JavaScript Plugin Technical Guide

Tony Voss

Version 0.2 - <u>document history here</u>

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 underlined described here. This is noted because this is where the Cmake files originated.

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

Compiling and building

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 ..

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 build folder. The package builder picks up the data folder (and its included scripts folder) to be installed with the plugin and looks for it one level up from the build folder. You therefore need to have a copy of the folder in that place. I insert links to the copies of the files in the plugin folder.

Windows

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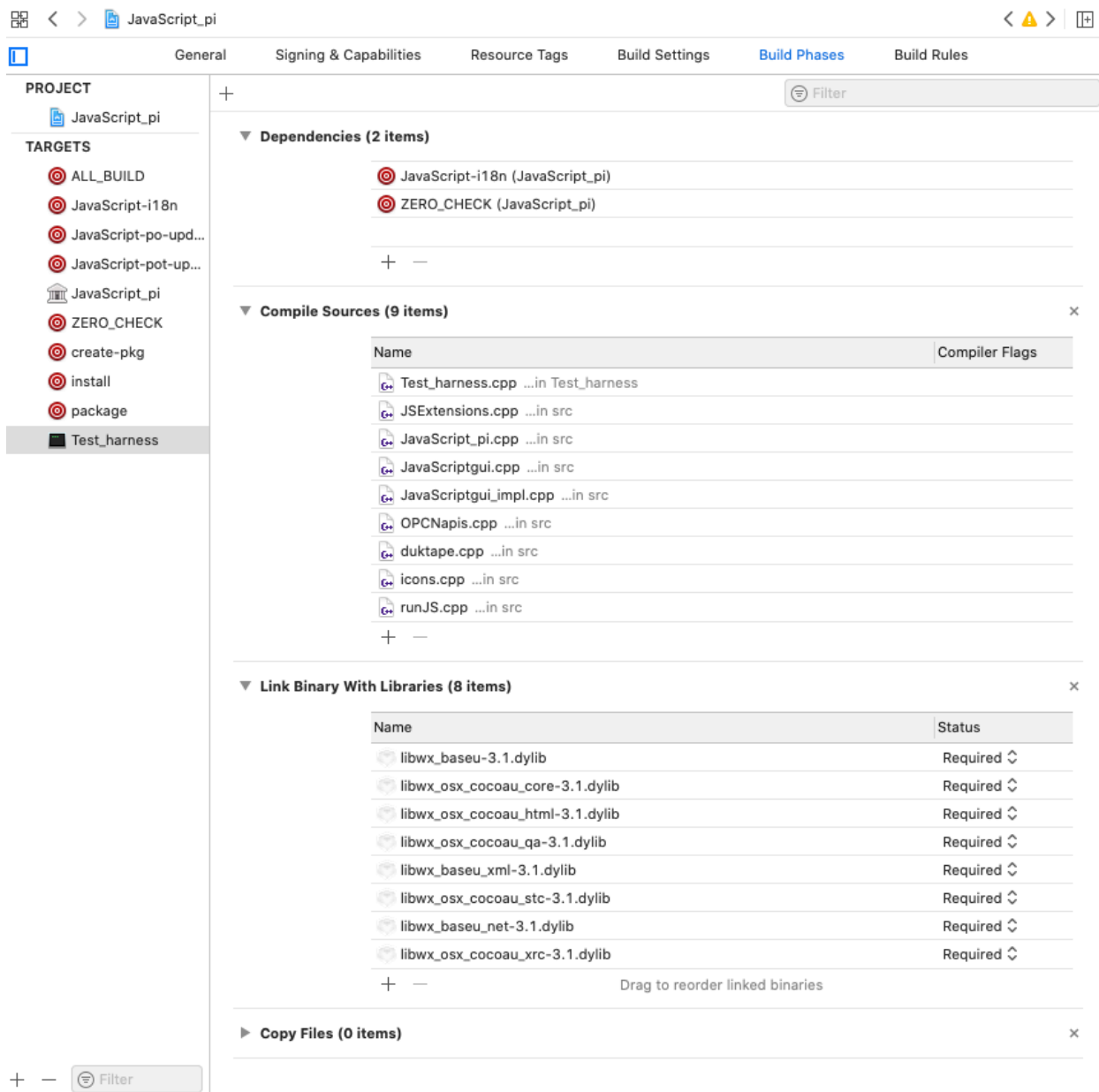
You can now build the test harness and the console window should open. While running the test harness, the full riches of Xcode are available to insert break points and inspect variables, etc.

When the require function is given a simple module name, it tried to load it from 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.

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 context 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`.

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. To allow exploration of what is happening on the stack, I created a conditional macro `MAYBE_DUK_DUMP` which can be inserted in the code. This is defined in `opcpn_duk.h` together with the environment variable `DUK_DUMP`. If `DUK_DUMP` is true, the macro dumps the stack as it is at that point into the output window.

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 request parts.

For the MacOS test harness it is necessary to search the etc 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 moulds (all 148) and compile them together with the plugin.

The following macros need to be defined:

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

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.

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.

Trouble-shooting character code issues

wxWidgets uses Unicode characters, whereas the JavaScript engine is expecting simple characters. Certain characters in scripts need to be fixed up before compiling, i.e. a prime character may might be represented as a forward tick or back tick.

If your script fails with bad character codes, narrow it down to which characters are causing the problem with a simple script as short as possible thus:
`"º'\€"`

Running this script should return a result of the content of the quoted string.

Once you have such a string that fails, use the `Test A` button. Before running the script, this will dump the script in decimal code, attempt a conversion to UTF-8 , dump again and then run it.

Please then submit this with a problem report including details of your computer OS.

Road map for future development

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

The plugin is not yet ready for beta testing but I would be interested to work with others in an alpha phase to develop ideas. I propose we use a Slack workspace I set up to liaise with Mike. If you would like to join in, please contact me by private message.

I anticipate developments will include:

• Addition of further APIs as need identified

• ~~Documentation and a user guide~~ ✓

• ~~Making the scripting window more programmer friendly. At present it knows nothing of tabs, indents and braces. For other than the simplest script, I presently use a JavaScript-aware editor (BBEdit in my case) and paste the scripts into the script window.~~ ✓

• Better resilience. At present there is no protection against a script loop. `while(1);` hangs OpenCPN!

• ~~Implementing the JavaScript `require()` function, which is like a C++ #include to allow loading of pre-defined functions, objects, and methods.~~ ✓

• ~~Running without the console window visible~~ ✓

- ~~Tidier and more consistent error reporting, even when the console is hidden~~ ✓

- 'Canned' scripts that start automatically

- At present, if you want to do separate tasks, you would need to combine them into a single script. I have ideas about running multiple independent scripts.

- I do not use SignalK but note its potential. I am interested in input from SignalK users to keep developments SignalK friendly.

- Other suggestions?

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