JavaScript Plugin Technical Guide

Tony Voss

Document history

Version	Date	
0.1	19 Jul 2020	Initial version to accompany the plugin v0.1

Contents **Page**

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. The Xcode scripts call on the original Cmake files.

Compiling and building for MacOS

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

Building for plugin installation

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

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

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

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

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

As of v0.1 the library of built in scripts are not installed by the installer. This folder scripts must be copied into place within the OpenCPN application at `Contents > SharedSupport > plugins > JavaScript_pi`. It should be at the same level as the icons folder `data`.

The test harness

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

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

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

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

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

Duktape test

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

Note on the Xcode environment

To build the plugin installer, the Xcode project uses the Cmake scripts inherited from the DR_pi. This is rather messy and there are various targets, such as `JavaScript-i18n` which seem to be used and should be left in place.

The test harness was set up I Xcode and is much cleaner in this respect.

The wxWidgets library references may need to be cleaned up to build other than as I have wxWidgets installed.

Building for non-MacOS

Presumably, it will be necessary to rework the original CMake scripts, which are still as for the DR-pi, so that thy compile the correct sources. The list of files to compile can be found by examining the build of the plugin in Xcode. The list of compile sources is to be found in the Build Phase. At the time of writing there are 154 files.

It will be necessary to ensure the relevant environment variables are set up as necessary.

It should be note that the Xcode build of the plugin uses the Cmake files. Care should be taken that regenerating the Cmake files does not disrupt this.

Duktape JavaScript engine

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

Duktape extensions

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

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

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 class and methods used to implement many of the APIs.

It takes some understanding of how to work with the Duktape context stack, especially when constructing objects such as that returned by the `OCPNgetNavigation` function. To allow exploration of what is happening on the stack, I created a conditional macro `MAYBE_DUK_DUMP` which can be inserted in the code. This is defined in `opcpn_duk.h` together with the environment variable `DUK_DUMP`. If `DUK_DUMP` is true, the macro dumps

the stack as it is at that point into the output window. I have only needed to use this from within the test harness.

The script window

The console has been created with wxFormBuilder as usual.

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

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

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

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

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

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

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

The JavaScript plugin now has a decent scripting window.

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

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

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

The require function

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

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

Road map for future development

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

The plugin is not yet ready for 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?