technical details and notes for developers who might be porting the plugin to different platforms.

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.

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
```

Technical documentation

This section includes technical documentation to aid plugin compilation, maintenance and implementation on new platforms. This is not needed for using the plugin.

Background

This plugin started with a cloning of the DR_pi as described here. This is noted because this is where the Cmake files originated.

I developed this plugin on MacOSX using Xcode extensively and have not compiled it otherwise. The Xcode scripts call on the original Cmake files.

Compiling and building for MacOSX

The developer tools and, notably, wxWidgets were installed as described here.

As set up, it is sufficient to select the `JavaScript_pi` target in Xcode and click on Run. This compiles as needed and builds the installer, which will be found in the `build` directory.

Note in particular that the build settings include the following pre-processor macro definitions:

```
#define JavaScript_USE_SVG
#define TIXML_USE_STL
#define SCI_LEXER
```

The C++ Language Dialect is set to GNU++ 14

Other settings can be determined by inspection and use `wx-config`.

Duktape JavaScript engine

This is included as single source file duktape.cpp which can be found together with header files and test programs in the Duktape Xcode group. As supplied, the source file was of type .c I changed the extension to .cpp to force it to compile as C++. The code is designed to compile as either.

There is a separate Xcode target `Duktape test`, which builds a command line utility for testing the JavaScript engine in isolation. I have not used this after an initial check, preferring to do testing in the test harness as described below.

The test harness

Building with the Test-harness target compiles the plugin together with the `Test_harness.cpp` main program, which allows the plugin to be run from Xcode without OpenCPN. Most of the development work was done this way and only after all was working in the test harness was it built as a plugin and installed into OpenCPN.

Running the test harness from Xcode provides full debugging tools including step-by-step execution and examination of variables.

To make this possible, `Test_harness.cpp` includes dummy stubs for what is missing in the absence of OpenCPN.

In a few cases it contains code to return sample data as if from OpenCPN so that subsequent processing can be developed within the debugging environment. An example is `GetActiveRoutepointGPX()`.

I found no way to dummy out the building of icons and so that code is not compiled if the macro IN_HARNESS is defined, as it is when building the test harness.

Duktape APIs

The code to provide non-OpenCPN-specific extensions (such as the print function) are included in `JSExtensions.cpp`.

The basic technique is that when setting up the Duktape context, an initialisation function is called, which loads into the context the details of the C++ functions to be called when JavaScript executes the function.

The extensions that provide the OCPN APIs are handled similarly in the file `OPCNapis.cpp`. The file `opcpn_duk.h` contains the definitions of the class and methods used to implement many of the APIs.

It takes some understanding of how to work with the Duktape context stack, especially when constructing objects such as that returned by the `OCPNgetNavigation` function. To allow exploration of what is happening on the stack, I created a conditional macro `MAYBE_DUK_DUMP` which can be inserted in the code. This is defined in `opcpn_duk.h` together with the environment variable `DUK_DUMP`. If `DUK_DUMP` is true, the macro dumps the stack as it is at that point into the output window. I have only needed to use this from within the test harness.

The script window

The console has been created with wxFormBuilder as usual.

The output pane is of type `wxTextCtrl` and originally I used this for the script window also. However, this was unsatisfactory as it lacks line numbers (useful in understanding script error messages) and does not support indenting with tabs. I resorted to writing almost all scripts externally in BBEdit and pasting them in.

A further complication is that `wxTextCtrl` uses Unicode characters, which are not acceptable to the Duktape engine. Smart quotes and primes are a particular problem. The stander wx conversion functions filter them out but do not convert them appropriately. I ended up converting them within `runJS.cpp` before passing the script to the Duktape engine.

Exploring in wxFormBuilder, I then discovered that there is an alternative type of field `wxStyledTextCtrl` which can support a wide variety of text layouts, one of which C++ is close enough to JavaScript and works well. However, including a `wxStyledTextCtrl` pane led to a dozen unresolved functions needed to support it. These are part of the Scintilla package, which is an optional extra part of wxWidgets not included in OpenCPN.

My build of wxWidgets includes the Scintilla package as the stc dylib - `libwx_osx_cocoau_stc-3.1.2.0.0.dylib` in the case of MacOSX. Linking this library into the test harness build resolved the issue and gave me a working test harness with a much superior script window.

However, I could not find a way of including this library in the plugin in a way that worked. The OpenCPN plugin loader only loads the one dylib. Editing the extra library into OpenCPN itself did not work. I spent considerable time exploring whether I could merge the stc dylib into the JavaScript_pi dylib but found no way.

The only solution seems to be to compile the Scintilla source code along with the plugin so that it gets incorporated into the one dylib. The source code is configured through the build process for wxWidgets and I deduced that the source files in my wxWidgets build had already been configured. I therefore copied the source files into a folder `scintilla` and added this as an extra Xcode group. After a lot of fiddling got it to compile. This required the addition of 148 compile sources in the Build Phase and I needed to set the following in the pre-processor macros

```
#define TIXML_USE_STL
#define SCI_LEXER
```

The JavaScript plugin now has a decent scripting window.

When the plugin is ported to a non-Xcode environment, much of this work can probably be achieved with Cmake scripting but this is not something I am familiar with.

It is to be note that the Scintilla package is comprehensive. It includes support for numerous languages including the likes of Cobol and Fortran. It is large and increased the size of the plugin from 527KB (including the JavaScript engine) to 2MB. It should be possible to drastically reduce the size overhead by dummying out unused code.

If this increase in size were a problem for installations with limited memory, it would be possible to revert to making the script window of type `wxTextCntl` but it is much less satisfactory. I favour reducing the size of Scintilla and will be looking at this.

The require function

Duktape provides a framework in which to implement a `require` function. In that framework, the included script is automatically compiled in a separate context and then exported to the user's context.

Despite two weeks of experimenting and testing, I found no way of exporting an object method. It was not recognised as callable. Eventually, I abandoned this approach and implemented a require function from scratch in which I compile the script as a function within the user's context.

Road map for future development

So far, I have only implemented the APIs needed to achieve my application above. I would like to hear of other possible applications so I can consider further APIs.

The plugin is not yet ready for alpha testing but I would be interested to work with others in a pre-alpha phase to develop ideas. This pre-alpha phase will be MacOS only without publishing of source code as I want to experiment and change it freely. I propose we use a Slack workspace I set up to liaise with Mike. If you would like to join in, please contact me by private message.

I anticipate developments will include:

- Addition of further APIs as need identified
- ~~Documentation and a user guide~~ ✓
- ~~Making the scripting window more programmer friendly. At present it knows nothing of tabs, indents and braces. For other than the simplest script, I presently use a JavaScript-aware editor (BBEdit in my case) and paste the scripts into the script window.~~ ✓
- Better resilience. At present there is no protection against a script loop. `while(1);` hangs OpenCPN!

- ~~Implementing the JavaScript `require()` function, which is like a C++ `#include` to allow loading of pre-defined functions, objects, and methods.~~ ✓
- ~~Running without the console window visible~~ ✓
- Tidier and more consistent error reporting, even when the console is hidden
- ‘Canned’ scripts that start automatically
- At present, if you want to do separate tasks, you would need to combine them into a single script. I have ideas about running multiple independent scripts.
- I do not use SignalK but note its potential. I am interested in input from SignalK users to keep developments SignalK friendly.
- Other suggestions?

Version history

Version	Date	
0.1		Initial alpha release for feedback