JavaScript Plugin Technical Guide

Tony Voss

Version 0.3 14 November 2020 - document history here

Contents **Page**

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.

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)

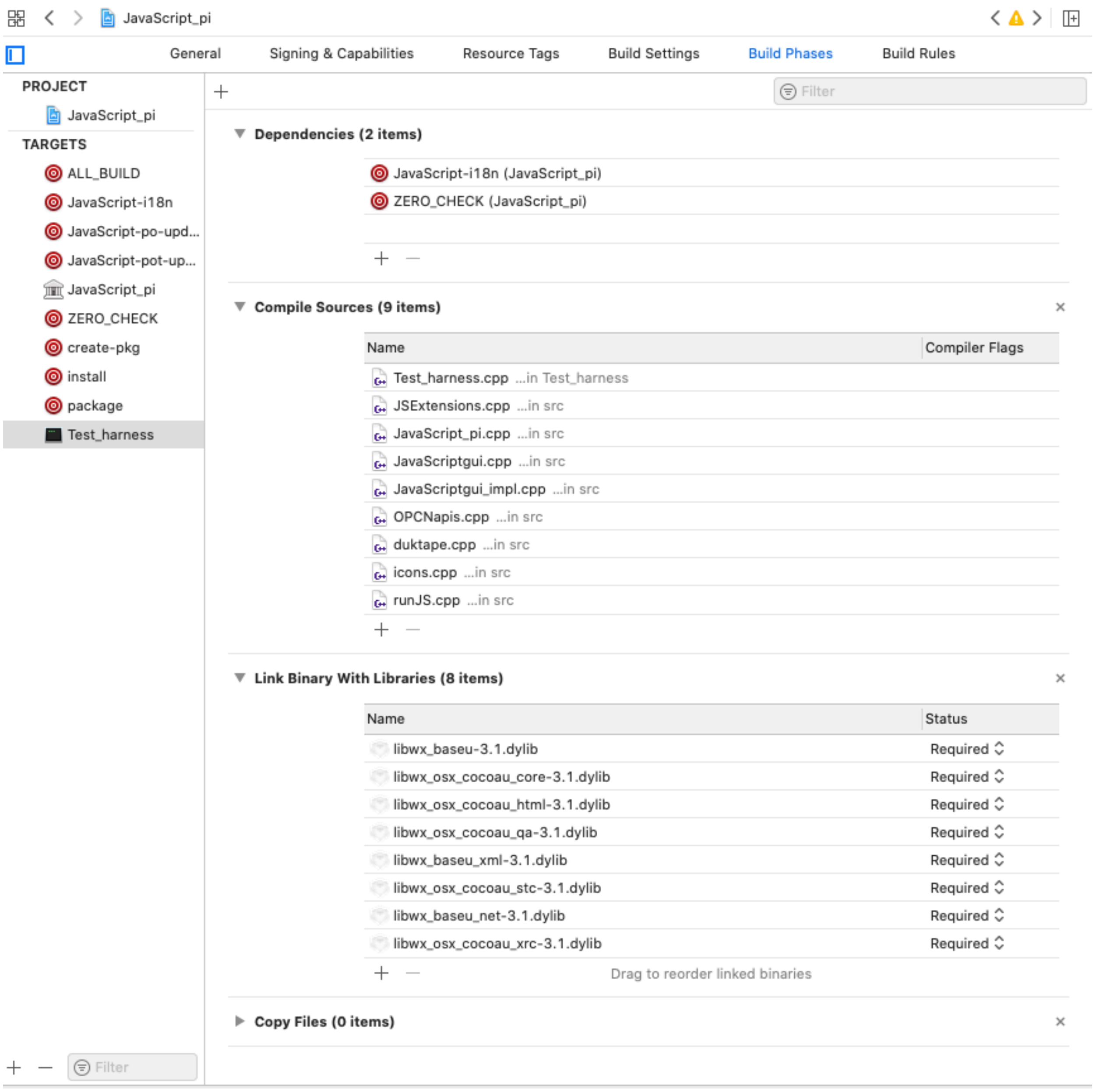
To establish the test harness in Xcode

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 to build 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. duktape.cpp
        7. runJS.cpp
        8. icons.cpp
        9. JSlexer.cpp
        10. duktape_timeout.cpp
        11. 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* 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 7 parts of the plugin plus the test harness itself.

It should now look something like this:



Dependencies (2 items)

- JavaScript-i18n (JavaScript_pi)
- ZERO_CHECK (JavaScript_pi)

Compile Sources (9 items)

Name	Compiler Flags
Test_harness.cpp ...in Test_harness	
JSExtensions.cpp ...in src	
JavaScript_pi.cpp ...in src	
JavaScriptgui.cpp ...in src	
JavaScriptgui_impl.cpp ...in src	
OPCNapis.cpp ...in src	
duktape.cpp ...in src	
icons.cpp ...in src	
runJS.cpp ...in src	

Link Binary With Libraries (8 items)

Name	Status
libwx_baseu-3.1.dylib	Required
libwx_osx_cocoau_core-3.1.dylib	Required
libwx_osx_cocoau_html-3.1.dylib	Required
libwx_osx_cocoau_qa-3.1.dylib	Required
libwx_baseu_xml-3.1.dylib	Required
libwx_osx_cocoau_stc-3.1.dylib	Required
libwx_baseu_net-3.1.dylib	Required
libwx_osx_cocoau_xrc-3.1.dylib	Required

Drag to reorder linked binaries

Copy Files (0 items)

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 `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 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() inlucuded 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 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.

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.

General description

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 does the following:

1. Creates a heap and is given a pointer to the context
2. Initialises the JS_control structure, which is where the plugin stores information about the state of operations
3. Submits the script to the compileJS function
4. The JScompile function
    - A. Cleans up the supplied script to remove unacceptable characters
    - B. Registers our extensions on the context (there are two sets of extensions)
    - C. Starts a timer in case of loops etc
    - D. Compiles and runs the script using duk_peval. The executing script may call one or more of the extensions or OCPN APIs.
    - E. duk_peval returns the outcome, which is either the result or an error message
    - F. We cancel the timer and act on the return.
    - G. The script may have set up call-backs. If so, we check whether those functions exist in the context. The user might not have provided the function to service a call-back.
    - H. We then check whether any call-backs are awaited [JS_control.waiting()]
    - I. JScompile returns this state.
5. If we are not waiting for anything, we clean up and destroy the heap and wait for the next user action.

6. If we are waiting for a call-back or for a window to be actioned on, the details will be held in the JS_control structure.
7. When the call-back arrives, we
    A. Check JS_control.JSactive is true to preclude call-backs being attempted when none set up, as might happen on plugin activation.
    B. Consult JS_control for instructions on what to do.
    C. If a function is to be called, we
        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

exitScript()

To get out of the script, this has to use `duk_throw()` but we don't want an error message. To avoid this, a flag is set in `JS_control` to treat this as a non-error.

onSeconds implementation

I tried and failed to get the wxTimer function working and so use the regular call-backs from OpenCPN to SetPluginMessage. Before checking whether it needs to process a requested message, the code examines the timer tables in JS_control.

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 `JS_control.m_backingOut` to manage this.

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

```
JS_control.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

CompileJS checks for an error return and displays the accompanying message which has been left on the stack, using using `JS_control.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 plugin detects error before invoking function

The plugin can display a message using JS_control as above. It must then clean up using `JS_control.clear()` and destroy the context.

Function invoked by JS_exec during call-back throws an error

JS_exec displays the error using `JS_control.display_error` and then returns false, 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. Trouble-shooting and debugging

The plugin has been compiled with

```
#define DUK_DUMP true
```

And so some debugging aids are available:

Viewing JS_control

The JS_control structure is where the plugin stores variables it needs in order to know what to do next or how to handle actions. It is defined in ocpn_duk.h.

You can dump to the output window selected elements of JS_control from C++ code by inserting the following at the required points:

```
 JS_control.message(STYLE_ORANGE,_("location"), JS_control.dump());
```

You can replace "location" with whatever to identify between different instances of this line.

You can dump this from within JavaScript with the following function call statement:

```
JS_control_dump("location");
```

You can omit the location parameter if not required.

Viewing the duktape stack and error objects

You can dump the duktape stack to the output window by inserting into C++ code:
`MAYBE_DUK_DUMP`

You can view any error object on the duktape stack by inserting into C++ code:
`ERROR_DUMP`

The error dump automatically includes the stack dump.

You can obtain the error dump from within JavaScript with the following function call statement:

```
JS_duk_dump();
```

Examples

```
onSeconds(timesUp, 3, "time is up");
JS_control_dump();

function timesUp(argument){
      JS_duk_dump();
      }
```

This script immediately displays

```
JavaScript JS_control dump
m_runCompleted:  false
Messages callback table
      (empty)
m_timerActionBusy:    false
Timers callback table
      timesUp    time is up
m_NMEAmessageFunction:
m_explicitResult:     false
m_result:
```

Then after 3 seconds it displays

```
JavaScript error dump
No error object

JavaScript Duktape context dump:
ctx: top=0, stack=[]
```

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.

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.

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

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	
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