

JavaScript Plugin Technical Guide

Tony Voss

Version 3.0 28 Mar 2024 - [document history here](#)

Contents	Page
1. Introduction	3
Before release into the OpenCPN catalogue	3
2. The JavaScript Engine	3
3. Compiling and building locally	3
4. Duktape extensions	3
5. Duktape OpenCPN APIs	4
6. The script window	4
7. The require function	4
8. How the plugin works	4
Major changes in version 0.4	4
Changes for v0.5	5
Major changes in version 2.0	5
Major changes in version 3.0	5
General description	5
Callbacks	6
OpenCPN plugin manager originated	7
wxWidgets timer event	7
Dialogue buttons	7
<ObservableListener>	7
One-shot vs multi-shot callbacks	7
Adding callbacks	7
exitScript()	8
Time ticks	8
Timing out	8
Closing consoles	9
Keeping the right console on top	9
Programmatically starting scripts	9
Inter-script communication	9

MessageBox	10
9. Windows scaled displays	10
wxWidgets prior to v3.2	11
wxFormBuilder	11
10. Configuring settings for console parking	12
11. Error handling	12
Error in the plugin code when no JavaScript is running	12
Error in the main script	12
Error detected in C++ extension or OpenCPN API code.	12
On call-back, the plugin detects error before invoking function	12
Function invoked during call-back throws an error	13
12. Testing	13
13. Updating the Duktape engine	13
14. Trouble-shooting and debugging	13
Tools panel diagnostics tab	13
Dump	13
Clean and dump string	13
Obtaining dumps from with code	14
From within C++ code	14
From within JavaScript	14
Throwing an error with C++ code	14
Re-entrancy and checking for the wx_widgets main thread	14
Tracing	15
Debugging using Xcode	15
15. The built-in scripts	16
pgnDescriptors	16
Appendices	17
A Road map for future development	17
B Stand-alone test harness	17
Instructions for building the Xcode test harness for the plugin v0.4	18
Duktape test	20
Document history	21

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](#). However, v0.5 adopted the new standard build methods and were set up for me by Mike Rossiter.

Versions through to 0.4 were primarily developed on MacOS using Xcode and a stand-alone test harness that allowed me to run the plugin without OpenCPN.

Changes since v0.4 are highlighted in yellow.

Before release into the OpenCPN catalogue

Before releasing a new version of the plugin, make sure the following:

- Update the NMEA2000 descriptors if needed. [See here](#).
- Check compiler debug tables are excluded. See [Debugging using Xcode](#).
- In the plugin.cmake file set the following appropriately

```
set(PKG_VERSION 3.0.0)
set(PKG_PRERELEASE "") # Empty, or a tag like 'beta'
```
- In version.h you can set a build number, which is displayed in new console first scripts

```
#define PLUGIN_VERSION_COMMENT "Build -d"
```
- in ocpn_duk.h,

```
#define DUK_DUMP false
```
- in trace.h, exclude tracing

```
#define TRACE_YES false // if true, include TRACE code. Should be false in release
```

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 locally

Having taken a clone of the plugin, locate the appropriate script within the ci folder.

Follow the instructions and you will end up with a tarball ready for importing into OpenCPN.

The code contains various experimental features not ready for release. These are not compiled and in some cases their inclusion or omission is set up in the buildConfig.h header file.

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 `opcpn_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, e.g. 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 now included in all OpenCPN builds from v5.5

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.

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.

The `require` API first checks for build-in C++-coded components by calling the `loadComponent` function in `optional.cpp`. This returns `true` if it recognises the component. If it returns `false`, the API searches for the JavaScript to load. In certain cases there are both C++ and JavaScript components. In these cases, `loadComponent` returns `false` so that both are loaded.

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 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.

Changes for v0.5

V0.5 uses the API 117 and makes extensive use of the extended API calls for waypoints and routes. It also implements additional APIs implemented in OpenCPN v5.5 to facilitate various JavaScript tasks.

OpenCPN v5.5 for MacOS moved up to wxWidgets v315 and so the plugin does likewise. There are a number of issues with v315 around closing windows. In addition to closing a window, it seems necessary to use sequences such as:

```
pWindow->Destroy( );  
delete pWindow;  
pWindow = nullptr;
```

OpenCPN v5.5 for MacOS includes the Scintilla library that was missing in v5.2. This means the Scintilla source no longer needs to be compiled into the plugin.

Major changes in version 2.0

Major functional changes were introduced in V2.0. Also similar things were being done in different places. These have now been regularised and the same function/method used in all applicable cases.

Decision making after some JavaScript has been run has been regularised. Boolean status flags are now held in mStatus.

Major changes in version 3.0

More major changes were introduced in v3.0.

Character code issues

Now that scripts can parse a position, such as 12° 27'N 019° 23.450'W, attention had to be given to the wide variety of formats used and, especially, of characters used for the degree symbol - degree, masculine ordinal, superscript 0 etc. All these are changed to degree through the JScleanString function.

However, there is particular problem with Windows, which will not accept the degree (\u00B0) as it is outside the basic ASCII set. To get around this, on Windows this character is changed to \u0007 on input and changed back to degree on output. These changes are scattered throughout the code.

The substitution used is defined in version.h. For non-Windows, the substitution is for itself.

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 console `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. After code has been run, the `afterwards()` method examines the Duktape completion return and the statuses and returns whichever of the various outcomes to prioritise. `wrapUp()` is then called to take the necessary action, such as clearing down the context or waiting for a callback.
7. On completion of the main script it cancels the timer and a check is made for errors, which are reported.
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. After the code has run, the same `afterwards()` and `wrapup()` methods are used to determine what happens next.

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.

Callbacks

Scripts can arrange for functions to be invoked via callbacks, such as when an OpenCPN event happens, a dialogue is completed or on a timer.

Such callbacks could trigger other callbacks and this could lead to a very deep call stack or even a loop. To avoid this, call-back functions are executed using the `wxWidgets` `CallAfter` method, whereby they are queued and called after the present thread has completed.

Information about how to handle the callback is stored in the console.

How the callback is implemented, depends on the type of callback.

OpenCPN plugin manager originated

Plugin manager calls the relevant method of the plugin. This method walks the console chain to determine which consoles are awaiting the call and calls the handling JavaScript function.

wxWidgets timer event

The plugin runs a one second timer. On each event, the plugin OnTimer event handler checks to see if any consoles need to have a script automatically run and then walks the chain of consoles to look for those waiting for this time. If so, it calls the handling JavaScript function.

Dialogue buttons

These use a custom wxButton event that includes the pConsole pointer thus identifying the console. The code is in JSdialog.cpp.

<ObservableListener>

OpenCPN v5.8 introduced the <ObservableListener> type for handling incoming data/messages.

The plugin consoles each maintain an array of <streamMessageCntl>, which contain the information needed handle the message and the listener itself. The handling method in the console is invoked through a lambda expression. This has to be set up in a method of the console. The Duktape API must therefore be a console method but the Duktape API invokes a c++ function in OPCNapis.cpp giving it only the Duktape context ctx. This minimal function uses its given ctx to identify the console and then calls the console's method to set up the listening.

One-shot vs multi-shot callbacks

Up to the plugin v2.1, all callbacks were single shot, i.e. when the callback was invoked, the listening was stripped down so any subsequent event would not call the function again. Any script that wanted a handler function to be called repeatedly needed to set up a new callback within the handler. Because of this, it was possible to miss an event while processing a previous one.

For events that happen regularly, such as receiving fix data, this was not a concern. Another event would be along shortly.

While developing the plugin v3, it was found that some events only occurred once and they could be missed. Notably, if a NMEA2000 request is send that all stations respond, the responses come only once from each and close together.

To cope with this, enduring callbacks were introduced for many APIs. For these, the listening is not stripped down so subsequent events call the handler again. For the script, they are distinguished by All in their name.

```
OCPNonNEA2000()           // one shot  
OCPNonAllNMEA2000()       // enduring
```

If a script wants to cancel an enduring callback, it can do so by calling the first form above with no parameters.

For both API variants, the code is mostly identical. Such functions are given an extra parameter which is true if an enduring call back is to be set up. The two APIs such as those above each invoke an outer function which calls the function that does the real work adding the extra parameter.

Adding callbacks

If you want to add further callbacks, it will be helpful to know how to do this. Here I list the steps for a very simple case - adding the onCloseButton callback.

1. In JavaScriptgui_impl.h add the console attribute needed to handle the callback. This will include at a minimum the name of the function to be called.
2. In same file, add dumping this value to the consoleDump() method. It is often useful to be able to inspect this.
3. Add to the clearCallbacks() method, clearing the attribute,
4. In the isWaiting() method add a test so it returns true if a callback is awaited.
5. Wherever notification of the event is received, process. In our example of the onCloseButton, the code is:

```
if (m_closeButtonFunction != wxEmptyString){
    wxString function = m_closeButtonFunction; // remember
    m_closeButtonFunction = wxEmptyString; // no longer waiting
    Completions outcome = executeFunctionNargs(function, 0);
    // check if script has other callbacks waiting else wrapup
    if (!isBusy()) wrapUp(outcome);
    return;
}
```

6. In JSextensions.cpp (if a non-OCPN function) or OCPNapis.js (if an interaction with OCPN) add the Duktape API. See examples. The aim after error checking is to set up the attribute added in step 1 that will be used to handle the callback.
7. In JSlexer.cpp add the name of the API so it gets coloured appropriately in the user's script.

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 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.

The function JSduk_timeout_check is in functions.cpp. The starting, stopping and management of the timeout function is handled through methods of the console.

To set up the calling of JSduk_timeout_check, the following is appended to duktape.h

```
/* added by Tony Voss for time-out support */
#define DUK_USE_INTERRUPT_COUNTER
#define DUK_USE_EXEC_TIMEOUT_CHECK JSduk_timeout_check // our timeout
check function
duk_bool_t JSduk_timeout_check(void *udata);
```


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 in the bin. A check is also made during plugin deinit to delete any consoles in the bin.

Keeping the right console on top

Prior to wx3.2, the consoles were 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. From v0.5 the wxSTAY_ON_TOP bit is then turned off again. This provides the expected behaviour.

With the move to wx3.2 the behaviour of windows changed fundamentally - particularly under MacOS. All windows are now created as type wxFrame rather than wxDialog. By default they have the wxFLOAT_ON_PARENT style, which keeps them on top of the OpenCPN frame but not on top of other applications. There is an option in the Consoles page of the Tool to turn this off. This has immediate effect, including for existing consoles.

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 this, the `CallAfter()` method is used, which schedules the call later. The call needs to be to a method and the console 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 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 <code>wrapUp()</code> , 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, the Brief 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 <code>HAD_ERROR</code> , <code>DONE</code> , <code>STOPPED</code> or <code>CLOSED</code>
wxString	.result	The result from the script, except that if <code>.resultType</code> is <code>HAD_ERROR</code> then the error message.
wxString	.function	The function to be invoked during the callback.

If a `scriptChain()` statement is executed, 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.

MessageBox

The `messageBox()` feature added in v2.0 presented a number of challenges, given that it uses a modal window to display the message. The advantage of the modal display is that it simplifies scripts that proceed through steps.

Complication arise when one script closes another console that has a modal `messageBox` displayed. Initially, I used `wxMessageBox()` for this feature but it turns out this window is very hard to close other than by clicking on one of its buttons. See various discussions on the web.

In the end, I use a `wxDIALOG()`. It is more complicated to construct the dialogue but it has the advantage that it can be closed externally. If `consoleClose()` finds a `messageBox` dialogue is active, it closes it with `EndModal(-1)`. The `messageBox` code checks for this, sets a special status of `CLOSING` and throws an error. This unwinds the stack. In afterwards() this is picked up and passed on as `CLOSED`. Instead of displaying the error message, it moves the console to the bin.

9. Windows scaled displays

MacOS handles the resolution of the display automatically and transparently.

Under Windows, the application has to take the scale of the display into account. `wxWidgets` v3.2 handles this mostly but not where the size or position of window is explicitly set.

There are built in methods to adjust for the display scale from Device Independent Pixels (DIP), namely

```
FromDIP()
ToDIP()
GetDPIScaleFactor()
```

Under MacOS, `FromDIP()` and `ToDIP()` have no effect - the returned `wxPoint` or `wxSize` is unchanged. `GetDPIScaleFactor()` does, however, return the scale factor. Source file

scaling.h defines a macro `SCALE(window)` which returns the scale for Windows and 1.0 for MacOS.

The plugin stores positions and sizes of windows in DIP and converts them to or from DIP as required. The values in the .ini file are these DIP values.

wxWidgets prior to v3.2

Scaled displays were introduced in wxWidgets v3.2. When building the plugin for earlier versions, it is necessary to avoid compilation errors.

The file `version.h` builds a boolean macro `SCREEN_RESOLUTION_AVAILABLE` which is set accordingly. When this is `false`, dummy versions of the `FromDIP` and `ToDIP` methods are compiled.

wxFormBuilder

A particular problem arises with windows set up using wxFormBuilder. Any window positions or sizes explicitly set in wxFormBuilder will be wrong for Windows scaled displays.

The first step in dealing with this was to examine the generated code and locate all constants therein. I found several could be eliminated by using default -1 values in wxFormBuilder where I had set explicit values unnecessarily.

I then created a window method which fixes up the wxFormBuilder generated code. It starts like this:

```
void fixForScreenRes(){    // fix up sizes for screen resolution
    double scale = SCALE(this);
    if (scale == 1) return;    // nothing to do
    wxSize size;

    m_notebook->SetSize(FromDIP(m_notebook->GetSize()));

    // Consoles tab
    Consoles->SetSize(FromDIP(Consoles->GetSize()));
    m_newConsoleName->SetMinSize(FromDIP(m_newConsoleName->GetSize()));
    m_oldNames->SetMinSize(FromDIP(m_oldNames->GetSize()));
    m_changedName->SetMinSize(FromDIP(m_changedName->GetSize()));
```

Fixes need to be applied to all occurrences. It is necessary to use `SetMinSize()` rather than `SetSize()` and best to use Client sizes where possible. There may not be a `Get` function for some attributes, such as `wxTextCtrl` `Wrap` parameters. In those cases the value and its adjustment has to be written into the adjuster method removing the effect of changing it in wxFormBuilder.

Finding them all without having a Windows scaled display is tricky. So there is a simulator which can be conditionally compiled in when required for testing on a non-scaled display. It overloads the functions with ones that perform a developer-chosen scaling. The windows thus appear larger than they would and this can be used to locate the needed corrections. When the enlarged windows looks correct, it should be correct on a scaled display.

10. Configuring settings for console parking

Consoles have a minimum size and the plugin can minimise a console and park it in the canvas frame's top bar. This requires constants which are platform dependent. This section describes how to configure these constants for a particular platform.

The constants are defined in the `consolePositioning.h` header file, are compiled conditionally depending on platform and are as follows

CONSOLE_STUB	Width to be left for things other than the console name
PARK_LEVEL	Level at which consoles are parked, relative to the frame
PARK_FIRST_X	Indent of first parked console from left-hand edge of the frame
PARK_SEP	Horizontal separation of parked consoles

To establish the necessary values for a given platform, use the Parking panel in the plugin tools. This allows you to customise the settings by taking you through manually parking two consoles. You can then display the appropriate above settings for incorporation into the `consolePositioning.h` file to set the default for a specific platform.

Please report any platform-specific changes so they can be included in the master file.

11. 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 message)
```

The style is one of

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

A newline is appended to the message 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 then returns.

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.

12. 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.

13. Updating the Duktape engine

The Duktape engine is compiled from a single file `duktape.cpp`.

To install a new version, replace this and its include files `duktape.h` and `duk_config.h`.

You will need to append the `timeout` function to the new `duktape.h` file. [See Time Out above](#).

The function `formErrorMessage(duk_context *ctx)` in the `functions.cpp` files is used to extract an error traceback from the Duktape stack. According to the Duktape documentation, this could be release dependent. If error tracebacks are not right, this might need adjusting.

14. Trouble-shooting and debugging

The following one-line script will display plugin version details

```
print(JSON.stringify(OCPNgetPluginConfig(), null, "\t"), "\n");
```

Tools 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 known unacceptable chars 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
```

which is to be found in ocpn_duk.h

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 JavaScript function `JS_throw_test(int1, int2)` which returns the sum of the two arguments except that

1. if the two arguments are equal, it throws an error within the C++ code.
2. If the two arguments are both 0, it invokes the console's `throw_error` method.

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 main script has completed. A flag `mRunningMain` is set until the main script has completed.

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

A flag `mJSrunning` is set true while the timed callback is running. This is checked before a callback function is executed to avoid re-entering the JavaScript object code.

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.

Tracing is set up in `trace.h` with

```
#define TRACE_YES true // if true, include TRACE code.
                        Should be false in release
#define TRACE_MAX 30 // TRACE level range (Level 0 always traced)
#define TRACE_MIN 25
#define TRACE_TO_WINDOW true // if true, trace will be to a window
```

If `TRACE_YES` is false, tracing code is omitted from the compilation. **This should be the case for releases.**

Otherwise `TRACE` statements with a level in the range between `TRACE_MIN` and `TRACE_MAX` will be traced. Level 0 is always traced.

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.

When `TRACE_TO_WINDOW` is false, tracing output is sent to the OpenCPN log file, so avoid over filling it.

Debugging using Xcode

On MacOS it is possible to debug the plugin using Xcode. These instructions assume you have set up Xcode and have been verified for Xcode v14.

You need to build the plugin with the debugging tables. In the `ci/circleci-build-macos.sh` script, you will find

```
cmake \
  "-DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE:-Release}" \
  -DCMAKE_INSTALL_PREFIX= \
  -DCMAKE_OSX_DEPLOYMENT_TARGET=${MACOSX_DEPLOYMENT_TARGET} \
  -DOCPN_TARGET_TUPLE="darwin-wx32;10;universal" \
  -DCMAKE_OSX_ARCHITECTURES="arm64;x86_64" \
  ..
```

In the first line above, replace `Release` with `RelWithDebInfo` or `Debug`

Don't forget to change this back when finished.

Now build the plugin and install into OpenCPN as usual.

Launch Xcode and create a workspace or, if you already have an Xcode workspace, invoke it.

Within Xcode select the project navigator, extreme top left, by clicking on the folder icon. Then use the command `File > Add files` to add the `src` folder of source files. Click on the loaded folder and you will see the list of source and header files. Set breakpoints where you need.

There are two ways of proceeding here.

Attach to a running OpenCPN process

Have OpenCPN running and within Xcode choose `Debug > Attach to process` and choose your running OpenCPN.

Sometimes it is not possible to attach - no list of running processes appears. In this case, use the following...

Launch OpenCPN from within Xcode

Quit OpenCPN if running

Choose `Debug > Debug Executable...` and choose the OpenCPN application.

Close the window that opens.

Click on the run triangle and Xcode will launch OpenCPN.

Getting to your breakpoint

Within OpenCPN take whatever action is needed to reach your breakpoint. When the breakpoint is reached, you will find yourself back in Xcode where you can examine variables and set through the code.

I am set up so that Xcode is locked to one desktop and OpenCPN to another. When moving between the two, the appropriate desktop slides into view.

15. The built-in scripts

Various JavaScript functions can be loaded using, e.g,

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

If the string argument to `require` is an unqualified name, as above, the corresponding file with its .js type is loaded from the built in scripts folder. This script is compiled and must return a function.

pgnDescriptors

This script contains a copy of the NMEA2000 PGN descriptors from the Canboat project [available here](#). The copy is built into the descriptors object.

On Windows, the JavaScript engine fails to compile this if it contains non-ASCII characters, which it does. The `cleanScript` function cleans ö Ö ü & Ü to ASCII equivalents.

However, PGN 127498 GPS Almanac Data contains numerous Greek characters. This PGN has been removed and is available separately as a custom descriptor for non-Windows users. When updating the library, it is important to omit PGN 127498. Otherwise the whole function will fail to compile silently.

To update the descriptors

1. copy from the `pgns.json` file the contents inside `[]` of the PGNs array and paste into inside the `[]` of the descriptor array in the `pgnDescriptors.js` file.
2. Delete PGN 129541
3. Update the comments at the start of the `getDescriptor` function with the information at the head of the `pins.json` file, including the version number.
4. Copy the entire `getDescriptors` function, paste into a console and run. This checks it compiles OK. It should give a result of undefined.

Appendices

A 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?

B Stand-alone test harness

The early development of this plugin was done using a stand-alone harness that replaced OpenCPN. This allowed the plugin to be run without OpenCPN for exploration and detailed debugging.

This test harness was last used for v0.4. To use with later versions would require significant reworking. The instructions for building the test harness are included below should anyone want to follow this route.

This approach would require significant reworking because:

- V0.5 uses the extended API calls available in API 117. The test harness used API 116 calls. New stubs would need to be created in the test harness.
- The v0.4 plugin for MacOS was built using Xcode. The instructions for building into the test harness include copying settings from the building of the plugin installer.

Instructions for building the Xcode test harness for the plugin v0.4

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

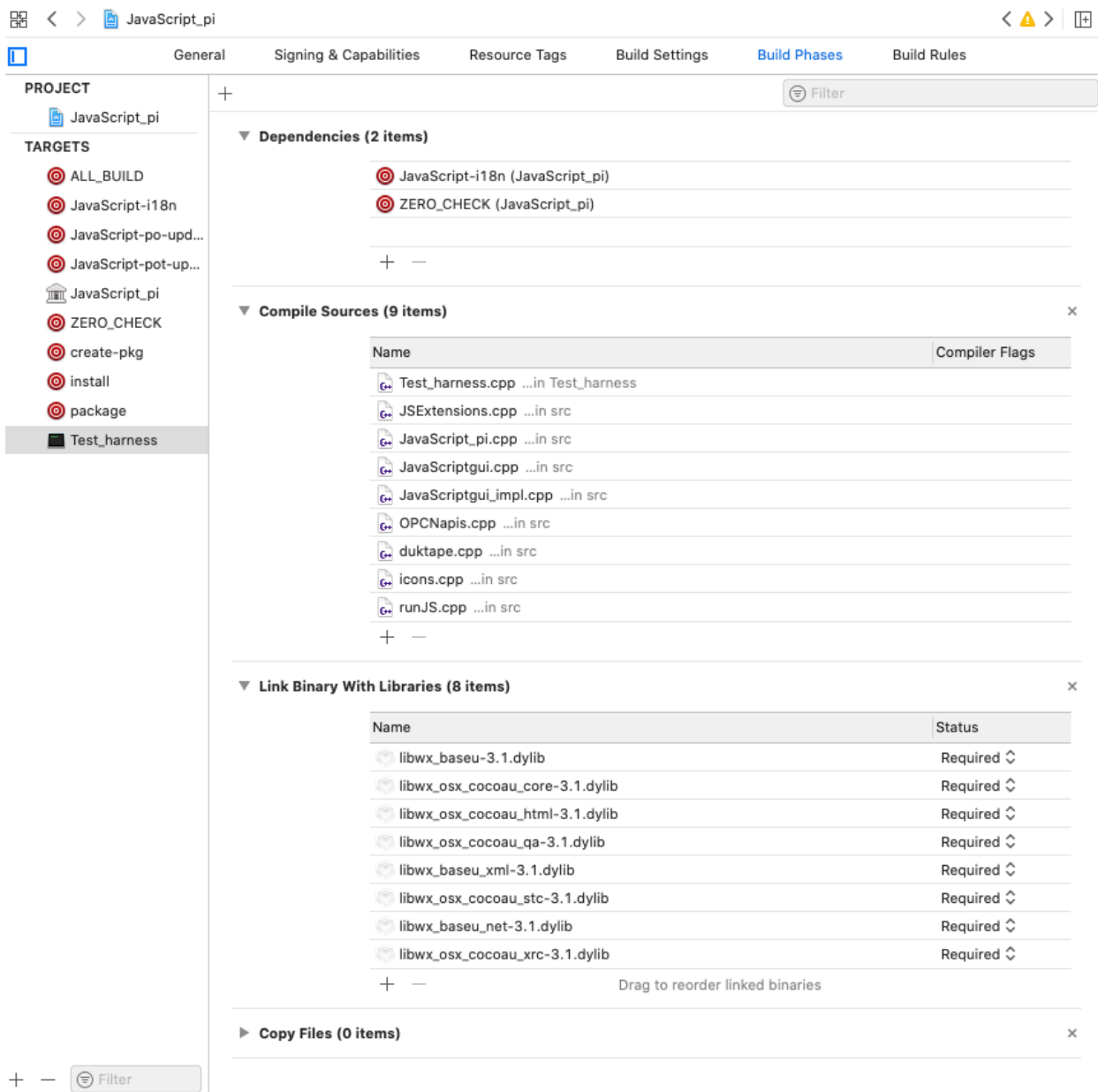
B. Add Dependencies selecting

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.

Document history

Version	Date	
0.1	19 Jul 2020	Initial version to accompany the plugin v0.1
0.2	20 Aug 2020	Update to accompany plugin release v0.2
0.3	14 Nov 2020	Update to accompany plugin release v0.3
0.4	7 Feb 2021	Update to accompany plugin release v0.4
0.5	26 Nov 2021	Update to accompany plugin release v0.5
1.0	26 Jan 2022	Re-issue to accompany plugin release v1.0
1.1	1 Mar 2022	Minor corrections and clarifications to accompany release v1.1
2.0	18 Jan 2023	Re-issue to accompany plugin release v2.0
2.1	14 Apr2023	Re-issue to accompany plugin release v2.1
3.0	28 Mar 2024	Re-issue to accompany plugin release v3.0
3.1	16 Dec 2024	Minor tweak to accompany plugin release v3.1