JavaScript Plugin

Tony Voss

Version 0.4 6 April 2021 - plugin history here

Contents Page

1. Introduction and summary

This document is a user guide and reference manual for the JavaScript plugin for OpenCPN.

The plugin allows you to run JavaScript and to interact with OpenCPN. You can use it to build your own enhancements to standard OpenCPN functionality.

There is a separate technical guide covering the inner workings of the plugin and instructions for building it from sources.

Changes since version 0.3 are highlighted in yellow.

Change in documentation only are highlighted in pale yellow.

The basics

Once the plugin has been enabled, its icon appears in the control strip.

Click on the icon to open the plugin console(s). The console comprises a script pane, an output pane and various buttons. You can write your JavaScript in the script pane and click on the **Run** button. The script will be compiled and executed. Any output is displayed in the output pane. You can adjust the boundary between the two panes by dragging the dot up or down - but you need to release before the change comes into effect.

As a trivial example, enter

```
(4+8)/3
```

and the result 4 is displayed. But you could also enter, say

```
function fibonacci(n) {
      function fib(n) {
            if (n == 0) return 0;
            if (n == 1) return 1;
            return fib(n-1) + fib(n-2);
            }
      var res = [];
      for (i = 0; i < n; i++) res.push(fib(i));
      return(res.join(' '));
      }
print("Fibonacci says: ", fibonacci(20), "\n");
```

Get code

The script displays

```
Fibonacci says: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181
```

This illustrates how functions can be defined and called, including recursively. So we have a super calculator!

This guide includes many JavaScript code examples. You can copy them and paste into the script window to try them or use as a starting point but the formatting does not survive this. For the non-trivial ones, a Get code link is given whereby you can access the code which you can copy. If the code is too long to copy from that displayed, you can use the `Raw` button to view it in a copyable form.

Note that the script pane displays line numbers - useful for understanding any syntax error messages. It also supports indent tabbing, which you should use to indent your script as in the above example. It colours the script to aid understanding as follows:

- Comments
- Strings
- JavaScript key words supported
- Plugin extensions to JavaScript documented in this guide
- JavaScript key words not supported - do not use these words

You can use the `Load` button to load JavaScript from a .js file. The file name string is displayed above the script. The `Save` button saves the script back to the named file and the `Save As` button allows you to save it to a different chosen file.

If a file name string is displayed when OpenCPN is closed down, that script will be reloaded when OpenCPN is re-opened.

You can also paste a script in from somewhere else. You might choose to prepare a non-trivial script in a JavaScript-aware editor. I use BBEdit on my Mac.

While a script is running, the Run button changes to Stop. This is relevant when a script is awaiting a call-back from OpenCPN (see later). Pressing Stop will cancel outstanding call-backs.

JavaScript and the embedded engine

A useful guide/tutorial on JavaScript can be found here. The engine fully supports ECMAScript E5.1 with partial support for E6 and E7.

Note that the embedded engine does not support:

- for-of loops
- classes
- arrow functions

The tutorial also covers JavaScript's use in web pages which is not relevant at this time.

File strings

Some functions files on your computer are accessed not by a dialogue window but by a file string, e.g. `my_projects/JavaScript/usefull_script.js`

A file string can be absolute - it identifies the location of the file explicitly. Sometimes, like the example above, it is relative to the plugin's *current directory.*

You can change the current directory in the Directory tab of the tools window, accessed via the tools button top-right in the console or via the plugin preferences button in the plugin's entry in OpenCPN options.

Multiple consoles

Since plugin version 0.4, you can have multiple consoles, each of which can run separate scripts. You can add extra consoles using the Consoles tab of the tools window.

You can delete a console using its close button. The console script must be cleared first as a precaution against accidental deletion and at least one console must remain.

The later section Working with multiple consoles covers some of the many things you can do using multiple consoles.

2. JavaScript plugin extensions

As the JavaScript engine is intended for embedding, it does not contain any input/output functionality, which is inevitably environment dependent.

I have implemented extensions to provide some output capability.

`print(arg1, arg2…)`

The print function displays a series of arguments in the results pane. Each argument can be of type string, number, boolean, array or object. If an argument is an array, the elements will be listed. If it is an object, it will be displayed as its JSON string. Example:

```
print("Hello world ", 10*10, " times over\n");
```

Displays `Hello world 100 times over.` It is often useful to include the string "\n" as the last character to deliver a newline. If the console has been hidden, it will be shown so that the output can be seen.

`print<colour>(arg1, arg2…)`

Where <colour> is one of `Red`, `Orange`, `Green` or `Blue`.

As for `print` but prints in the specified colour. Example:

```
printGreen("This line will print in green\n");
```

`alert(arg1, arg2…)`

This is similar to `print` but the output is displayed in an alert box. The final newline is less necessary here.

~~While the box is displayed, other screen functions are not available.~~

This function returns immediately, leaving the alert displayed so as not to hold up OpenCPN. The alert window has a button with which the user can dismiss the alert once read.

Because of the immediate return, it is possible to raise a subsequent alert before the previous one has been dismissed. In this case, the subsequent alert text will be added to the existing alert.

If you call alert with a single argument of `false`, any existing alert will be dismissed.

If you call alert with no argument, it just returns the existing status.

The function returns `true` or `false` indicating whether the alert is being displayed, so you can test whether the alert has been dismissed by calling it without any arguments.

Example:

```
alert("This is the first alert");
alert("\nThis text will be added to the first alert");
[… other script steps]
if (alert()) print("The alert has not yet been dismissed\n");
alert(false);    // dismisses alert
```

The script will not complete until any alert has been dismissed, although you can use the `stopScript()` function or the Stop button to force script termination.

The alert box can be dragged where you wish and this repositioning will be remembered for subsequent alerts, including across OpenCPN relaunches.

`printLog(arg1, arg2…)`

Prints to the OpenCPN log file. No final newline is needed. Use sparingly.

`readTextFile(fileNameString)`

Reads the text file `fileNameString` and returns the text as a string. Example:

```
input = readTextFile("/Users/Tony/myfile");
print("File contains: ", input, "\n");
```

If the given fileNameString is not absolute, it will be looked for in the current directory as currently set for the plugin.

`writeTextFile(text, fileNameString, mode)`

Writes the text to the `file fileNameString.`

`If mode = 0, the file must not exist.`

If mode = 1, any existing file will be overwritten.

If mode = 2, the text will be appended to any existing lines in the file.

Example:

```
writeTextFile("/Users/Tony/myfile.txt");
```

If the given fileString is not absolute, it will be looked for or created in the current directory as currently set for the plugin.

`require(moduleName)`

Loads and compiles the given module. See the Modules section.

`timeAlloc(milliseconds)`

If a script takes too long to run, it will time out. This function grants more time and returns the time remaining at the time of the call. See the Execution time limit section.

`consoleHide()` or `consoleHide(name)`

Hides the console.

If a console name is given, that console will be hidden.

If hidden, the console will reappear when any output is added to the output window and on script termination.

NB Prior to plugin version 0.4, this function took a value of true or false. This use is deprecated.

`consoleShow()` or `consoleShow(name)`

Shows the console.

If a console name is given, that console will be shown.

`stopScript()` or `stopScript(string)`

Causes the script to stop. If a string argument is supplied, it becomes the result.

Event handling

Often it is necessary to set up a response to an event. Many functions set up a call-back to a function you supply and their function names include ***on***. The first argument is the name of the function to be called on the event (not in quote marks). These calls set up one call-back only. If you want the function to be called repeatedly, it needs set up the next call within it. When the 'on' function is executed, a check is made that the nominated function exists within your main code. Usually any error will be reported on compilation. However, where a call is made within a called-back function, the error can only be discovered at that time.

`onSeconds(functionName, seconds[, parameter])`

Sets up a call to `functionName` after a time of `seconds` have elapsed. Optionally, you may include a third parameter, which will be used as the argument for the call-back. Example:

```
onSeconds(timesUp, 15, "15 seconds gone");
function timesUp(what){
      print(what, "\n");
      }
```

After 15 seconds, this would display the message `15 seconds gone`.

Unlike other call-backs, you may set up up to 10 timed call-backs to different functions or the same function but each call-back is fulfilled only once. If multiple call-backs are due at the same time, they are fulfilled in the order they were set up.

Calling onSeconds with no parameters onSeconds( ) cancels all timers and their callbacks.

onDialogue(function, dialogue)

Opens a dialogue window as defined in the `dialogue` argument which must be an array of structures each describing an element of the dialogue.

This function returns immediately to avoid holding up OpenCPN while you respond to the dialogue.

When you select one of the action buttons, the specified `function` is called with a modified copy of the dialogue structure as its argument. Example:

```
onDialogue(process, [{type:"field", label:"name"}]);

function process(dialogue){
      print("Name is ", dialogue[0].value, "\n");
      }
```

This script displays a dialogue with a single field labelled `name` together with an `OK` button. When the button is selected, the entered name is printed.

Complex dialogues with multiple components can be constructed and processed. This is described in the separate section Dialogues.

The script will not complete while a dialogue remains open, although you can use the Stop button or `exitScript()` to force script termination.

The dialogue box can be dragged where you wish and this repositioning will be remembered for subsequent dialogues, including across OpenCPN re-launches.

Understanding the result

After a script has completed, the *result* is displayed in blue in the output window after any other output, such as from print statements.

Implicit result

The result is usually the result of the last executed statement, so for

```
3+4;
3 == 4;
```

the result is `false`. The `3+4` is not the last statement. The last statement has a boolean value of `false`.

For

```
print("Hi there!\n");
```

this will display `Hi there!` and the result is undefined as the print function does not return a result.

If there are callbacks, the display of the result will be held over until the last callback has been completed or the script is stopped or an error has been thrown.

Explicit result

Instead of the implicit result, you can make it explicit using the `scriptResult` function:

scriptResult(arg1, arg2…)

The arguments are the same as for `print`. This sets the result to what would be printed and it is displayed as the result later.

If scriptResult( ) is called more than once, the last call overrides previous calls.

The function returns the result that will be displayed, so you can manipulate previous results.

```
scriptResult("My result");
...
scriptResult("Previous result was: ", scriptResult());
```

This would leave a result of `Previous result was: My result`

If the scriptResult is set to the empty string "", the result is suppressed entirely.

3. OpenCPN APIs

I have developed a number of APIs to access the functionality of OpenCPN.

These are all functions with names starting with *OCPN*.

OCPNpushNMEA(sentence)

Sentence is an NMEA sentence. It will be truncated at the first * character, thus dropping any existing checksum. A new checksum is appended and the sentence pushed out over the OpenCPN connections. Example:

```
OCPNpushNMEA("$OCRMB,A,0.000,L,,Yarmouth," +
"5030.530,N,00120.030,W,15.386,82.924,0.000,5030.530,S,00120.030,E,V");
```

OCPNgetMessageNames()

Returns a list of the message names seen since the console was created. The list is one name per line. If a call-back is outstanding for that message, the name of the function is also displayed. Example:

```
print(OCPNgetMessageNames());
```

This is primarily used to determine what messages are being received and their precise names.

OCPNsendMessage(messageName[, message])

Sends an OpenCPN message. messageName is a text string being the name of the message to be sent, as reported by OCPNgetMessageNames. Optionally, you may include a second parameter being the JSON string of the message to be sent. Example:

```
routeGUID = "48cf3bc5-3abb-4f73-8ad2-994e796289eb";
OCPNsendMessage("OCPN_ROUTE_REQUEST",JSON.stringify({"GUID":routeGUID}));
```

Before making this call, you should have set up a call-back function using `OCPNonMessageName`.

OCPNonNMEAsentence(functionName)

Sets up a function to process the next NMEA sentence received by the plugin. The function returns a structure containing `OK` -a boolean value concerning the validity of the checksum - and `value` - the sentence itself. Example:

```
OCPNonNMEAsentence(processNMEA);

function processNMEA(result){
      if (result.OK) print("Sentence received: ", result.value, "\n");
      else print("Got bad NMEA checksum\n");
      }
```

OCPNonMessageName(functionName, messageName)

Sets up a call-back to `functionName` next time a message with the name `messageName` is received. The function is passed the message, which is in JSON format. Example:

```
routeGUID alert "48cf3bc5-3abb-4f73-8ad2-994e796289eb";
OCPNonMessageName(handleRT, "OCPN_ROUTE_RESPONSE");
OCPNsendMessage("OCPN_ROUTE_REQUEST",JSON.stringify({"GUID":routeGUID}));

function handleRT(routeJS){
      route = JSON.parse(routeJS);
      try {print("RouteGUID ", routeGUID, " has the name ",
            route.name, "\n");}
      catch(err){print("No such route\n");}
      };
```

Get code

Notes:

- I have here set up the call-back before sending the request to be sure the call-back is in place when the message arrives.
- If the route GUID does not exist, the print will fail, so I am using JavaScript's `try & catch` to handle this.

OCPNgetNavigation()

This function returns the latest OpenCPN navigation data as a structure as shown:

Attributes	.fixTime		Time of fix in seconds since 1st January 1970
	.position	.latitude	latitude in degrees
		.longitude	longitude in degrees
	.SOG		Speed Over Ground
	.COG		Course Over Ground
	.HDM		Heading Magnetic
	.HDT		Heading True
	.variation		Magnetic variation
	.nSats		Number of satellites

Example use:

```
fix = OCPNgetNavigation();
print("Last fix had ", fix.nSats, "satellites\n");
```

While developing this API, I experimented with making it Signal K friendly and returned a Signal K style structure, which is much more complicated. That version remains available as

OCPNgetNavigationK();

If you want to explore this, you can print the structure.

OCPNgetARPgpx()

This function returns the active route point as a GPX string or an empty string if there is no active route point. You need to parse the GPX string as required. Example:

```
APRgpx = OCPNgetARPgpx();  // get Active Route Point as GPX
if (APRgpx.length > 0){
      waypointPart  = /<name>.*<\/name>/.exec(APRgpx);
      waypointName = waypointPart[0].slice(6, -7);
      print("Active waypoint is ", waypointName, "\n");
      }
else print("No active waypoint\n");
```

Get code

OCPNgetNewGUID()

This function returns a new GUID string as generated by OpenCPN.

OCPNgetWaypointGUIDs()

Returns an array of the waypoint GUIDs. Example:

```
var GUIDs;
GUIDs = OCPNgetWaypointGUIDs();
print("There are ", GUIDs.length,
      " waypoints and number 3 has the GUID ", GUIDs[3], "\n");
// prints in my case There are 236 waypoints and number 3 has the GUID
// 5caa0922-3e7c-432d-b075-afe34fbb19b1
```

OCPNgetPluginConfig()

This function returns a structure detailing the plugin configuration with the following attributes:

.versionMajor	Plugin version
.versionMinor	
.comment	Any comment about the version
.APIMajor	API version being used
.APIMinor	
.inHarness	True if plugin running in the test harness, else false

config = OCPNgetOCPNconfig()

Returns the OCPN configuration as a JSON string.

You could print this to see what information is available.

By default this is the configuration at the time the plugin was activated. If the configuration might have changes, you can get an updated version through the `OCPNsendMessage` and `OCPNonMessageName` mechanism.

OCPNrefreshCanvas()

Refreshes the canvas window. If your script has made changes to displayed information such as waypoints or routes, this will update the display accordingly.

OCPNgetAISTargets()

Returns an array of the AIS objects each with the following attributes:

.GUID	
.markName	The waypoint mark name
.position	.latitude .longitude
.iconName	
.isVisible	True if waypoint is displayed
.description	
.hyperlinkList	Array of hyperlinks, each containing .description text to be linked .link .type
.creationDataTime	Creation date/time in seconds since 1st January 1970

.MMSI	The target's MMSI number
.shipName	Ship name (if received)
.class	0 if Class A; 1 if Class B
.callSign	Radio callsign
.IMO	Ship identification number
.shipType	Number representing the ship type, including: 19 pleasure vessel 34 vessel diving 36 sailing vessel 37 pleasure craft 40 high speed craft 50 pilot vessel 52 tug 70 cargo ship Fuller list here.
.navStatus	Number representing the navigational status. The following values are believed to have the meaning ascribed: 0 underway 1 at anchor 5 moored 14 AIS SART 15 AIS SART test Fuller list here.
.position.latitude .position.longitude	Position
.range	Range in nm

.bearing	Bearing °T
.CPAvalid	if true, CPA details valid
.CPAminutes	Time to CPA in minutes
.CPAnm	Nautical miles distance at CPA
.alarmState	0 no alarm 1 alarm set 2 alarm acknowledged

APIs for positions

A position is a latitude and longitude pair. Th difference between two positions is a vector comprising a bearing and distance pair.

OCPNgetVectorPP(fromPosition, toPosition)

Returns the vector to move from the first position to the second position. Example:

```
move = OCPNgetVector({latitude:61,longitude:2},
                     {latitude:60,longitude:2});
print(move, "\n");
// prints {"bearing":180,"distance":60}
```

OCPNgetPositionPV(fromPosition, vector)

Given a position and a vector, returns the position after applying the vector. Example:

```
start = {latitude:55, longitude:-1};
vector = {bearing:180, distance:60};
end = OCPNgetPositionPV(start, vector);
print("end position ", end, "\n");
// prints end position {"latitude":54.001, "longitude":-1}
```

Note: any methods in the start position are not inherited in the returned position - only the latitude and longitude are returned. See Objects and Methods for instructions on how to create a new position with methods.

OCPNgetGCdistance(Pos1, pos2)

Returns the great circle distance between two positions.

APIs for waypoints

Later in this guide in Objects and methods I describe the **position**, **waypoint** and **route** objects, which are the most convenient way of handling these concepts in JavaScript. These objects come with methods to perform actions on them and I recommend that approach.

This section documents the underlying APIs used to implement them. You can call these APIs directly, if you wish, but the returned objects will not include any methods.

A waypoint object has he following attributes:

OCPNgetSingleWaypoint(GUID)

Returns a waypoint object for the given GUID. Throws an error if the GUID does not exist.

Example:

```
GUID = "137eecdd-e3e0-4eea-9d72-6cec0e500dbe";
myWaypoint = OCPNgetSingleWaypoint(GUID);
print("Waypoint name is ", myWaypoint.markName, "\n");
```

OCPNdeleteSingleWaypoint(GUID)

Deletes a single waypoint, given the GUID.

Example:

```
OCPNdeleteSingleWaypoint("6aaded39-8163-43ff-9b6d-13ad729c7bb1");
```

Throws an error if there is no existing waypoint with the given GUID.GUID = OCPNaddSingleWaypoint(waypoint)

Adds a single waypoint into OpenCPN. The argument must be a waypoint object, such as created by the Waypoint constructor.

If `waypoint.GUID` contains a GUID, that will be used. If that GUID already exists, an error is thrown. If no GUID is provided, a new GUID will be obtained for you. This function returned `the` GUID used. If you do not supply a GUID, you need to save the returned one if needed.

Example:

```
Waypoint = require("Waypoint");
newWaypoint = new Waypoint(50.33, -1.3);
newWaypoint.markName = "Demo Waypoint";
newWaypoint.iconName = "anchor";
newWaypoint.isVisible = true;
newWaypoint.description = "Good pub close by ashore";
newWaypoint.hyperlinkList.push({description:"Pub website", link:
      "https://coachandhorses.co.uk"});
GUID = OCPNaddSingleWaypoint(newWaypoint);
newWaypoint.GUID = GUID;
```

Get code

`GUID = OCPNaddSingleWaypoint(waypoint)`

Adds a new waypoint into OpenCPN. The argument must be a waypoint. If the waypoint has no GUID, a new one will be allocated. If a GUID is provide, no existing waypoint with that GUID is allowed. In any case, the GUID used is returned.

`OCPNupdateSingleWaypoint(waypoint)`

Updates a single waypoint into OpenCPN. The argument must be a waypoint object with an exiting GUID.

Throws an error if there is no existing waypoint with the given GUID.

APIs for routes

A route object has the following attributes:

.GUID	
.name	Route name
.to	To text
.waypoints	Array of waypoints in route

OCPNgetRoute(GUID)

Returns a route object for the given GUID, complete with an array of the waypoints. Throws an error if route does not exist.

Example:

```
myRoute = OCPNgetroute("137eecdd-e3e0-4eea-9d72-6cec0e500abc");
print("Route name is ", myRoute.name, "\n");
```

OCPNdeleteRoute(GUID)

Deletes a route, given the GUID.

Example:

```
OCPNdeleteRoute("6aaded39-8163-43ff-9b6d-13ad729c7abc");
```

Thows an error if the route does not exist.

GUID = OCPNaddRoute(route)

Adds a route into OpenCPN. The argument must be a route object, such as created by the Route constructer and should contain an array of waypoints.

If `route.GUID` contains a GUID, that will be used. Otherwise a new GUID will be obtained for you. This function returned `the` GUID used. If you do not supply a GUID, you need to save the returned one if needed. If you supply a GUID and it is already in use, an error is thrown.

Example:

```
Route = require("Route");
myRoute = new Route;
myRoute.name = "My created route";
waypoint1 = new Waypoint(50.33, -1.3);      // create some waypoints
waypoint1.markName = "First Waypoint";
waypoint1.iconName = "diamond";
waypoint2 = new Waypoint(51, -2);
waypoint2.markName = "Second Waypoint";
waypoint2.iconName = "diamond";
myRoute.waypoints.push(waypoint1);          // add waypoints into route
myRoute.waypoints.push(waypoint2);
GUID = OCPNaddRoute(myRoute);               // add route into OpenCPN
myRoute.GUID = GUID;
```

OCPNupdateRoute(route)

Updates a route into OpenCPN. The argument must be a route object with an exiting GUID.

Throws an error if there is no existing route with the included GUID.

4. Error handling

Many of the extensions 'throw' an error when an error situation is encountered. The script will be terminated with an error message. This makes for simple scripting - you do not need to test for errors in these cases.

If you wish to handle the error yourself and continue with the script, you can catch it using the try/catch JavaScript construct:

```
try   {
      OCPNdeleteWaypoint("non-existent GUID");
      }
catch(error) {
      print("Caught error ", error.message, "\n");
      // corrective action here
      }
```

Catch is passed the error object of which error.message is the most useful. The attributes are:

.message	The error message.
.fileName	Name of file where error was thrown.
.lineNumber	Line number within file. If an error is thrown from your script, this will show the line number. However, errors are often thrown from within the plugin and the fileName and lineNumber are for the plugin code rather than your script.
.stack	Stack trace back from the throw.

5. Objects and methods

JavaScript supports the use of objects. These can be a convenient way of representing complex data structures. You can create your own objects and these may have associated methods. The `require` function can load an object constructor. By convention, constructors have an initial capital letter as a reminder that they are constructors.

`Position(lat, lon)` or

`Position({latitude:lat, longitude:lon})`

This constructs a `position` object as follows:

To load constructor	require("Position")	Note: the constructor can take optional latitude & longitude values
		myposition = new Position(60, -1.5);
		or it can take a latitude & longitude pair
		myposition = new Position({latitude:60,longitude:-1.5});

Attributes	.latitude	latitude in degrees
	.longitude	longitude in degrees
	.fixTime	time of position fix if recorded, else 0
Properties	.formatted	Is the position formatted for the human eye
	.nmea	Is the position formatted as used in NMEA sentences. You need to add the comma before and after, if required.

Methods	.NMEAdeode(sentence, n)	decodes the NMEA sentence and sets the position to the nth position in the sentence
	.latest()	Sets the position to the latest position available from OpenCPN and .fixTime to the time of that fix. If no fix has been obtained since OpenCPN was started, the time will be zero.

Example 1:

```
Position = require("Position");       // loads the constructor
myPosition = new Position(58.5, -1.5);// constructs a position
myPosition.longitude = 0.5;// change the longitude
print(myPosition.formatted, "\n"); // displays 58° 30.000'N 000° 30.000'E
print(myPosition.NMEA, "\n");       // displays 5830.000,N,0030.000,E
```

Example 2:

The Position constructor can also be given a latitude & longitude pair structure. Extending the code in Example 1, we could write:

```
shiftedPosition = new Position(OCPNgetPositionPV(myPosition,
           {bearing:180,distance:30}));
print(shiftedPosition.formatted, "\n");
// prints 58° 00.071’N 000° 30.000’W
```

Example 3: Decode an NMEA string and print the second position for the human eye:

```
Position = require("Position");
thisPos = new Position;
sentence =
"$OCRMB,A,0.000,L,,UK-
S:Y,5030.530,N,00121.020,W,0021.506,82.924,0.000,5030.530,S,00120.030,E,V,A*69";
thisPos.NMEAdecode(sentence,2);
print(thisPos.formatted, "\n"); // displays 50° 30.530’N 001° 20.030’W
```

Waypoint()	constructs empty waypoint with its methods
Waypoint(lat,lon)	constructs waypoint for the given latitude and longitude
Waypoint(position)	constructs waypoint for the given position
Waypoint(waypoint)	constructs copy of the given waypoint

The constructed `waypoint` object is as follows:

To load constructor	require("Waypoint")	
Attributes	.postion	As described for Position
	.GUID	
	.markName	
	.iconName	
	.isVisible	`true` if the mark is visibly displayed, else `false`
	.creationDateTime	A timestamp from when the waypoint was first created in OpenCPN recorded as seconds since 1st January 1970.
	.description	Free text description
	.hyperlinkList	Array of hyperlinks (see Hyperlinks)

Methods	.add(GUID)	Adds the waypoint into OpenCPN using the optional GUID, which must not already exist. If GUID is omitted, a new GUID will be obtained. Returns the GUID if successful, else an error is thrown. You must save the GUID if needed. If .creationDateTime is undefined, it is set to the present time. OpenCPN bug: as of v5.2, the .creationDateTime attribute is ignored when adding a waypoint.
	.get(GUID)	Gets the waypoint from OpenCPN and sets the object to it. If GUID is supplied, that is the waypoint loaded. If GUID is omitted, the GUID in `waypoint.GUID` is used. Returns the GUID if successful, else an error is thrown. You must save the GUID if there is any doubt which one was used.
	.update()	Updates the waypoint in OpenCPN to match the contents of this object. The GUID in `waypoint.GUID` must already exist else an error is thrown.
	.delete(GUID)	Deletes the waypoint in OpenCPN with GUID. If GUID is omitted, uses the GUID in `waypoint.GUID.` `An error will be thrown if a` waypoint with the GUID does not exist.
	.summary( )	Returns a brief readable summary of the waypoint markName and position.

Hint

A waypoint returned from OpenCPN is 'bare' - just containing the attributes and no methods. To add the methods to a bare waypoint, construct a copy using, say

```
 bareWaypoint = OCPNgetWaypoint(GUID);
 fullWaypoint = new Waypoint(bareWaypoint);
 // now you can use...
 print(fullWaypoint.summary(), "\n");
```

About hyperlinks

Waypoints and routes can have a description attribute. They can also have one or more hyperlinks - attributes which load a web link or a local file. A hyperlink is itself an object thus:

In a waypoint object, the hyperlinks exist as an array of objects in the .hyperlinks attribute. Herewith an example of adding hyperlinks to a a waypoint:

```
myWaypoint = newWaypoint;
var link1 = {description:"OpenCPN", link: "https://opencpn.org"};
var link2 = {description:"OpenCPN team", link:
      "https://opencpn.org/OpenCPN/info/team.html"};
// push the hyperlinks onto the array
myWaypoint.hyperlinkList.push(link1);
myWaypoint.hyperlinkList.push(link2);
```

`Route()`	constructs a route object with its methods
`Route(route)`	constructs a copy of the given route adding methods

The constructed `route` object is as follows:

To load constructor	require("Route")	
Attributes	.name	The route name
	.GUID	
	.from	The from text
	.to	The to text
	.waypoints	An array of the waypoints in the route, each being a waypoint object.
Methods	.add(GUID)	Adds the route into OpenCPN using the optional GUID, which must not already exist. If GUID is omitted, a new GUID will be obtained. Returns the GUID which you may need to save. An error is thrown if the GUID is already in use.
	.get(GUID)	Gets the route from OpenCPN and sets the object to it. If GUID is supplied, that is the waypoint loaded. If GUID is omitted, the GUID in `waypoint.GUID` is used. Returns the GUID if successful, else an error is thrown. You must save the GUID if there is any doubt which one was used. `route.waypoints` will be an array of the route's waypoints.
	.update()	Updates the route in OpenCPN to match the contents of this object. The GUID in `route.GUID` must already exist otherwise an error is thrown.
	.delete(GUID)	Deletes the route in OpenCPN with GUID. If GUID is omitted, uses the GUID in `route.` If a route with the GUID does not exist, an error is thrown.
	.purgeWaypoints()	Deletes all waypoints within the route object, including the waypoint's hyperlinks.

Hint

A route returned from OpenCPN is 'bare' - just containing the attributes and no methods. To add the methods to a bare route, construct a copy using, say

```
 bareRoute = OCPNgetRoute(GUID);
fullRoute = new Route(bareRoute);
// now you can use, for example, ...
fullRoute.update();
```

About JavaScript objects and OpenCPN objects

It is important to understand the difference between objects in OpenCPN and the objects in a JavaScript representing them. Consider the following:

JavaScript	What changes in JavaScript	What changes in OpenCPN
`myRoute = new Route()`	New JavaScript route object created	Nothing
`myRoute.add()`	Nothing	Route is added
`myRoute.purgeWaypoints()`	Waypoints are purged from the JavaScript object	Nothing
`myRoute.delete()`	Nothing	OpenCPN route is deleted
`delete myRoute`	JavaScript object is deleted	Nothing

6. Modules

Above, you learnt how to load a constructor from the built-in library.

You can also load code from your own file space using the `require` function. If the require argument is a simple name without a suffix or file path separator, `require` looks for a built-in component. Otherwise it uses the require parameter to look for a file. If the parameter is a relative file string, it looks relative to the current directory set for the plugin. If it is an absolute file path, then it loads that file.

Loading your own functions

As an example of how to write your own functions to load with `require`, consider the fibonacci function shown above.

You could save this into a file, load it with a require statement and call it, e.g.:

```
require("myJavaScipts/fibby.js");
print("Fibonacci said: ", fibonacci(10), "\n");
```

Writing and loading your own object constructors

Constructors work similarly to functions but construct an object . Here is a trivial constructor for an object which includes a method

```
function Boat(_name, _make, _model, _length){
      this.name = _name;
      this.make = _make
      this.model = _model
      this.length = _length;
      this.summary = function(){return(this.name + " is an " +
            this.make + " " + this.model + " of length " +
            this.length +"m\n");}
      }
```

Note how the attributes are set when the constructor is called. As this is a constructor, an object must be created from it, once loaded. Example:

```
Boat = require("myjavascripts/Boat.js");
myBoat = new Boat("Antipole", "Ovni", 395, 12);
myBoat.length = 12.2; // correction
print(myBoat.summary(), "\n");
// prints Antipole is an Ovni 395 of length 12.2m
```

7. Working with Date Time

OpenCPN counts the time in seconds since 1st January 1970.

The JavaScript Date object uses milliseconds since the same epoch, so it is necessary to convert as required. The following script illustrates this:

```
Position = require("Position");
latestPos = new Position();
latestPos.latest();              // sets to latest position
presentTime = new Date()/1000;   // convert to seconds
print("Latest position ", latestPos.formatted);
print("was acquired ", (presentTime - latestPos.time).toFixed(1),
      "s ago");
print(" at ", Date(latestPos.time*1000), "\n");  // time to msecs
```

When I tested the above, it displayed:

Latest position 50° 41.054’N 002° 5.307’W was acquired 2.7s ago at 2020-08-12 09:47:34.762+01:00

8. Execution time limit

Your script could get into a continuous loop and never end. This might be because of a simple scripting error or because some condition for ending the script was not being met. As simple example, the following script never ends because true is always true:

```
while(true) ;
```

This would lead to OpenCPN being locked up with the only way out to force-quit OpenCPN - not something you want to happen during navigation!

To protect against this, the plugin places a time limit on script execution and will terminate it if the limit is exceeded. By default this is set at 1000ms. Each callback gets its own 1000ms limit.

The `timeAlloc` function extends the time limit and returns the number of milliseconds remaining at the time of the call before it is extended. Optionally you may provide a new time allocation in place of the default. This new allocation will be used at the call and all subsequent calls not specifying a different one. Subsequent call-backs will be given this allocation.

For a long script, you might use timeAlloc to grant extra time once you have reached a point where time might be exhausted.

Beware of using `timeAlloc` in a loop. If the script gets stuck in the loop, it might repeatedly allocate more time thus defeating the timeout mechanism.

```
[ long script steps reach a point where further time will be needed]
print("At this point ", timeAlloc(2000), "ms remain\n");
[ more script steps for which 2000ms have been granted ]
```

There is a detailed time-out tester available.

Get code

9. Dialogues

The onDialogue API provides a way of creating and completing dialogues in a way that does not prevent other functioning of OpenCPN. It is possible to build quite complex dialogues with multiple buttons and this is described in this section.

The basic call is:

`onDialogue(function, dialogue)`

where `function` is the function to be called when a button is selected and `dialogue` is a descriptor of the dialogue to be presented. This function returns immediately so that functioning of OpenCPN is not suspended while the user responds to the dialogue.

`dialogue` is an array of one or more structures each describing an element of the dialogue to be displayed. Each element of the dialogue array must include its type attribute. Which other attributes are applicable depends on the type.

The specified function is given a copy of the dialogue array, in which certain elements will be changed to reflect the action taken with the dialogue, as described in the table in purple. An additional element will have been added identifying the button used to dismiss the dialogue.

type	Purpose	Other attributes Grey items are optional	Explanation
"caption"	Specify caption in dialogue bar	value:"caption"	If value is omitted, the caption will be blank. If no caption element is provided, the caption defaults to "JavaScript dialogue".
"text"	Places text in the dialogue	value:"text"	The text from the value attribute in placed in the dialogue. Multiple text elements can be used to place information as required.
"field"	Provide an input field.	label:"text"	Text to form label for field.
		value:"text"	This attribute will always be included in the returned structure and will be set to the value of the field on completion of the dialogue. If this attribute is included in the call, it will be displayed in the field as place holder text which can be edited/ replaced while the dialogue is open.
		width:number	Width of field. Default is width:100.
		height:number	Height of field. Default is 22 or whatever is needed for larger text set by style.
		multiLine:boolean	If true, the field will be be multi-line.
		sufix:"text"	Suffix text to be displayed after the field, e.g. "°T"
		fieldStyle	See below on styling

type	Purpose	Other attributes Grey items are optional	Explanation
"tick"	Provide a tick box	value:"text"	The text against the tick box. If the value starts with "*" that character will not be displayed but the box will be pre-ticked. In the returned structure value will be true or false.
"tickList"	Provide a list of items to tick	value:["A", "B"…]	In the returned structure, value is an array of the ticked items only. If none, it will be an empty array.
"choice"	Choose one from a list of items.	value:["A", "B"…]	The first item is the default value. In the returned structure, value is the selected value.
"radio"	Provide a set of radio buttons, of which just one can be selected. No more than 50 buttons will be displayed.	label:"text"	Text to form label for the buttons. Omit to suppress label.
		value:["one","two"…]	Array of texts specifying the button choices. In the returned structure, this attribute will be set to the single button selected on completion of the dialogue (not in an array).
"slider"	Provides a horizontal slider allowing selection of an integer value	range:[start, end]	Numeric values for the start and end of the slider range
		value:number	Initial value of slider
		width:number	Width of slider (not a string). Default is width:200.
		label:"text"	Text to form label for the slider. Omit to suppress label.
"spinner"	Provides a numerical field that can be spun up or down	range:[start, end]	Numeric values for the start and end of the spinner range
		value:number	Initial value of spinner. Defaults to zero.
		label:"text"	Text to form label for the spinner. Omit to suppress label.
"hLine"	Horizontal line	None	Adds a horizontal line as a separator.

type	Purpose	Other attributes Grey items are optional	Explanation
"button"	Add one or more action buttons	label:"button" or label:["one","two"...]	The label for the button. If more than one, these are specified in an array. If the button starts with '*', it will become the default button, which can be acted on using the enter key. The '*' is not displayed. Example: "*Done" In the returned structure, this attribute will be set to the single button selected on completion of the dialogue (not in an array) and without any *. If there is no element of type button, a default button "OK" will be added by the plugin. No corresponding element will be added to the returned structure as OK will be the only action choice.

Simple example:

```
myDialogue = [
     {type:"text", value:"Complete this field"},
     {type:"field"},
     {type:"button", label:["Cancel", "*OK"]}
     ];
onDialogue(action, myDialogue);

function action(dialogue){
     if (dialogue[dialogue.length-1].label == "OK")
          print("Completed field is: ", dialogue[1].value, "\n");
     else print("Cancelled\n");
     }
```

Styling

You may want to adjust the style of text in a dialogue. You can include the `style` attribute with any of the above but it will not have any effect on some dialogue components.

For the field type the style operates on the label, field and any suffix. You can override the style for the field itself with fieldStyle

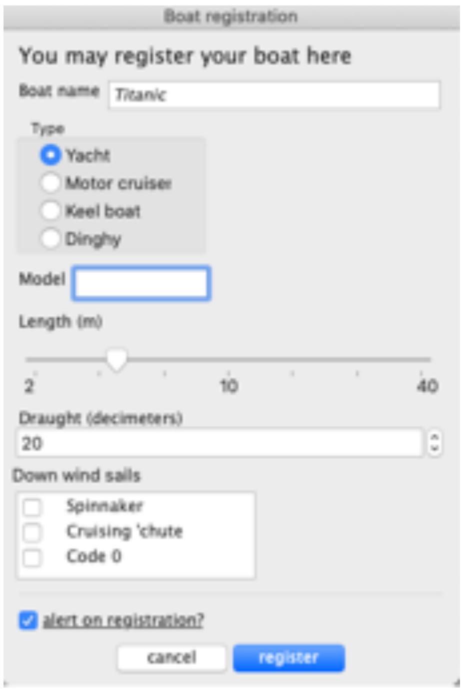
Styling is not included in the returned version of the dialogue array.

style:{style attributes}		Available with fieldStyle?
size:<number>	Font size e.g. size:20	
font:<string>	Font name e.g. font:"courier" If the font name does not match one in your system, it may prevent other style components from working.	✓
italic:<bool>	e.g. italic:true	✓
bold:<bool>	e.g. bold:true	✓
underline:<bool>	e.g. underline:true	

Example with styling

Here is an example showing various types and including some styling.

Get code.



In the demonstration scripts, there is practical application which builds race routes through a series of dialogues.

10. Automatically running scripts

It is possible to arrange for a script to be automatically loaded and run when OpenCPN starts up, without the need to load the script and run it manually.

Your script needs to be stored in a .js file. Test it before attempting to run it automatically.

When a script has been loaded from a file or saved to one, the auto run button will be shown at the top of the console. If this is ticked before OpenCPN closes normally, then when the plugin is activated, that script will be loaded and run automatically.

If the script hides the console, the console will not be seen until it is unhidden or the script produces output in the output pane or the script terminates.

If a script is running while hidden and you need to stop it, you can make the console appear by toggling the tool bar icon. You could then stop the script if required.

To stop a script running automatically, untick the option before quitting OpenCPN.

In the unlikely event that a script were to crash OpenCPN and that script were being run automatically, OpenCPN might crash immediately on launch before you could stop it running. To get out of this situation, open the `opencpn.ini` file and in the JavaScript section change `AutoRun=1` to `AutoRun=0`. This will stop the script running automatically on launch.

11. Working with multiple scripts

You can link scripts in a chain to be run successively. This can be used to break up long scripts into successive 'chapters'. A script can pass a brief to its successor.

`chainScript(fileString [, brief]);`

Loads the script in the file `fileString` into the script window, gives it a brief if supplied and runs it.

The successor script can collect its brief with

`brief = getBrief();`

Example:

Let a file successor.js contain the script

```
print("Found brief ", getBrief(), "\n");
```

And run the script

```
chainScript("successor.js", "Brief text", true);
```

This last will load and run successor.js, which will print

```
Found brief Brief text
```

Although the brief is limited to a text string, an array or structure could be passed as a JSON string.

12. Working with multiple consoles

You can have more than one console. Each console has its own script, which runs independently, apart from interactions detailed later.

To create an additional console, use the Consoles tab in JavaScript tools to give it an alphanumeric name and create it. You can also access the tools through the Preferences button in the plugin entry in the list of plugins in the OpenCPN Options panel.

To delete a console, use its close button. As a precaution against accidental loss of a script, the script window must be cleared before being closed. You cannot delete the last and only console.

Communicating between scripts

The OpenCPN messaging system can be used to send messages between scripts. The receiving script must be waiting for the message when it is sent.

Working with multiple consoles in scripts

It is possible to create and use multiple consoles from within a script.

To use the facilities described in this section, you need to load the optional Console extensions with
`require("Consoles");`

`consoleAdd(consoleName)`

Adds the console specified, the same as adding via the tools.

An error will be thrown if the console already exists.

`consoleExists(consoleName)`

Returns true if the console exists, else false.

`consoleClose(consoleName)`

Closes the console.

An error will be thrown if the console is busy. You cannot close the console running this script step.

`consoleGetOutput(consoleName)`

Returns the contents of the output pane of the console.

`consoleClearOutput(consoleName)`

Clears the contents of the output pane of the console.

`consoleLoad(consoleName, scriptFile)`

Loads the script into the script window.

`consoleRun(consoleName [,brief])`

Runs the script in the console, optionally giving a brief.

`consoleBusy(consoleName)`

Returns true of the console is busy running a script or waiting for callbacks, else false;

`onConsoleResult(consoleName, function [, brief])`

Runs the script in the console and sets up a call-back to the specified function on completion. The other script is given the brief, if supplied.

On completion, the function is invoked and given an argument being the outcome from the other as a structure with attributes:

.type	The type of outcome as an integer 0 other script threw an error 1 other script completed normally 2 other script executed a scriptStop( ) step
.value	If an error, the error reason. Otherwise, the script result

Example

```
require("Consoles");

name = "TestConsole";
if (!consoleExists(name)) consoleAdd(name);
consoleLoad(name, "myJavaScript.js");
onConsoleResult(name, allDone, "Go well");

function allDone(result){
     if (result.type == 1)
          throw("myJavaScript threw error " + result.value);
     print("Result from myJavaScript was ", result.value, "\n");
     }
```

This script creates the console if it does not already exist, loads it with a script and runs it giving it a brief. On completion, the callback to function allDone checks for an error and throws an error in itself and otherwise prints the result.

13. Tidying up

Sometimes you may need to tidy up after a script terminates, a console is closed or is terminated because OpenCPN has quit. As an example, the Tack Advisor script creates a temporary two-point route to suggest where to tack. If OpenCPN were to quit with Tack Advisor displaying this route, the route would still exist when OpenCPN is next run although it would then be meaningless.

`onExit(functionName)`

This call specifies a function to be called after the script has completed, including when a console is closed or when OpenCPN quits. This can be used to clean up. In the above example, any route created to advise where to tack is deleted.

The function is called at the end of the wrapping up process and so some actions within such a function are meaningless. For example, if call-backs are set up they will have no effect. If the function throws an error, it will be displayed in the output window but if the window is being closed, it would vanish along with the console. It would be prudent to test the function in a situation where the console remains visible.

14. Trouble-shooting character code issues

The JavaScript engine uses the ECMA-6 7-bit character set, corresponding to the ASCII set. In broad terms, this excludes the extended characters available by use of the Option key and accented characters.

If you prepare or edit your script in an external program, it may introduce characters not compatible with the JavaScript engine. Examples

- smart quotes around "Hello" like this: “Hello”
- Smart single quotes around 'goodbye' like this: ‘goodbye’
- The apostrophe can be useful as itself or as an alternative string delimiter, as in
  `'This string includes a quote character "'`
  The apostrophe ' might get entered as any of ‘ ’ ‛ ′ ´ `
- wxWidgets uses Unicode characters and copying text from OpenCPN could introduce characters which would throw the JavaScript engine.

The plugin tried to fix up unacceptable characters in scripts before compiling. If your script fails with the engine tripping over bad characters, 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 contents of the quoted string.

In the diagnostics tab of the utilities window is a facility to examine characters and their translation. Please submit the dumped code analysis with a problem report.

Under Windows, the plugin is unable to convert the prime character ′ and it will likely cause a JavaScript error.

Working with non-7-bit characters such as the degree symbol

If you use characters not included in the 7-bit set, it may or may not work and you may have compatibility issues across different platforms. It is safest to generate these characters within a script using the String.fromCharCode( ) function that return the required character.

A relevant case is the degree symbol ° which has the decimal code 176 and is not in the 7-bit set. If you display a bearing with, say,

print("Bearing is " , bearing, "°T\n");

this works under MacOS but not under Windows.

Instead you could use

```
print("Bearing is ", bearing, String.fromCharCode(176), "T\n");
```

15. Demonstration Scripts

In this section, you will find a number of scripts that demonstrate aspects of in the plug-in. They are chosen for their ability to demonstrate the capabilities of the plugin and perhaps act as starters for creating your own applications. You can copy the scripts and paste them into the script window. In many cases they do things that can be done in OpenCPN itself but aim to show how these things can be done programatically.

A. Process and edit NMEA sentences

This script addresses an issue someone had whereby their RMC sentences did not include magnetic variation, which was available in their HDG sentences. This script captures variation from the HDG sentences and inserts it into any RMC sentences that do not already have the variation.

(Hint to help you understand this: the .split method splits a string at each of the specified character into an array, here called splut. .join does the reverse.)

```
// insert magnetic variation into RMC sentence
var vardegs = "";
var varEW = "";

OCPNonNMEAsentence(processNMEA);

function processNMEA(input){
     if (input.OK){
          sentence = input.value;
          if (sentence.slice(3,6) == "HDG")
               {
               splut = sentence.split(",");
               vardegs = splut[4];   varEW = splut[5];
               }
          else if (sentence.slice(3,6) == "RMC")
               {
               splut = sentence.split(",");
               if ((splut[10] == "") && (vardegs != ""))
                    { // only if no existing variation and
                      // we have var to insert
                    splut[10] = vardegs; splut[11] = varEW;
                    splut[0] = "$JSRMC";
                    result = splut.join(",");
                    OCPNpushNMEA(result);
                    }
               }
          }
     OCPNonNMEAsentence(processNMEA);
     };
```

Getcode

B. Counting NMEA sentences over time

This script NMEA-counter.js counts down for 30 seconds and then lists the OpenCPN messages and NMEA sentences it has seen. The NMEA sentences are sorted by count and then alphabetically.

Get code

C. Locate and edit waypoint, inserting hyperlinks

This script locates a waypoint called "lunch stop" and changes its icon name to "Anchor". It nudges the waypoint slightly north, adds a description and adds some hyperlinks referencing the nearby pub.

```
Waypoint = require("Waypoint");  // loads the constructor
allWaypoints = OCPNgetWaypointGUIDs();      // get array of all waypoints
for (i = 0; i < allWaypoints.length; i++){
      // look for our waypoint
      if (allWaypoints[i].markName == "lunch stop"){
            lunchWaypoint = new Waypoint;
            lunchWaypoint.get(allWaypoints[i].GUID);
            break;
            }
      }
if (typeof lunchWaypoint == "undefined") throw("Waypoint not found");
// we have our waypoint - now update it
lunchWaypoint.iconName = "Anchor";
// nudge the position north towards shore
lunchWaypoint.position.latitude += 0.001;
lunchWaypoint.description = "Great anchorage with pub close ashore";
lunchWaypoint.hyperlinkList.push({description:"Pub website",
      link:"https://goldenanchor.co.uk"});
lunchWaypoint.hyperlinkList.push({description:"Menu",
      link:"https://goldenanchor.co.uk/menu"});
lunchWaypoint.update();     // update OpenCPN waypoint
```

Get code

D. Build routes from NMEA sentences

This script listens for routes being received over NMEA in the form of WPL and RTE sentences and creates OpenCPN routes from them.

There is an option to match received routes with any existing route of the same name and replace it. In this case a check is made that the existing routes have unique route names.

There is an internal simulator. In simulation mode, the script does not listen for real NMEA sentences but generates simulated ones which are passed to the sentence processor.

As a JavaScript example, this script is interesting because it:

- it has a built-in simulator allowing testing without having incoming NMEA data
- makes full use of the Position, Waypoint and Route constructors
- has to deal with the complication that RTE sentences may be sent in instalments, as necessitated by the 80 character length limit
- It makes good use of JavaScript arrays, including:
  - pushing items onto an array
  - pulling (shift) items off the front
  - joining items into a string

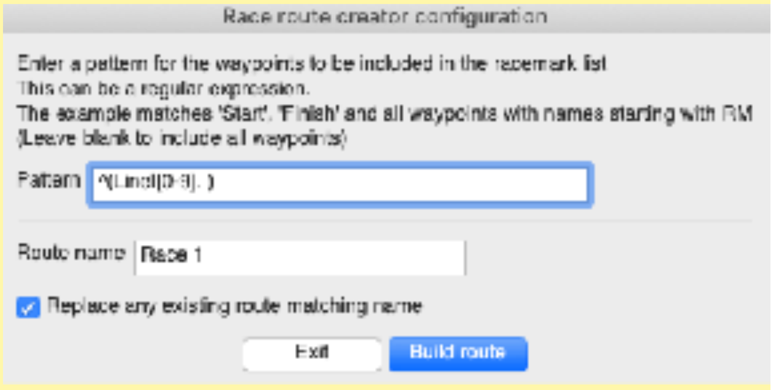
This script was written as a demonstrator for researchers at the Technical University of Denmark.

Get code

E. Build race courses

This script was inspired by bobgarrett's wish to be able to create race course routes from a list of waypoint names rather than hunt for them on the chart.

The script allows the user to specify a regular expression pattern by which to select those waypoints which are race marks.


Race route creator configuration

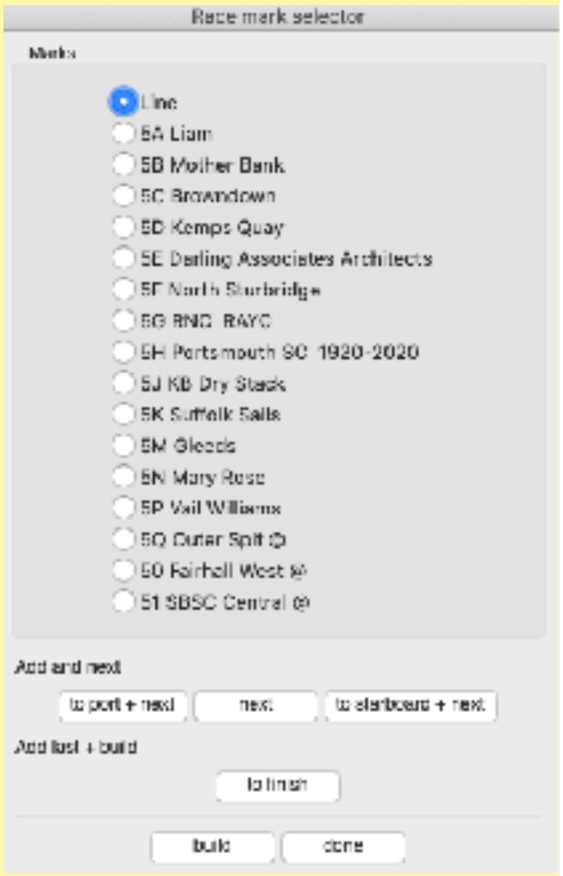
Enter a pattern for the waypoints to be included in the racemark list
This can be a regular expression.
The example matches 'Start', 'Finish' and all waypoints with names starting with RM
(Leave blank to include all waypoints)

Pattern: ^(Line|[0-9].)

Route name: Race 1

[✓] Replace any existing route matching name

In the eastern Solent, the race mark names all start with the digit 5 followed by another character and a space. In this example there is also a waypoint *Line* placed on the start line and we are going to build a route for *Race 1.* When you click on *Build route*, you are presented with the Race mark selector.


Race mark selector

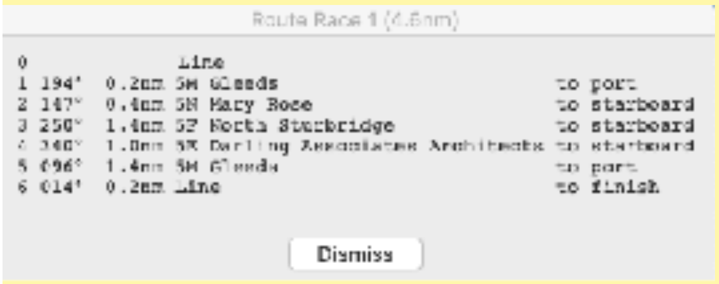
Marks

- [✓ Line]
- 5A Liam
- 5B Mother Bank
- 5C Browndown
- 5D Kemps Quay
- 5E Darling Associates Architects
- 5F North Stoolbridge
- 5G RNC RAYC
- 5H Portsmouth SC 1920-2020
- 5J KB Dry Stack
- 5K Suffolk Sails
- 5M Gleeds
- 5N Mary Rose
- 5P Vail Williams
- 5Q Outer Spit (S)
- 50 Fairhall West (P)
- 51 SBSC Central (P)

In this dialogue you select the course marks in order adding them to the course. You can indicate whether they are to be left to port or starboard.

In this example, the finish is through the start line, so the final selection is *Line* and the button *to finish.*

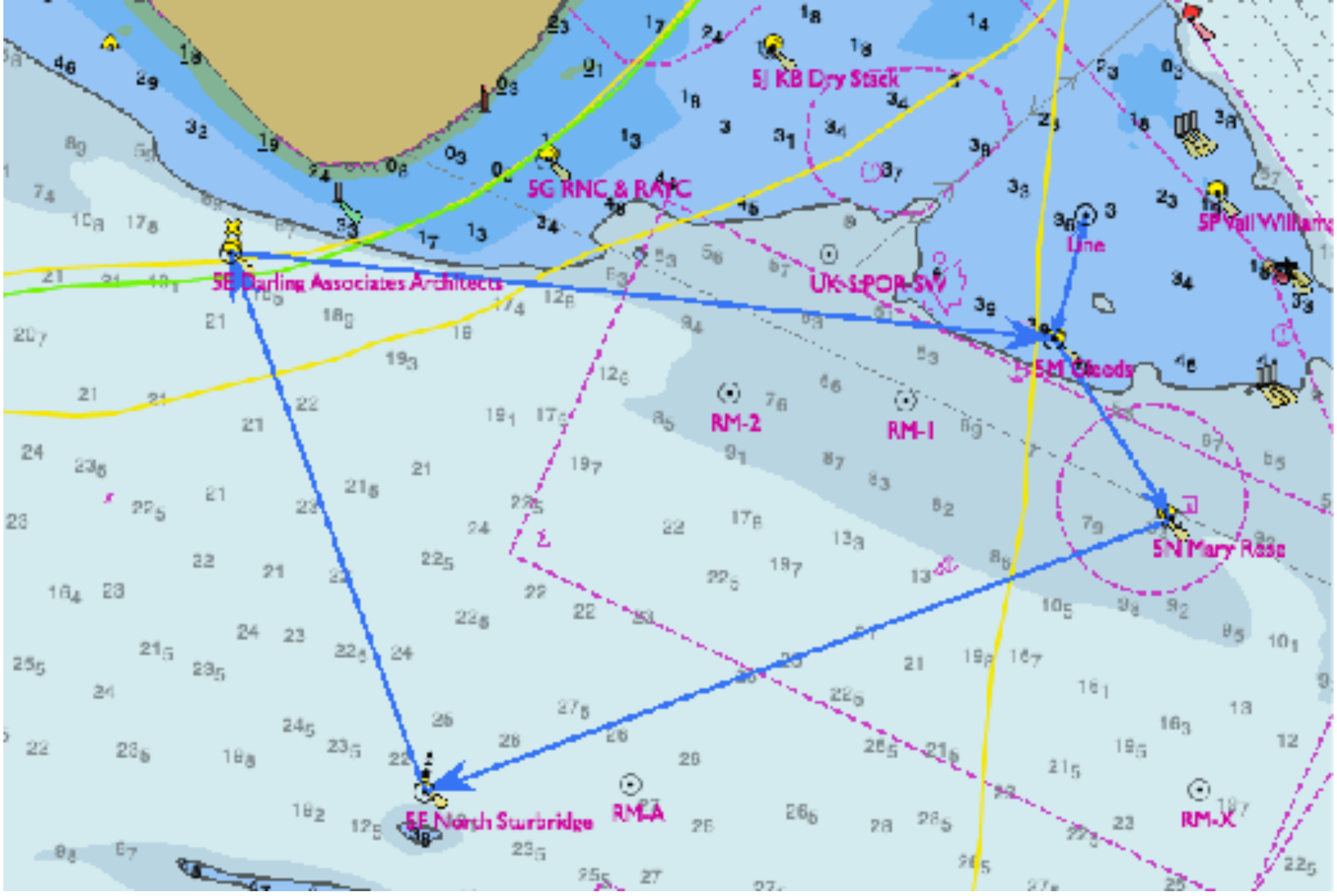
The script then builds the route in OpenCPN and also displays the route with the list of waypoints indicating the bearing and distance to each and which side to pass. The cation includes the course length.


Route Race 1 (4.6nm)

```
0                Line
1 194°  0.2nm 5M Gleeds                      to port
2 147°  0.4nm 5N Mary Rose                   to starboard
3 250°  1.4nm 5F North Starbridge            to starboard
4 340°  1.0nm 5E Darling Associates Architects to starboard
5 096°  1.4nm 5M Gleeds                      to port
6 014°  0.2nm Line                           to finish
```

Get code

This script makes extensive use of the onDialogue function and is a useful example to work from.



F. Driver

This is a simulator that can be used to drive the ship in the absence of actual NMEA inputs. It is an alternative to the ShipDriver plugin but does not use steering to gradually change course. It generates GLL, VTG and WML NMEA sentences. You could add others as required.

You can set Speed Over Ground (SOG), Course Over Ground (COG), Wind angle and wind speed. Selecting Compass course will then drive the boat along the selected course. The angle to the wind is displayed.

You can instead specify an angle to the wind and port or starboard tack. It will then calculate the required COG. Selecting the opposite tack will tack the boat.

Driver can be run in its own console and used, for example, to experiment with or test the `TackAdvisor` and `SendActiveRoute` scripts running in their own consoles.

Get code

G. TackAdvisor

This script monitors for when you have an active waypoint and will need to tack to reach it. It then displays the two tack legs required.

If you are running off the wind to an active waypoint, and will need to gybe to reach it, it displays the two legs and hence the recommended point to gybe.

TackAdvisor does not take cross-current offsets into account and will not give an accurate tack point if the cross-current is significant.

If TackAdvisor is standing by and not displaying your tacks when you are expecting it to, check for the following. It will not display tacks under any of these conditions:

- No active waypoint
- You are off the wind by more than the configured amount, i.e. reaching
- You are heading too close to the wind to be sailing
- You are running close to straight for the waypoint

You can exercise TackAdvisor without being underway by running Driver in a separate console. When you set Driver to a beat or near run, TackAdvisor will display the necessary tacks.

Get code

H. SendActiveRoute

This script monitors for when you have an active route and sends a series of NMEA sentences so that another device such as a chart plotter or a device running iNavX will follow the route itself. Any updates to the route, such as modifying a route point or advancing from one route point to the next will be updated within the receiving device.

When the script detects an active route, it sends the following NMEA sentences:

A. A series of WPL sentences defining the waypoints in the route

B. A group of RTE sentences creating a route comprising the waypoints

C. A BOD sentence with the bearing from the position at which the leg was activated to the next route point.

These sentences cause a device running iNavX to hold a mirror copy of the current route and navigate to the active point within the route.

While a route is active, OpenCPN sends APB sentences with the routeName. Unfortunately, the route name is truncated. This script fixes up the APB sentences to carry the full route name.

Instructions for making this work with a device running iNavX

1. Have OpenCPN receive NMEA data on one port - say 60001
2. Have OpenCPN send NMEA data on a different port - say 60002. Because my Wifi router only handles a single TCP connection, I send using UDP.
3. Connect your iPad/iPhone to the same WiFi network.
4. Within iNavX select Instruments > TCP/IP and set the protocol and port number to as in Step 2 above.
5. On the same panel, Enable Waypoints and enable Link. You should now see the NMEA sentences scrolling in the monitoring pane of this panel.
6. Click Done and then select the Chart.

The device should now follow the ship's navigation using the ship's navigation data.

When a waypoint becomes active in OpenCPN, it becomes active in iNavX.

When a route is activated in OpenCPN, it appears as the active route in iNavX. As OpenCPN advances the routepoint, so iNavX advances its active routepoint. Progress along the route is available in the route tab of the ribbon at the top of the iNavX screen, together with predicted time on route and ETA.

If you wish to force an advance to the next routepoint, this is best done on OpenCPN, whereupon iNavX will update too. Should you advance the routepoint in iNavX, it will start ignoring changes in the active routepoint send from OpenCPN. To restore this, go to the panel used for step 5 above, turn Enable Waypoints off and back on again.

Should OpenCPN fail/crash/hang-up etc., iNavX will continue to navigate the route independently. Should the ship's navigational data over NMEA fail, an iOS device with GPS will use its location service instead and continue to navigate the route.

This has been tested on iNavX running on an iPad and iPhone. It should also work on an Android device running iNavX but I have not tested that.

Get code

A. Plugin version history

Version	Date	
0.1	20 Jul 2020	Initial alpha release for feedback
0.2		• Error reporting regularised • Added various APIs including those to access GUIDs, waypoints & routes • Script window greatly enhanced for writing JavaScript • Output window brought into line with script window • Dealing with spurious characters such as accents improved • User and technical guides developed • Builds for Windows and Linux added • Established on GitHub

Version	Date	
0.3		• The script window now highlights plugin extensions and unsupported keywords by colourising them. • The result is now displayed last after any callbacks have completed rather than at the end of the main script. The scriptResult( ) function can be called to set the result. • Error handling has been improved and makes proper use of the Dukcode error object. • Various APIs now throw an error rather than returning a boolean result, namely ✦ OCPNgetSingleWaypoint( ) ✦ OCPNdeleteSingleWaypoint( ) ✦ OCPNaddSingleWaypoint( ) ✦ OCPNupdateSingleWaypoint( ) ✦ OCPNgetRoute( ) ✦ OCPNdeleteRoute( ) ✦ OCPNaddRoute( ) ✦ OCPNupdateRoute( ) • Print & alert now accept arrays and objects as arguments • Alert no longer holds up OpenCPN • Scripts will now timeout if they take too long, such as if in a loop. • timeAlloc( ) allows management of the time limit. • Extensive support for creating and responding to dialogue windows. • OCPNonSeconds( ) has been renamed to onSeconds( ) • New JavaScript extensions ✦ print<colour>( ) ✦ printLog( ) ✦ timeAlloc( ) ✦ scriptResult( ) ✦ consoleHide( ) ✦ onDialogue( ) ✦ exitScript( ) • New APIs added ✦ OCPNgetPluginConfig( ) ✦ OCPNrefreshCanvas( ) ✦ OCPNgetAISTargets( ) ✦ OCPNgetVectorPP() ✦ OCPNgetPositionPV( ) ✦ OCPNgetGCdistance( )

Version	**Date**	
0.4		• Position.NMEA precision increased from 3 to 5 decimal places • Added writeTextFile • Console Hide & Show now separate calls • Added script auto-start ability • Added chainScript • Added JavaScript tools panel and current directory concept • Added support for multiple consoles • Added support for inter-console calls • Errors thrown from within the plugin APIs now show the line number and trace-back where applicable • Added onExit( ) capability • Bug fix: hidden console was reappearing if OCPNdeleteRoute failed • Extra example scripts

B. 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
0.2.1	3 Sep 2020	Code source links now to to gist itself rather than the raw window. They no longer need to be changed if gist is updated.
0.3	16 Nov 2020	To accompany plugin v0.3
0.3.1	22 Dec 2020	Correction to demo script *Process and edit NMEA sentences*
0.4	20 Apr 2021	To accompany plugin v0.4
0.4.1	27 Sep 2021	Section on character sets expanded to assist with the degree symbol.