

# JavaScript Plugin Technical Guide

Tony Voss

Version 0.4 6 April 2021 - [document history here](#)

<b>Contents</b>	<b>Page</b>
1. Introduction	3
2. The JavaScript Engine	3
3. Compiling and building	3
Windows	3
Linux	3
MacOS	3
Building from terminal	3
Building using Xcode	4
The Xcode test harness	4
Duktape test	7
4. Duktape extensions	7
5. Duktape OpenCPN APIs	7
6. The script window	7
7. The require function	8
8. How the plugin works	8
Major changes in version 0.4	8
General description	8
exitScript()	9
Time ticks	9
Timing out	9
Closing consoles	10
Keeping the right console on top	10
Programmatically starting scripts	10
Inter-script communication	10
9. Error handling	11
Error in the plugin code when no JavaScript is running	11
Error in the main script	11

Error detected in C++ extension or OpenCPN API code.	11
On call-back, the plugin detects error before invoking function	11
Function invoked during call-back throws an error	12
10. Testing	12
11. Trouble-shooting and debugging	12
Preferences panel diagnostics tab	12
Dump	12
Clean and dump string	12
Obtaining dumps from with code	13
From within C++ code	13
From within JavaScript	13
Throwing an error with C++ code	13
Re-entrancy and checking for the wx_widgets main thread	13
Tracing	13
Appendices	15
Road map for future development	15
Document history	16

# 1. Introduction

This document is a technical guide for the JavaScript plugin for OpenCPN. It is intended for those maintaining the plugin and building it from source.

This plugin started life 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. Many thanks to Mike Rossiter for fixing up the Cmake files, testing it under Windows and Linux and providing the builds.

Changes since the last release are highlighted in yellow.

## 2. The JavaScript Engine

The JavaScript engine used is [Duktape](#), which is optimised for being built into an application such as this. It implements ECMAScript E5/E5.1 with partial support for later developments. The web site provides full details of the Duktape API - the interface between the engine and the host application. This is not to be confused with the OpenCPN API through which the plugin interacts with OpenCPN.

A more detailed explanation is provided in [How the plugin works](#)

## 3. Compiling and building

### *Location of the build folder*

It is traditional to create the build folder inside the plugin folder and hence the following instructions locate the scripts one level up with two `..`. The following instructions follow this.

For my purposes, I prefer to keep the build folder out of the plugin folder so that it does not get included in the github presence. I replace the `..` with the file string of the JavaScript\_pi folder. The package builder looks for the data folder (and its included scripts folder) to be installed with the plugin one level up from the build folder. You therefore need to have a copy of that folder in place. I insert into these folders links to the copies of the files in the plugin folder.

### Windows

Consult the [OpenCPN instructions here](#).

### Linux

```
$ mkdir build
$ cd build
$ cmake ../ (note the two dots and forward slash)
$ make
$ sudo make package
```

You will find the package in the `'_CPack_Packages'` folder.

### MacOS

The developer tools and, notably, wxWidgets were installed [as described here](#).

### Building from terminal

```
$ mkdir build
$ cd build
```

```
$ export MACOSX_DEPLOYMENT_TARGET=10.09
$ cmake .. (note the two dots)
$ make
$ make create-pkg
```

## Building using Xcode

You will need a working Xcode IDE on your Mac, which you can establish using [the guide here](#).

You do not need a build of wxWidgets as the necessary files are provided in the *JavaScript\_pi/buildosx* folder.

Within your copy of the JavaScript\_pi folder, create a directory for the build, called, say, build-Xcode. In terminal:

```
$ cd build-Xcode
$ export MACOSX_DEPLOYMENT_TARGET=10.14
$ cmake -G Xcode ..
```

If this runs without issue, you will then find in this folder your Xcode project package JavaScript\_pi.xcodeproj. Launch this to open it in Xcode.

When you select the JavaScript\_pi target in Xcode and run it, it compiles and produces the dylib JavaScript\_pi.dylib, which you will find in the Debug folder within your build. To create the installer package, you first need to move the dylib up one level so that it is directly with your build folder. You can then run create-pkg to create the installer.

You can avoid this tediousness by automating it thus:

1. Select the JavaScript\_pi target and then the Build Phases setting
2. Disclose the CMake PostBuild Rules. You will see just one line of shell script starting make -C
3. Add the following three shell steps where 'mybuild' is replaced by the name of your build directory:

```
1. cd mybuild
2. cp Debug/libJavaScript_pi.dylib libJavaScript_pi.dylib
3. make -C . -f CMakeScripts/create-pkg_cmakeRulesBuildPhase.make$CONFIGURATION
   OBJDIR=$(basename "$OBJECT_FILE_DIR_normal") all
```

With this addition, when you run the build of JavaScript-pi, it will carry on and build the installer automatically.

### The Xcode 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 break points, 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.

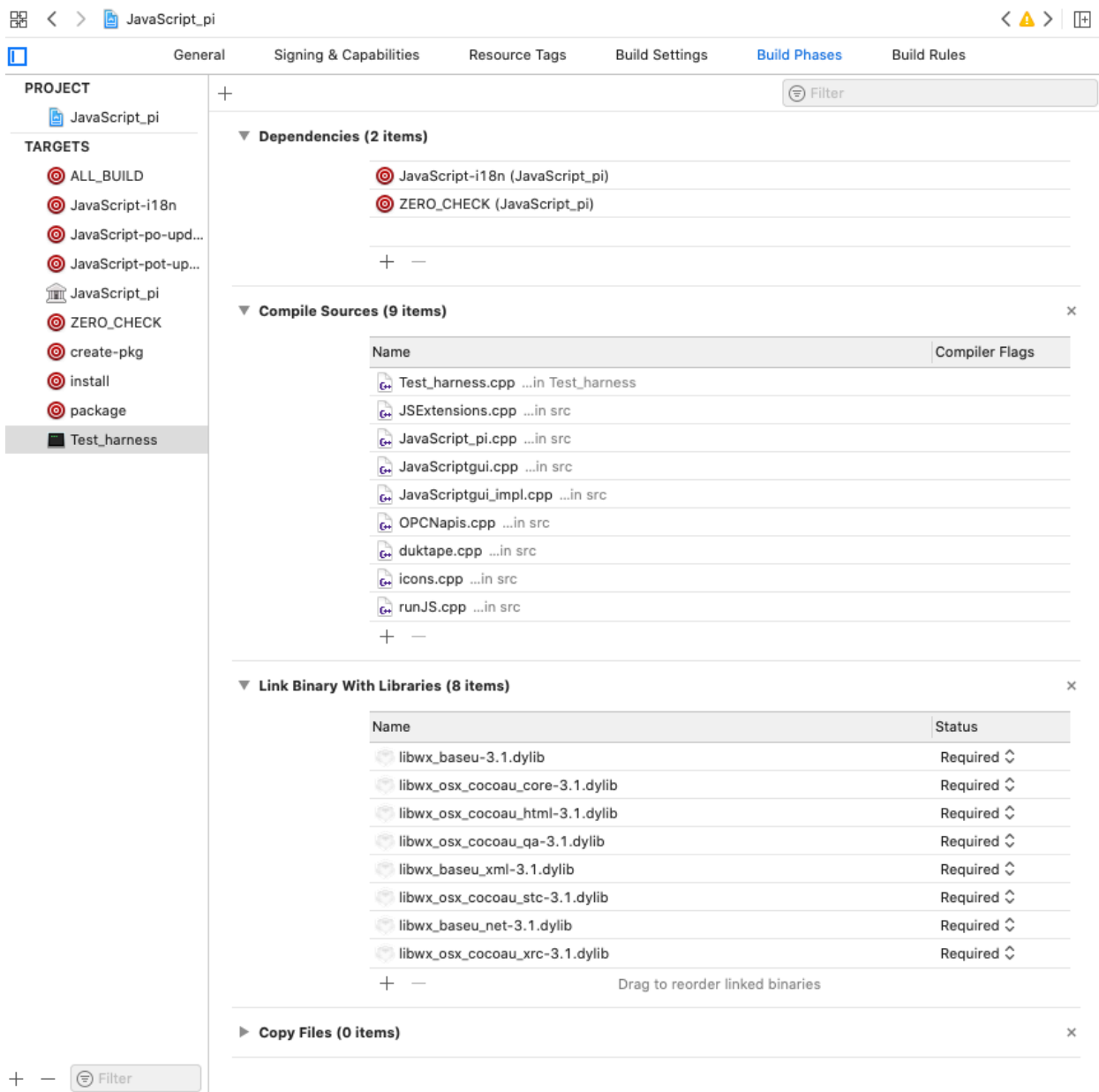
### ***Building the test harness in Xcode (verified for Xcode v11.6 & wxWidgets 3.2)***

To establish the test harness in Xcode

1. Select File-> New > Target...
2. Select target type of *Command line tool* and click on Next
3. Enter *Test-harness* as the product name and Finish. This creates a target of Test-harness and a yellow group called Test-harness, which will contain a dummy main.cpp
4. Control-click on this group to add files and choose *Test-harness.cpp*, which is located in the Test\_harness folder. Delete and remove to the trash the provided dummy main.cpp
5. Select target Test-harness
6. For the next steps it is best to open a second window (File>New>Window), so you can have both the settings for building the plugin and the test harness side-by-side.
7. In Build settings, select the All tab to disclose what is needed
8. Copy the following settings from the JavaScript-pi target Build Settings and paste them into the equivalent setting in the Test\_harness settings. To copy all the settings, click on them once so they are selected and copyable without opening them as a list - otherwise you would have to move them one at a time.
  1. Within *Search Paths*, the *Header search paths*
  2. Add to the search paths the folder *JavaScript\_pi/buildosx/wx\_includes* which contains wxWidgets headers. You can add an empty line using + and then in Finder drag the folder into this space.
  3. Copy across within *Apple Clang - Preprocessing*, the *Preprocessor Macros*
  4. To the Preprocessor macros add an extra line to define: `IN_HARNESS`
9. In Build phase settings
  - A. add Compile sources using the + button to add the following files from the group *Source Files*:
    1. JavaScript\_pi.cpp
    2. JavaScriptgui.cpp
    3. JavaScriptgui\_impl.cpp
    4. JSExtensions.cpp
    5. OPCNapis.cpp
    6. optional.cpp
    7. duktape.cpp
    8. icons.cpp
    9. JSlexer.cpp
    10. JSdialog.cpp
    11. functions.cpp
    12. toolsDialogGui.cpp
    13. toolsDialogImp.cpp
    14. and from the *Test\_harness group*, *Test\_harness.cpp*
  - B. Add Dependencies selecting
    1. JavaScript-i18n
    2. ZERO\_CHECK

10. To build the test harness, you will need to link it to the required wxWidgets libraries normally provided through OpenCPN. They are available in JavaScript\_pi/buildosx
  1. In the *Build Phase* tab, disclose the *Link Binaries With Libraries* list and drag all the libraries from *buildosx/wx\_libs* (8 files as of now) into this list.
  2. In *Build Settings* within *Search Paths*, open *Library search paths*
  3. Drag the icon for *JavaScript\_pi/buildosx/wx\_libs* into the field. It may insert the full file path or something like  $\$(PROJECT\_DIR)/for\_MacOS/wx\_libs$

NB When the plugin is built as a dylib, we have to include in the compile list all 145 source files for scintilla as we cannot link its library into the installer. For the test harness, we can link them from the stc library, so it is only necessary to compile the plugin code files plus the test harness itself. It should now look something like this:



You can now build the test harness and the console window should open. While running the test harness, the full riches of Xcode are available to insert break points and inspect variables, etc.

When the `require` function is given a simple module name, it looks for the scripts library build into the plugin. When running the test harness, it attempt to load the module from

the library in the OpenCPN application. So it is necessary that the plugin has been installed in the application first and that it is located within the applications folder.

### **Duktape test**

There is a folder JavaScript\_pi/Duktape which contains a command-line utility for testing Duktape stnd-alone. You could build a separate Xcode target `Duktape test` for it. I have not used this after an initial check, preferring to do testing in the test harness as described above.

## **4. Duktape extensions**

The JavaScript embedded engine has little access to its environment and performs no input or output. I have, therefore built in various extensions as functions.

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

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

## **5. Duktape OpenCPN APIs**

The extensions that provide the OCPN APIs work similarly and are to be found in the file `OPCNApis.cpp`. The file `opcnpn_duk.h` contains the definitions of the classes 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.

## **6. The script window**

The console has been created with `wxFormBuilder` as usual.

The script and output windows are of type `wxStyledTextCtrl`. This requires the Scintilla package, which is an optional extra part of `wxWidgets` not included in OpenCPN.

For Windows and Linux builds, it is sufficient to search the `stc` library to include the required parts.

For the MacOS test harness it is necessary to search the `stc` dylibs. For the Mac OSX OpenCPN plugin, I have found no way of including the extra libraries as the plugin loader only looks for the one plugin dylib. To resolve this, we include the source code files (all 148) and compile them together with the plugin.

The following macros need to be defined:

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

The script window uses 'lexing' of the script to aid understanding. Various words are coloured, as described in the user guide. This is set up in the function `JSlexit()` included in the file `JSlexit.cpp`. I had help learning how to use multiple keyword lists through [this post](#).

The JavaScript plugin now has a decent scripting window.

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 dummyming 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 `wxTextCtrl` but it is much less satisfactory. I favour reducing the size of Scintilla.

## 7. The require function

Duktape provides a framework in which to implement a JavaScript `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.

## 8. How the plugin works

This section provides a description of how the plugin works in relation to the JavaScript engine. It does not cover standard OpenCPN plugin matters.

### Major changes in version 0.4

Version on 0.4 introduced the ability to have any number consoles, each with their own instance of the JavaScript engine. This required a very major re-write of the code and much tidying up was undertaken in this process.

All code that relates to a console is now a method of the `Console` class. Data used to control the execution of the JavaScript were previously in a single structure of class `JS_control`. These have now been included in the `Console` class so a separate instance exists for each console. The few variables which apply in all instances have been moved into the JavaScript class.

### General description

It is often necessary to find the `JavaScript_pi` instance. Its constructor places a pointer to itself in a global variable

```
JavaScript_pi* pJavaScript_pi;
```

Each console is implemented as an instance of the `Console` class. The consoles are chained in a linked list with the first one found at

```
Console* pConsole = pJavaScript_pi->mpFirstConsole;
```

And the next in the chain at `pConsole->mpNextConsole`.

Each instance of the engine uses a heap. For us, the most important aspect of the heap is the context which contains the Duktape stack - not to be confused with the C++ stack. All our actions are stack-based and care is needed to pop off the correct number of items at the right moment.

All communication with the JavaScript engine is by calls to the Duktape API, which all start with `duk_` and reference the relevant context declared as

```
duk_context *ctx;
```

When the user clicks on the Run button, the plugin calls the `run(wxString script)` method, which, in summary, does the following:

1. Initialises the attributes that will be used during the run of Duktape
2. Creates a Duktape heap and records a pointer to the context



3. Initialises the JS\_control structure, which is where the plugin stores information about the state of operations
4. Starts a timer in case of loops etc
5. Invokes the Duktape engine to compile and run the script. During execution of the script, API calls to plugin functions may be called. These may set up timers or other call-backs. These are noted in the Console.
6. On completion of the main script it cancels the timer and a check is made for errors, which are reported.
7. A check is also made using the `isWaiting()` method to determine whether a callback is outstanding. If not, the `clearAndDestroyCtx()` method is used to clean up and release the Duktape heap.
8. When the JavaScript plugin receives an action from OpenCPN, it searches the consoles to see whether it satisfies an outstanding call-back. If so, it will
  - a. Load the global object (the previously compiled script) onto the stack
  - b. check that it contains the required function
  - c. Call JS\_exec, which
    - i. Starts the timer
    - ii. Calls the function using `duk_pcall`
    - iii. Cancels the timer
    - iv. Checks for an error return
  - d. If there are no outstanding call-backs or action, we clean up, which includes restoring the Run/Stop button to Run

When the Duktape engine calls an API or returns an error, the Duktape context is available but it does not indicate which console this is for. The function

```
Console *findConsole(duk_context *ctx)
```

is used to search each console for a matching context and return it.

## exitScript()

When a Duktape process completely successfully, it returns `DUK_EXEC_SUCCESS`. If this is not true, an error has been encountered. To get out of the script when `exitScript()` has been called, we have to throw an error but we don't want an error message. To avoid this, we look for a flag `mStopScriptCalled` and, if set, return `STOPPED`. Only after this do we check for a real Duktape error.

## Time ticks

A `wxTimer` is run at one second intervals and used to check for timer events due.

## Timing out

The code for this. Is to be found in the file `duktape_timeout.cpp`. The duktape engine is set up to call the C++ function `JSduk_timeout_check` at regular intervals. If the allowed time has been exceeded, this function returns 1 else 0. It has to do this repeatedly while the stack unwinds. It clears down only on the first timed-out call, using a flag `pConsole->m_backingOut` to manage this.

When the Duktape engine calls `JSduk_timeout_check`, there is no context nor indication which script instance it is for. We make the assumption that only one instance will be running at a time and have planted a pointer to the console in the global `Console* pConsoleBeingTimed`. This is a safe assumption as I have no evidence that OpenCPN multithreads calls to plugins and a given console's actions are single threaded. If the plugin was compiled with `DUK_DUMP` defined, the JavaScript extension `JS_mainThread()` allows a script to check whether it is running in the main thread. I have never seen it otherwise.

## Closing consoles

Closing a console window turns out to be tricky. It is a bit like cutting off the branch of a tree you are sitting on. The solution adopted is to unhook the console from the chain of consoles and then add it to a bin of closed consoles. Then on the regular timer tick, a check is made to delete consoles from the bin. Check is also made during plugin deist to delete any consoles in the bin.

## Keeping the right console on top

The consoles are instantiated as wxDialogs but not modal. This is necessary to ensure they stay on top of the main canvas frame. An initial problem was that the dialogues overlapped each other in order of creation. When a console is activated, the wxSTAY\_ON\_TOP bit is turned off for the deactivating console and turned on for the newly active console. This provides the expected behaviour.

## Programmatically starting scripts

When a script is chained or a script in another console is to be run, simple calls to this process could lead to a build up of call nesting and running for long periods without yielding. To avoid the CallAfter( ) method is used, which schedules the call later. The call needs to be to a method and methods doRunCommand and doExecuteFunction handle these, invoking the underlying method to do the work.

## Inter-script communication

Scripts can communicate using OpenCPN messaging. However, the recipient must be listening when a message is sent. This plugin has another method of inter-script communication, described here.

When a script runs, it is passed a *brief*, which it stores in its console structure. When a stir script is initially run through the console Run command, it is given a null brief. A brief contains the following:

bool	.fresh	If true, this brief has not yet been seen by a script. A brief can be read multiple times by the first script it is passed to but the same brief may not be used by any successor script it might chain to. Just before the script runs, this is set to false. During wrapUp( ), run at the end of every script, if .fresh is still false, we know that no new brief was set during this script and .hasBrief is set to false, making the script unavailable to any successor script.
bool	.hasBrief	If true, theBrief contains a string, being the actual brief for the script.
wxString	.theBrief	The brief itself. If not a string, it should be represented as its JSON string.
bool	.callback	If true, a different briefing script is to be called back when this script or chain of scripts ends.
wxString	.briefingConsoleName	If .callback is true, this is the name of a different console that is to be called-back.

wxString	.function	The function to be invoked during the callback.
When a script terminates, the wrapup() method examines the brief and, if it finds .callback true, will attempt to invoke the specified function running the specified console. It does this by calling its onExecute() method, passing to it a ConsoleCallbackResult structure, which is as follows:		
Completions	.resultType	The type of the result. This might be one of ERROR, DONE or STOPPED,
wxString	.result	The result from the script, except that if .resultType is ERROR then the error message.
wxString	.function	The function to be invoked during the callback.

If a scriptChain() statement, this preempts any callback and the brief is passed to the successor script - perhaps with the brief itself updated by the call. The callback details are thus inherited by the chained-to script. Thus when a console sets up a callback from another console, the callback is made from the last script of any chain of scripts that run in it.

When a console callback is to be made, the calling back console has to check that the console to be called still exists and that it is busy - hopefully awaiting the callback. If the specified function is no longer available, the call-back console will throw an error.

## 9. Error handling

How an error is handled depends on where it occurs.

### Error in the plugin code when no JavaScript is running

This is the simplest situation and the error can be displayed in the output window using `pConsole->message(int style, wxString messageAttribute, wxString message)`

The style is one of

```
enum {
    STYLE_BLACK,
    STYLE_RED,
    STYLE_BLUE,
    STYLE_ORANGE,
    STYLE_GREEN
};
```

Two strings follow being the message to be displayed and a newline is appended by the function.

### Error in the main script

`run(wxString script)` checks for an error return and displays the accompanying message which has been left on the stack, using `message` as above.

### Error detected in C++ extension or OpenCPN API code.

The C++ code should push an error object onto the Duktape stack and then throw a Duktape error with `duk_throw()`. Do not use the C++ `throw()` - that will kill OpenCPN!

### On call-back, the plugin detects error before invoking function

The plugin can display a message using `message` as above. It must then clean up using `clearAndDestroyCtx(ERROR)`.

## Function invoked during call-back throws an error

A called-back function is executed with the `executeFunction()` method. If an error is thrown during execution of the function, it returns `ERROR (-1)` indicating the plugin should clean up.

This is also the route taken if C++ code invoked by the called-back function throws an error.

## 10. Testing

There is a set of test scripts in the folder `Test_scripts`. Within this folder is a script `00_main.js` which is a test manager through which you can run the tests individually or all of them as a sequence.

To use this, set the current directory to be this folder using the plugin tools, load `00_main.js` and run it.

The test manager requires that many aspects of the plugin are working sufficiently, especially timers, dialogues and console calls. If it is not working sufficiently well, you will need to run the relevant script directly from its file.

The test manager running all the tests is a thorough check on the functioning of the plugin.

## 11. Trouble-shooting and debugging

### Preferences panel diagnostics tab

There are presently two diagnostic functions available.

#### Dump

This opens a new window and dumps selected diagnostic information

1. Environment and version information
2. For each console, selected attributes, including the addresses used to chain the consoles together.
3. If a console has a Duktape context, the Duktape stack is dumped. In this can be seen global JavaScript variables, names of functions and the stack stacking.

#### Clean and dump string

During development we encountered several problems with difficult character codes. The Duktape engine only accepts 16 bit characters as defined in the ECMA standard.

`wxWidgets` as used by `OpenCPN`, on the other hand, uses an extended character set.

Further, if scripts have been prepared or edited in word processors, they may include many esoteric characters such as "smart quotes". These all throw Duktape. To deal with this, a script is cleaned using the function

```
wxString JScleanString(wxString given)
```

which translates know unacceptable charrs into the most likely acceptable to Duktape. But we have had problems, especially with the Windows environment.

If you encounter a problematic character, this tab lets you examine the character coding before and after cleaning and should help identify any character not being translated satisfactorily.

## Obtaining dumps from with code

It is also possible to obtain dumps at an individual console level if the plugin has been compiled with

```
#define DUK_DUMP true
```

### From within C++ code

The console method `dukDump()` returns the current Duktape stack dumped into a string.

The console method `consoleDump()` returns a string being a dump of the console, including the Duktape stack.

You can display these as you will, perhaps using TRACE such as

```
TRACE(3, pConsole->dukDump());
```

### From within JavaScript

The Duktape stack can be returned as a string using `duktapeDump()` and `consoleDump()`.

Example:

```
print("Near end of script ", duktapeDump());
```

## Throwing an error with C++ code

There is a function `JS_throw_test(int1, int2)` which returns the sum of the two arguments except that if the two arguments are equal, it throws an error within the C++ code. This can be used to check correct functioning of this process.

## Re-entrancy and checking for the wx\_widgets main thread

To avoid re-entrance issues, it is assumed the plugin is always running on the main `wx_widgets` thread. There is a JavaScript function `JS_mainThread()` which returns true if this is the case.

I have never seen anything other than main thread but, if in doubt, this function could be used to check.

However, it seems that if the main script sets up a call-back, such as by `OCPNonMessageName()`, that may be called before the manuscript has completed. A flag `mRunningMain` is set until the main script has completed to prevent calls to `clearAndDestroy()` taking everything down too soon.

Similarly, a flag `mTimerActionBusy` is used to guard against timer actions piling up on each other.

Descriptions of the above JavaScript functions are omitted from the user guide. The script window lexer colours them orange to warn users off.

## Tracing

There are many trace statements in the code of the form

```
TRACE(int level, wxString message);
```

These have been extremely useful to observe what is going on.

`level` is the level of tracing with 1 for key points that should always be traced and a higher number such as 5 for detailed tracing within loops etc.

The header file `trace.h` sets this up with the macro `TRACE_LEVEL` set to the required level. All trace statements whose level does not exceed `TRACE_LEVEL` will output their message. Set `TRACE_LEVEL` to 1 for key points only and to a higher number for detailed tracing of iterating loops etc.

If you want to trace just a point or two, use a level of 1 so this will avoid the other numerous `TRACE` statements.

When `TRACE_LEVEL` is set to zero, there is no tracing and the tracing is omitted entirely from the compiled code. **This should be the case for releases.**

When `TRACE_TO_WINDOW` is false, tracing output is sent to the OpenCPN log file, so avoid over filling it. When running in the harness, output is sent to `stdout`. However, neither of these are desirable when long tracing is occurring. When `TRACE_TO_WINDOW` is true, a separate window will be opened to receive the trace. Be aware that this window is lost if OpenCPN quits or crashes.

# Appendices

## Road map for future development

I am interested in working with others to develop ideas for this plugin. I set up a Slack workspace 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?

## Document history

Version	Date	
<b>0.1</b>	19 Jul 2020	Initial version to accompany the plugin v0.1
<b>0.2</b>	20 Aug 2020	Update to accompany plugin release v0.2
<b>0.3</b>	14 Nov 2020	Update to accompany plugin release v0.3
<b>0.4</b>	7 Feb 2021	Update to accompany plugin release v0.4