JavaScript Plugin

Tony Voss

Version 4.0 April 2026 - plugin history here

Contents

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 the previous version 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..

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

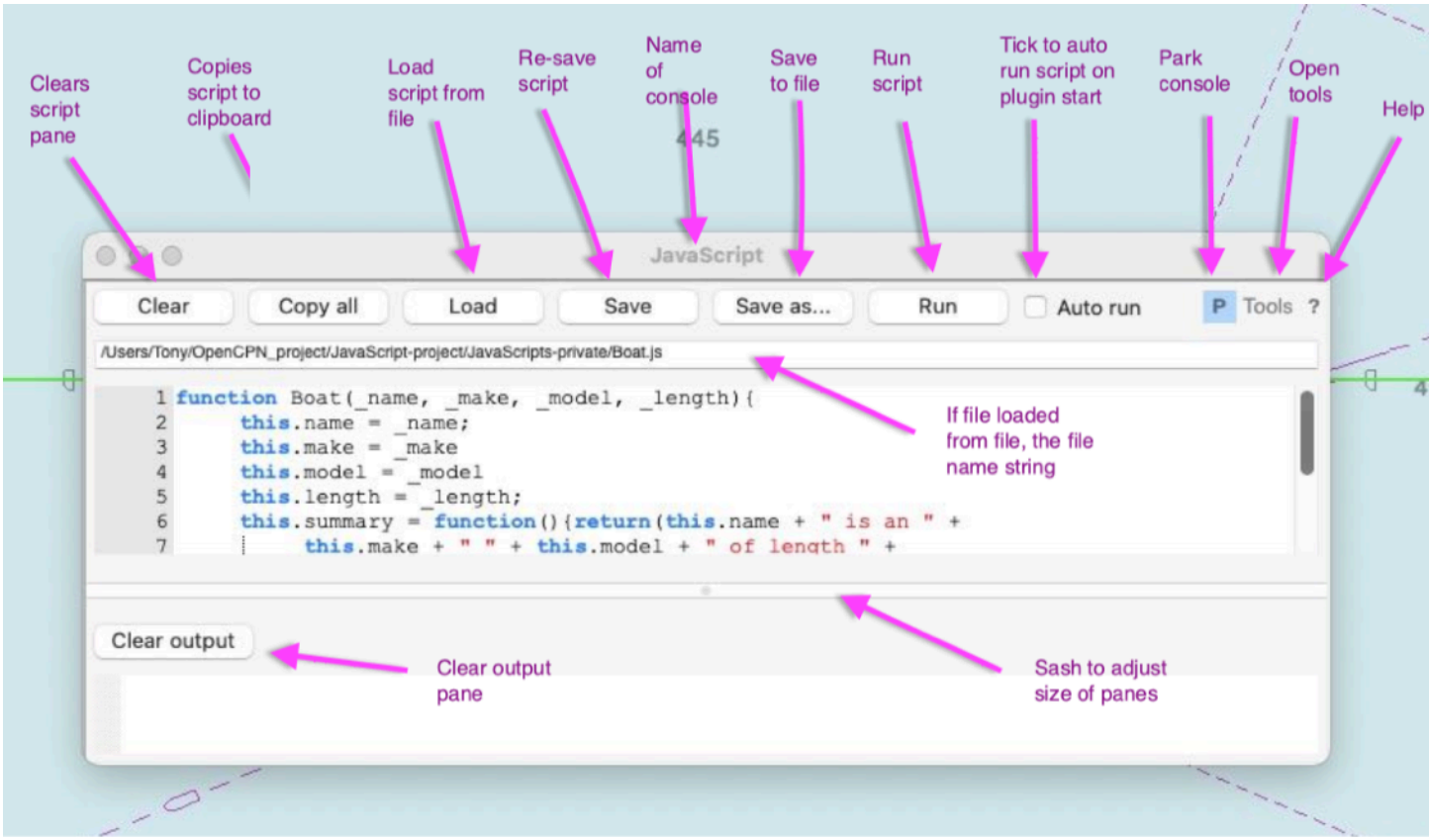
```
Fibonacci says: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181
```

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. The plugin colours the script to aid understanding as follows:

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



Layout of a console

This is how a console appears on MacOS. The details will differ on different platforms and icons may not be so clear. The above shows what is where.

A console contains two panes, one for the script and its buttons and the other for the output and its button. They are like two panes of a sash window.

There is a sash bar between them, and the space available for each can be changed by moving & the sash bar up or down.

You may find a console is in the way of other uses of OpenCPN. You can hide all consoles by toggling the JavaScript icon in the OpenCPN tool bar. Toggle it again to make all consoles visible. A script can hide itself (or another console). To make it visible again, you need to toggle the JavaScript icon once or twice until all consoles are visible. A hidden console will automatically become visible again if anything is written to its output pane.

Parking

A better way to reduce clutter is to park the console using the Park button in the top right of the console. This minimises it and parks it out of the way top left. You can unpark a parked console by clicking on its 'close' button. This restores it to where it was before it was parked. It remembers where it was parked, so re-parking will use the same parking space. If you hold the shift key down while unparking, the parking space is released and forgotten.

If anything is written to the output pane when a console is parked, it will unpark itself so you can see the output.

It is possible to configure where consoles park using the plugin tools.

A script can park its console or another console. See later.

When a script terminates, the *result* (see later) is delivered to the output pane, thus making any hidden or very small console visible. It is possible to suppress the result so that this does not happen.

The close button

The console close button has several functions other than actually closing the console, depending on circumstances. What it does is explained thus:

1. If a running script has set a Close button call-back function, that function is called.
2. Else if the console is parked, it will be unparked
3. Else the console is to be closed/deleted.
   a. If the console has an active script running, a prompt is given to stop the script.
   b. Else if the script window contains a script, a prompt is given to clear it
   c. Else the console is closed and deleted.

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 you have previously loaded scripts, `Load` will display a list of the ten most recently loaded scripts as well as an `Other` button. You can use the `Organise` button to move recent scripts into a Favourites list where they will endure.

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

You can also paste a script in from somewhere else.

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.

If you are online, Load also allows you to load a .js file from an https URL. The URL is recorded in the recent files and can be made a favourite. When the plugin restarts, it will reload a script from the URL. You might want save the script locally for offline use.

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

Script layout and trouble-shooting

It is helpful and conventional to indent script lines to show the structure of the script, e.g.

```
if (a == 6){
      print("a == 6\n");
      }
```

When you type a return to start the next line, the plugin adds indenting to where it thinks it should be.

⌘+F (Apple) or Ctrl+F (Windows/Linux) will re-flow the indentation over the whole script.

If compiling your script leads to `parse error - end of script`, this is almost certainly due to mis-matched brackets. Re-flowing the indentation will usually show where the issue lies.

File strings

Some functions access files on your computer 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

You can add extra consoles using the Consoles tab of the tools window. Each console can run its own script independently but can also communicate with other scripts. Typically, you may have a separate console for each task for which you are using the plugin.

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.

When OpenCPN terminates normally or the plugin is deactivated, the consoles, their positions and sizes are remembered for next time the plugin is activated.

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 in itself contain any input/output functionality, which is, inevitably, environment dependent.

I have implemented a number of extensions to provide interaction. Interactions with OpenCPN are covered in the next section. In this section we cover non-OpenCPN interactions..

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 character "\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");
```

printUnderlined(arg1, arg2…)

Displays the text underlined - useful for headings.

alert(arg1, arg2…)

This is similar to `print` but the output is displayed in an alert box. A trailing newline is not need here.

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 starting on a new line.

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

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

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 naturally until any alert has been dismissed by the user or by the script executing `stopScript()`.

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.

messageBox(message)

Displays a message box/. There are various options available

```
choice = messageBox(message);
choice = messageBox(message, "OK");
choice = messageBox(message, "");
```

All these display the message in a message box with `OK` and `Cancel` buttons. The message may contain "\n" to split it across lines.

```
choice = messageBox(message, "YesNo");
```

Instead of the OK button, Yes and No buttons are provided

By default, the message box has a caption identifying it is from the JavaScript plugin and the console name. You can supply your own caption as a third argument. An empty string suppresses the caption.

```
choice = messageBox(message,"YesNo","My caption");
```

The value of choice indicates which button was selected

1	OK
2	Yes
3	No

The cancel button raises a cancel error, which will normally terminate the script. You can handle the error yourself by catching it with the JavaScript try/catch capability - see Error Handling.

The message is displayed as a modal dialogue so the script does not continue until a button is selected. This is useful for multistep processing. It may impact the functioning of OpenCPN and the accessibility of other windows and its effect should be checked if conducting critical navigation. If called from within a callback, it will block all other callbacks. To avoid these issues, use `alert` or `onDialogue`.

require(moduleName)

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

toClipboard(text)

Copies the supplied text string to the clipboard.

See an example application here.

text = fromClipboard()

Returns the contents of the clipboard as a string.

text = cleanString(string)

When text is copied from documents or text editors, it may contain non-obvious obscure characters, which make it difficult to parse. This filter 'cleans' the given string by converting selected non-ASCII characters to ASCII equivalents. This filter is automatically applied to JavaScript code which might be pasted in from elsewhere. Examples:

‘`´	→	'	back & forward primes to prime
¨”	→	"	smart quotes to ordinary quote
º0	→	°	masculine ordinal and superscript 0 to degree

You can test this filter in the plugin tools diagnostics tab.

timeAlloc(milliseconds)

If a script takes too long to run, it will time out to avoid blocking other functions of OpenCPN. 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.

consolePark() or consolePark(name)

Parks the console out of the way or if a name is given that console is parked. Parking a console is usually better than hiding it as you can gain access to it by clicking on the Close button without toggling the hide/show status, which effects all the consoles.

If no other consoles are parked, it will be parked in the left-most position at the top of the screen.

If other consoles are already parked, the console will be parked to the right of the rightmost parked console.

When a parked console is moved to a different level, it is no longer regarded as parked.

If this script step is performed in a console already parked, it will be minimised in the same parking position.

The parking place and spacing has a default location which is platform dependent. You can use the `Parking` tab in the tools to set your own custom location. This takes you through manually parking two consoles by which the plugin learns how you want consoles to be parked.

consoleName(name)

Changes the name of the console to that given. Returns the name.

If the argument is omitted, the existing name is returned.

You can also change any console's name in the consoles pane of the tools.

stopScript() or stopScript(string)

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

keyboardState()

Returns a boolean structure representing the present state of certain keys on the keyboard.

Attribute	MacOS	Windows & Linux
.shift	Shift	
.control	control	Windows
.option	option	Alt
.menu	⌘ command	Ctrl

Example:

```
state = keyboardState();
if (state.shift && state.option) print("Shift and Command/Control keys down\n");
```

Check carefully that these keys do not have any undesired side-effects for your use.

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 null or the empty string `""`, the result is suppressed entirely.

```
scriptResult(null);
```

Callbacks

Often you will want a script to process an event, such as a timer expiring, a user clicking on a button or OpenCPN receiving data etc.

You do this by setting up a callback on the event and providing a handler to process the event. Here is a very simple example:

```
onSeconds(timesUp, 15);

function timesUp(){
      print("Time is up\n");
      }
```

The `onSeconds` call sets up a callback after 15 seconds to the function named `timesUp`. As written, this is a one-time action. Once the timer has fired, the callback is completed.

Most callbacks have an `All` variant, such as

```
onAllSeconds(timesUp, 15);
```

In this case, the handler will be called every 15 seconds until it is cancelled.

With plugin version 3.2 and earlier, all callbacks had to be to a named function, as in the above example. Now a callback can be to an anonymous function or an object's method.

Using an anonymous function, the above could be written:

```
onSeconds(function() {print("Time is up\n");},15);
```

Sometimes it is better to organise the event handling within an object.

```
clock = {  // clock object
      count:0,
      tick: function   (){   // clock method
            clock.count++;
            if (clock.count >= 5){
                  onSeconds(id);   // cancel this timer
                  print("Clock counted to 5\n");
                  }
            }
      }
id = onAllSeconds(clock.tick, 1);      // repeated calls to clock tick
method
```

Here the workings of the clock object are contained within it and each tick is handled by the tick method.

Cancelling callbacks

Sometimes you may want to cancel one or more callbacks. When a callback is set up, it returns an identifier that can be used to cancel the callback. In the above example, the identifier is captured as `id` and calling the setup API with just the id as an argument will cancel the callback. This action returns `true` if the callback was found and cancelled, else `false`;

If you call the API with no arguments, all callbacks of that type will be cancelled.

[For compatibility with previous versions of the plugin, an argument of `false` has the same effect but this use is deprecated.]

Sometimes you may want to know which callbacks are still in effect. If you call the API with the special query argument "?", it will return an array contains all identifiers existing for that type of callback. Consider the following:

```
function report(){print("Reporting\n");}
id1 = onSeconds(report, 1);
id2 = onSeconds(report, 2);
id3 = onSeconds(report, 3);

print("Step 1:", onSeconds("?"), "\n");
onSeconds(id2);
print("Step 2:", onSeconds("?"), "\n");
onSeconds();
print("Step 3:", onSeconds("?"), "\n");
```

Here, three callbacks are set up and their ids printed. The second callback is immediately cancelled and the ids for the remaining callbacks printed. Finally, the remaining callbacks are cancelled and the (empty) array printed. The actual output for me is

```
Step 1:[249,250,251]
Step 2:[249,251]
Step 3:[]
```

Timers

onSeconds(handler, seconds[, parameter])

Sets up a call to `handler` 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`.

Other callbacks

onDialogue(handler, 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 `stopScript()` 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.

Dialogues can be dismissed programatically.

It is also possible to create a modal dialogue, which suspends the script until the dialogue is dismissed.

For details, see the separate section Dialogues.

onCloseButton(handler)

Sometimes, you might want to gain control of a script that is running without stopping it. This lets you set up a call-back to a function when you click the console's close button while the ⌘-Command (Control/Alt on Windows) key is down.

Clicking on the Close button while the **option key** is down (Alt on Windows) will close all windows, thus closing OpenCPN.

3. OpenCPN APIs

This section documents how to interact with OpenCPN through a number of Application Program Interface functions or 'API's

Their names all start with `OCPN`, so it is clear you are interacting with OpenCPN.

Note for Windows users about the degree symbol

On Windows, the JavaScript engine cannot handle the degree symbol ° (code \u00B0) or any of the other characters that look like it.

To work around this, on Windows the plugin changes all incoming occurrences of ° (including in scripts), to an alternative code (\u0007 - bell) and all out-going occurrences of this alternative back to °. This is mostly not apparent to the user.

However, if you print the actual character code, you will see the substitute. This will also be the case if you examine the JSON representation of the symbol or match it by character code.

OpenCPN messaging

OpenCPN has a system of sending messages between itself and plugins or between plugins.

Each console keeps a record of the messages it has received.

OCPNgetMessageNames()

Returns an array of the OpenCPN message names seen since the plugin started.

Example:

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

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

reply = 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.

If the recipient responds directly, that response is returned. Otherwise `reply` is an empty string.

OCPNonMessageName(handler, messageName)

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

This example sets up a call-back to handleRT when a response is received to the message requesting a route's details.

```
routeGUID = "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 callback before sending the request to be sure the callback is in place when the message arrives.
- If the route GUID does not exist, the print will fail and throw an error. I am using JavaScript's `try & catch` to handle this.

As mentioned previously, call-backs are usually one-shot. But you can set up an enduring callback using the *All* variant

```
OCPNonAllMessageName(hanler, messageName)
```

Here functionName will be called each time the named message is received.

Cancelling message processing

The `OCPNonMessageName` and `OCPNonAllMessageName` calls return an identifier. You can use this to cancel the callbacks.

```
id = OCPNonMessageName(handleRT, "OCPN_ROUTE_RESPONSE");
```

and later

```
OCPNonMessageName(id);
```

You can cancel all message callbacks with

```
OCPNonMessageName();
```

Accessing OpenCPN's navigational data

You can access OpenCPN's navigational data in several ways.

```
navdata = 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

```
navdata = OCPNgetNavigationK();
```

If you want to explore this, you can print the structure.[1]

[1] Printing a JavaScript object in an easy-to-read format is covered in Examining objects

navdata = OCPNonNavigation(handler)

This is an alternative to `OCPNgetNavigation()`. The handler is called with the navigation data as its argument the next time the navigational data changes. The data structure is same as that retuned by OCPNgetNavigation except that .HDM and .nSats are not included.

navdata = OCPNonAllNavigation(function)

This invokes the function each time the navigational data changes.

It can be cancelled with a parameterless call:

```
OCPNonNavigation();
```

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

OCPNonActiveLeg(handler)

Sets up a handler to process the next active leg information received by the plugin. The function returns a structure containing the following attributes:

.markName	Destination waypoint name - abbreviated to six characters
.bearing	Bearing to waypoint
.distance	Distance to waypoint
.xte	The cross-tack error
.arrived	`true` if within the arrival circle, else `false`

Example:

```
OCPNonActiveLeg(processInfo);

function processInfo(info){
      print(info.distance, " nm to go to mark ", info.markName, "\n");
      }
```

Calling `OCPNonActiveLeg()` with no argument cancels any outstanding request.

Data Streams

OpenCPN has various data streams as set up in the Options Connections panel.

Prior to OpenCPN v5.8, all data streams were managed according to the settings in this panel. In v5.8 and later, you can direct output to a specific stream using its *handle* - a text string identifying it.

The following APIs can be used to work with handles:

handles = OCPNgetActiveDriverHandles()

This returns an array of text strings being the handles for the active streams.

We are advised treat the handles as opaque strings and not to interpret the data in them.

attributes = OCPNgetDriverAttributes(handle)

Given a handle, this returns a structure containing the handle's attributes.

The following script lists all attributes of all active streams

```
handles = OCPNgetActiveDriverHandles();
for (h = 0; h < handles.length; h++){
      attributes = OCPNgetDriverAttributes(handles[h]);
      print(handles[h], "\t", attributes, "\n");
      }
```

Get code

NMEA0183 Data

Sending data

OCPNpushNMEA0183(sentence)

OCPNpushNMEA0183(sentence, driverHandle)

`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:

```
OCPNpushNMEA0183("$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");
```

A check is made that the sentence starts with an NMEA heading. Officially, the maximum length of an NMEA sentence with checksum is 80 but OpenCPN allows much longer sentences.

If `driverHandle` is specified, the sentence will be pushed through the specified connection. Otherwise it will be pushed to all relevant connections.

The previous API `OCPNpushNMEA` is deprecated in favour of `OCPNpushNMEA0183`.

Receiving data

OCPNonNMEA0183(handler)

OCPNonNMEA0183(handler, ident)

The earlier form `OCPNonNMEAsentence` is deprecated

Sets up a callback to process the next NMEA0183 sentence received by the plugin. The callback returns a structure containing `OK` -a boolean value concerning the validity of the checksum - and `value` - the sentence itself. Example:

```
OCPNonNMEA0183(processNMEA);

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

If `ident` is omitted, any NMEA0183 sentence will be passed to the function. Only a single one of these 'any sentence' calls can be outstanding.

If `ident` is provided, it must be 3 or 5 letters specifying the sentence type to be received. If 5 letters, the talker ID is ignored. You can have multiple requests outstanding, allowing you to

handle different sentence types handled in different handlers. This is much more efficient than receiving every sentence and then deciding what to do depending on the sentence type.

You can have both one 'any sentence' and many specific sentence calls.

`OCPNonAllNMEA0183(functionName)`

`OCPNonAllNMEA0183(functionName, ident)`

With these variants the function is passed every sentence without the need to set up a further call.

`OCPNonAllNMEA0183()`

With no parameters, this cancels all outstanding NMEA0183 listening.

`checkChars = NMEA0183checksum(sentence)`

Returns a two character string being the correct checksum for sentence. Any existing checksum is ignored.

The plugin handles the NME0183 checksum internally. You do not normally need to check it or calculate it. However, this function returns the check characters, should you require them.

NMEA2000 Data

Working with NMEA2000 data is much more complicated than with NMEA0183. OpenCPN has the capability built in to receive NMEA2000 data for many data types (PGNs).

It is also possible to work with NMEA2000 data within JavaScript. You might want to look at the data in more detail than is available through OpenCPN or to send or receive PGNs not built in to openCPN.

Receiving data

`OCPNonNMEA2000(handler, pgn)`

`OCPNonAllNMEA2000(handler, pgn)`

Sets up a handler to process the next payload for the given PGN.

The handler will be called with three arguments as in this example:

```
OCPNonNMEA2000(handle, pgn);
function handle(payload, pgn, source){
      // payload is binary message
      // source is the OpenCPN handle for the input driver
```

The payload comprises an Actisense header followed by the NMEA2000 data

Byte offset	Containing
00, 01	0x93, xx
02	priority
03-05	PGN (binary)
06	destination
07	source address
08-11	timestamp (undefined in OpenCPN)
12	count for NMEA2000 data
13 onwards	NMEA2000 data

See Working with NMEA2000 for how to work with this.

When receiving navigational data that is regularly repeated, setting up one-shot handlers may be most convenient. But, if it is important to catch all data without missing some, the 'All' call would

be preferable. For example, if you request a response from all stations, the data could arrived rapidly and once only.

`OCPNonAllNMEA2000()`

With no parameters, this cancels all outstanding NMEA2000 listening.

Sending data

`OCPNpushNMEA2000(handle, pgn, destination, priority, payload)`

Sends out NMEA2000 data, where

handle	driver handle
pgn	the PGN of the data to be sent
destination	the destination address to which to be sent. 255 is all stations.
priority	priority of message
payload	binary payload as above

The binary data is difficult to work with. It is strongly recommended to use the plugin's NMEA2000 object, which can decode and encode payloads. See Working with NMEA2000 Data.

Position APIs

A position has two attributes - a latitude and longitude pair.

We can create a position thus:

```
myPosition = {latitude:61,longitude:2};
```

The 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}
```

The new position inherits its prototype from the start position.

OCPNgetGCdistance(Pos1, pos2)

Returns the great circle distance between two positions.

OCPNgetCursorPosition()

Returns the cursor position as a latitude and longitude pair.

Waypoint APIs

In the JavaScript plugin, waypoints have the following attributes

Property	Description	Notes
.GUID		
.position	.latitude .longitude	Required
.markName	The waypoint mark name - default is Circle	
.description	Free text description of waypoint	
.isVisible	`true` if waypoint is displayed - default ??	
.isNameVisible	`true` if waypoint mark name is displayed - default is `false`	
.iconName	Can be set in mixed text case but is always returned in lower case only	
.iconDescription	The icon name in mixed case, as in the property's window	Read only
.nRanges	Number of range rings - default is 0	
.rangeRingSpace	Space between rings - default 1.0	
.rangeRingColour	In HTML hex format - default #FF0000 (red)	
.useMinScale	If true, only display waypoint at large scale - default ??	
.minScale	If .useMinSCale is true, the minimum scale for waypoint to be displayed - default 1e9	
.hyperlinkList	Array of hyperlinks, each containing .description text to be linked .link The URL to link to .type (use unknown - do not use)	
.creationDateTime	Creation date/time in milliseconds since 1st January 1970 (to 1 second)	
.isFreeStanding	If `true`, this waypoint exists independently of any route and appears in the waypoints list. If `false`, this waypoint exists only by being included in one or more routes.	Read only
.routeCount	Count of the number of routes in which this waypoint occurs	Read only

Later in this guide in Objects and methods I describe the **position**, **waypoint** and **route** objects, which are the most powerful way of handling these concepts in JavaScript.

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

OCPNgetWaypointGUIDs(selector)

Returns an array of waypoint GUIDs.

`selector` determines which GUIDs are returned and can take the following values	
0 or selector omitted	objects not in layer
1	objects in a layer
-1	all objects both in layer and not

NB Prior to OpenCPN v5.8 and plugin v2.1 this selector is ignored and all objects are returned, as if it were -1. Thus the behaviour of `OCPNgetWaypointGUIDs()` has changed.

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
```

Note: the returned array includes the self-standing waypoints that are listed in the waypoints tab of the Route & Mark Manager and also routepoints that are only used in routes which are not included in that list.

OCPNgetActiveWaypointGUID()

Returns the GUID of the active waypoint or `false` if none.

OCPNgetSingleWaypoint(GUID)

Returns a waypoint structure 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 structure.

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 returns `the` GUID used. If you do not supply a GUID, you need to save the returned one if needed.

Example:

```
newWaypoint = {position:{latitude: 60, longitude:-1}};
newWaypoint.markName = "Near pub";
newWaypoint.iconName = "anchor";
newWaypoint.isVisible = true;
newWaypoint.description = "Good pub close by ashore";
GUID = OCPNaddSingleWaypoint(newWaypoint);
newWaypoint.GUID = GUID; // store the allocated GUID back in newWaypoint
```

OCPNupdateSingleWaypoint(waypoint)

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

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

Route APIs

Route structures have the following attributes

Attribute	Meaning
.GUID	
.name	Route name
.from	From text
.to	To text
.description	The route description
.isVisible	`True` if the route is being displayed else `false`
.isActive	`True` if this route is active
.waypoints	Array of waypoints in route

OCPNgetRouteGUIDs(selector)

Returns an array of route GUIDs. See OCPNgetWaypointGUIDs for specification of selector.

OCPNgetActiveRouteGUID()

Returns the GUID of the active route if any, else `false`.

When a route is active, `OCPNgetActiveWaypoint()` returns the GUID of the active waypoint within the route. If the same waypoint occurs more than once in the same route, it is not possible to learn which of its occurrences is the active one.

OCPNgetRoute(GUID)

Returns a route structure 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");
```

Throws an error if the route does not exist.

Any waypoints in the route that are not free-standing and are not included in any other route will be deleted.

GUID = OCPNaddRoute(route)

Adds a route into OpenCPN. The argument must be a route structure, which should contain an array of waypoints.

If a waypoint is an existing one, only the GUID need be supplied. The other attributes will be ignored.

If a waypoint is new, do not include a GUID. A new point will be created using the attributes of the point.

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.

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.

This API can be used to

- update the attributes of the route structure as listed above
- add, remove or reorder waypoints in the route

Note: if an included waypoint has no GUID, it will be added as a new routepoint. If it has a GUID, that will be assumed to be an existing waypoint and the rest of the waypoint structure is ignored.

If you want to update a waypoint's details, use the OCPNupdateSingleWaypoint function on that waypoint.

Track APIs

Attribute	Meaning
.GUID	
.name	Track name
.from	From text
.to	To text
	St
.waypoints	Array of trackpoints, each of which comprises a subset of a waypoint: .creationTimeStamp position.latitude

OCPNgetTrackGUIDs(selector)

Returns an array of route GUIDs. See OCPNgetWaypointGUIDs for specification of `selector`.

GUID = OCPNaddTrack(track)

Adds a track into OpenCPN. The argument must be a track structure, which should contain an array of trackpoints.

If a trackpoint is an existing one, only the GUID need be supplied. The other attributes will be ignored.

If a trackpoint is new, do not include a GUID. A new point will be created using the attributes of the trackpoint.

If `track.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.

OCPNgetTrack(GUID)

Returns a track structure for the given GUID, complete with an array of the trackpoints. Throws an error if the track does not exist.

OCPNdeleteTrack(GUID)

Deletes a track, given the GUID.

OCPNupdateTrack(track)

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

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

This API can be used to

- update the attributes of the track structure as listed above
- add, remove or reorder trackpoints in the track

Note: if an included trackpoint has no GUID, it will be added as a new trackpoint. If it has a GUID, that will be assumed to be an existing trackpoint and the rest of the trackpoint structure is ignored.

Changing details of a trackpoint is not supported.

Notifications

You can manage OpenCPN notifications. In a script, a notification has the following attributes:

Attribute	Meaning
level	A number being the importance of the notification 0 informational 1 warning 2 critical
message	The text message displayed with the notification. It can be multilined.
timeStart	The time in seconds initially set for the notification to automatically self-dismiss
timeRemaning	The time in seconds before the notification will automatically self-dismiss
guid	A string being the notification's GUID
action	"POST" or "ACK" only present in response to `OCPNonNotificationAction` (see below)

```
guid = OCPNraiseNotification(level, message [,timeRemaining])
```

Raises a notification and returns its GUID. If timeRemaining is omitted or set to -1, the notification is given the default timeRemaining for its level.

```
ok = OCPNackNotification(guid)
```

Given a notification GUID, this acknowledges it. Returns `true` if successful or `false` if `guid` does not match an existing notification.

```
notifications = OCPNgetNotifications()
```

Returns an array of the notifications with attributes as above, excluding `action`. This includes all notifications - not just those created by the script. A script could, therefore, acknowledge any OpenCPN notification - not just those it has created.

```
id = OCPNonNotificationAction(handler)
```

Calls the handler when there is notification action I.e. a notification is raised or acknowledged. The handler is given the notification as in the above table, including the attribute for the action.

```
id = OCPNonAllNotificationAction(handler)
```

As above but the handler is called on every notification action.

```
OCPNonNotificationAction(id)
```

Cancels the notification action with the id.

```
OCPNonNotificationAction("?")
```

Returns an array of outstanding notification action ids.

```
OCPNonNotificationAction()
```

Cancels all notification action function calls

Notes on notifications

- If the same notification is repeated, in the notifications panel they are summarised with a repeat count rather than displaying them in multiple lines. The individual notifications exist within OpenCPN.
- If the same notification has been repeated with different time-outs, all those notifications will time-out when the shortest time expires.

- If you acknowledge a notification, it acknowledges all notifications whose text matches it - not just the one matching the GUID. You can acknowledge using any of the GUIDs for that notification.
- The colour of the notification icon on the chart corresponds to the maximum level of any notification. However, the icon on the list of notifications corresponds to the level of the first notification. It is not changed by raising the same notification at a different level.
- If you set up a function to be called on a notification action, and a repeated notification is acknowledged, the function will be called multiple times as each notification instance is acknowledged.
- There is nothing to prevent you using `OCPNonNotificationAction` to call multiple functions.

Miscellaneous APIs

`OCPNsoundAlarm()`

Plays an alarm sound. Returns `true` if successful.

`OCPNsoundAlarm(file)`

Plays the sound file specified in the `file`. Returns `true` if successful.

`file` is resolved in the same way as when reading a file. So you could use the "?" convention to open a file dialogue.

OCPNgetAISTargets()

Returns an array of the AIS objects each with the following attributes. Not all attributes are present. Notably, class B targets transmit only a subset.

Attribute	Meaning
.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 ºT
.SOG	Speed over ground
.COG	Course over ground
.HDG	Heading
.ROTAIS	Rate of turn
Derived values	These are not transmitted but derived by OpenCPN
.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

OCPNonAIS(handler)
OCPNonAllAIS(handler)

The `OCPNgetAISTargets()` API above returns AIS data as interpreted by OpenCPN.

You can also receive the raw AIS data as text strings using these calls. They are called in the same way as `OCPNonNMEA0183()` described earlier.

The returned text needs to be decoded to make any sense.

OCPNgetNewGUID()

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

OCPNgetPluginConfig()

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

Attributes	.PluginVersionMajor	Plugin version
	.PluginVersionMinor	
	.patch	Patch number
	.comment	Any comment about the version
	.ApiMajor	OpenCPN API version number
	.ApiMinor	
	.wxWidgets	wxWidgets run-time version number
	.DuktapeVersion	Duktape JavaScript engine version number
	.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.

OCPNgetCanvasView()

Returns a selection of attributes of the present canvas viewport.

Attributes	.centre	The centre of the canvas as a lat & long position
	.ppm	The current scale in screen pixels per metre
	.chartScale	The scale of the chart. This scale is displayed bottom right as 1:<scale>. The displayed number is rounded and may differ slightly.
	.pixWidth	The width of the canvas in screen pixels
	..pixHeight	The hight of the canvas in screen pixels

OCPNcentreCanvas(position [, ppm]);

Centres the canvas at the given position. An optional second argument can provide a new scale, expressed in screen pixels per metre. If omitted, the scale is unchanged.

If you want to zoom in or out when centring the chart, you can make the scale relative to the current scale with something like this:

```
// centre and zoom out x4
OCPNcentreCanvas(pos, OCPNgetCanvasView().ppm/4);
```

OCPNrefreshCanvas()

Refreshes the canvas window. If your script has made changes to displayed information and the display is not being updated,, this will update the display.

OCPNisOnline([timeout])

Returns true if on-line - otherwise false.

If the optional parameter is given as 0, it only checks if OpenCPN has a connection, which could be to a local network which might not be connected through to the internet.

If the parameter is omitted or > 0 , a check is made for a full internet connection. The value of the parameter is used as a timeout for this test. It defaults to 2 seconds.

OCPNformatDMS(1|2, number [, precision])

Given a number being a lattitude or longitude, returns a text string formatted in degrees, minute and second, as set in the user's display preferences.

The first parameter is either 1 for a latitude or 2 for a longitude.

The optional precision is true (the default) for a high precision string or false for normal precision.

```
print(OCPNformatDMS(2, -60.5, false), "\n"); // prints 060° 30.0' W
```

This API is used internally by the Position object but could be called directly if required.

OCPNparseDMS(string)

Given a string being a latitude or longitude in degrees, degrees and minutes or degrees, minutes and seconds and the hemisphere, returns the latitude or longitude as a number.

If no number can be deduced, it returns `NaN`.

This API uses OpenCPN's built-in parser.

The optional precision is true (the default) for a high precision string or false for normal precision.

```
print(OCPNparseDMS("60° 30'W"), "\n");   // prints -60.5
```

This API is used internally by the Position object but could be called directly if required.

Context Menu Actions

You can create context menu items and specify a handler for when you select the menu item.

OCPNonContextMenu(handler, menuItem [,info])

Adds a menu item `menuItem` to the context menu. When this is selected, the handler is invoked and passed information as described in the table below.

You can also add content menus to waypoints, route, tracks or AIS targets with the variants

OCPNonWaypointContextMenu(handler, menuItem [,info])

OCPNonRouteContextMenu(handler, menuItem [,info])

OCPNonTrackContextMenu(handler, menuItem [,info])

OCPNonAISContextMenu(handler, menuItem [,info])

All these are 'one time' - once the item ha been selected, it is removed.

All the above have 'all' variants that persist until cancelled. They can be selected repeatedly. These are

```
OCPNonAllContextMenu(handler, menuItem [,info])
OCPNonAllWaypointContextMenu(handler, menuItem [,info])
OCPNonAllRouteContextMenu(handler, menuItem [,info])
OCPNonAllTrackContextMenu(handler, menuItem [,info])
OCPNonAllAISContextMenu(handler, menuItem [,info])
```

Cancelling context menus

All of the above calls return an identifier that can be used to cancel the menu by calling with just the identifiers a parameter.

```
id = OCPNonAllWaypointContextMenu(handler, "Extra waypoint action");
```

and later cancelled with

```
OCPNonAllWaypointContextMenu(id);
```

Calling any of the above with no parameter will cancel all context menus for the script.

```
OCPNonContextMenu();
```

You can also query which context menus are active

```
ids = OCPNonContextMenu("?");
```

This returns a numeric array of the ids still active. If none, it will; be an empty array.

Handling context menu selections

The handler is passed a structure with the following attributes

Cursor position	.latitude	
	.longitude	
	.objecttype	"" (Empty string) for an canvas context menu, otherwise "Waypoint","Route", "Track" or "AIS"
For waypoints, routes or tracks	.GUID	The object's GUID
For AIS	.MMSI	The target's MMSI
	.menuName	The name of the menu selected
	.info	The 'info' parameter from the set up call or an empty string none.

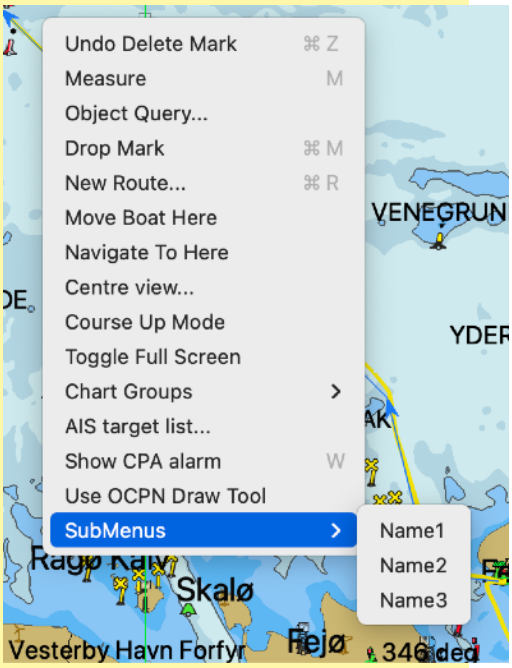
Although the structure contains extended information, it is a valid JavaScript position and can be used as such. Example:

```
menuName = "Drop Anchor";
OCPNonAllContextMenu(dropMark, menuName);

function dropMark(location){
      waypoint = {
            position: location,
            markName: "Anchorage", iconName:"Anchor",
            minScale: 52000, useMinScale: true,
            }
      OCPNaddSingleWaypoint(waypoint);
      }
```

This script adds a context menu item Drop Anchor, which can be used to drop a bespoke anchor mark, which, as written here, only displays at large scale.

Submenus

It is also possible to create a branch of submenus.



Each submenu item can have its own handler or can share a handler. For convenience, there is a submenu constructor that can be used for this.

You need to create the branch and use it place of the handler in the set up call. The illustration was created with:

```
contextSubMenu = require("contextSubMenu");
myMenu = new contextSubMenu();
myMenu.add(display1, "Name1");
myMenu.add(display2, "Name2");
myMenu.add(display1, "Name3");
id = OCPNonContextMenu(myMenu, "SubMenus");
```

Notes that in this example two of the submenus share the handler `display1` while the other does not.

The submenus are a single group. In this example, when any one of them is selected, the entire group will be removed.

To cancel the submenus, you use the id retuned by the set up.

```
OCPNonContextMenu(id);
```

This will cancel the entire group. You cannot cancel individual submenu items.

4. JavaScript objects and methods

JavaScript supports the use of objects. These can be a convenient way of representing complex data structures together with their own properties and methods.

You can construct your own objects within your script or load a constructor from a file using the `require` function. By convention, we give constructors an initial capital letter as a reminder that they are constructors.

The JavaScript plugin has a library of useful constructors for constructing relevant objects and these are described here:

Position constructor

`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 optionally
		1. latitude & longitude values, e.g.
		myposition = new Position(60, -1.5);
		2. a latitude & longitude pair, e.g.
		myposition = new Position({latitude:60,longitude:-1.5});
		3. an existing position object e.g.
		myposition = new Position(existingPosition);
		4. A position text string e.g.
		myPosition = new position("20° 14.56'N 2° 1.5'W");

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
	.parse(position string)	Sets the position by parsing a formatted text string. Handles many styles in which people write positions, such as 12.34', 12'34, 12.'34.
	.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
```

Waypoints constructor

`Waypoint()`

constructs empty waypoint object with its methods

`Waypoint(lat,lon)`

constructs waypoint object for the given latitude and longitude

`Waypoint(position)`

constructs waypoint object for the given position

`Waypoint(waypoint)`

constructs waypoint object for the given waypoint, thus adding the waypoint object methods to a waypoint structure.

The constructed `waypoint` object is as follows:

To load constructor	require("Waypoint")	
Attributes		As for waypoints
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.
	.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 constructor

`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
	.description	The description field
	.waypoints	An array of the waypoints in the route, each being a waypoint object.
	.isVisible	Determine whether the route is displayed.
	.isActive	Read-only. The value is ignored when updating OpenCPN
	.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.

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

Adding attributes to a bare object

Many of the objects created by the above constructors include methods you can call or accessors such as the `.formatted` string in a position.

Most objects created by an OpenCPN API are 'bare' - they do not include methods etc.

If you want add the methods to a bare object, there are two ways.

Some of the constructors will accept a bare object as their argument.

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

Another way is to use a JavaScript statement that adds the missing attributes.

```
Object.assign(bareRoute.prototype, fullRoute);
```

This assigns to `bareRoute` prototype the prototype of `fullRoute`. This technique can be used even when the constructor does not accept a bare object as an argument.

5. Modules

Optional modules are loaded using the `require` function.

The following modules are included with the plugin as standard

"Position"	Returns the Position object constructor	
"Waypoint"	Returns the Waypoint object constructor	
"Route"	Returns the Route object constructor	
"Track"	Returns the Track object constructor	
"File"	Returns the File object constructor	
"NMEA2000"	Returns the NMEA2000 object constructor	See Working with NMEA2000
"pgnDescriptors"	Takes a numeric argument being a NMEA2000 PGN. Returns the PGN descriptor. If the argument is empty, it returns an array of all the descriptors. Example: `descriptor = require("pgnDescriptors")(129029);`	
"canboatAnalyzer"	Returns the canboatAnalyzer function.	
"shell"	Returns a shell function	
"pluginVersion"	Takes an argument being the minimum plugin version required by the script. Throws an error if this is not met. Example: `require("pluginVersion")("3.2.1");`	

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.:

```
fibonacci = 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 by assigning things to its `this` value. 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
```

6. 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

7. Working with files

Files can accessed in the following modes[2]:

Mode	Meaning
`READ`	File made available read-only
`WRITE`	File will be over-written. No read access.
`READ_WRITE`	Available for read and write. File can be updated.
`APPEND`	Data will be appended to existing file. No read access.
`WRITE_EXCL`	Like WRITE but the file must not already exist

Files are located using a `fileString`. A simple `file name` can be converted to a `fileString` with

```
fileString = getFileString(file[, mode=READ])
```

What happens

Argument	What happens	Notes
Is `fileString`	No change	Includes one or more directories
Is `fileName`	Looks for the file in the plugin current directory and returns the `fileString`	
Starts with `'??'`	A dialogue is opened so you can choose the file. The text after "??" is used as the prompt.	If mode is WRITE or WRITE_EXCL, it will be a save dialogue, otherwise an open dialogue.
Starts with single `'?'`	The rest of the string will be treated as a fileName or fileString. If the file is not found, a dialogue will be opened so you can choose the file.	

Simple file access

`readTextFile(file)`

`file` is passed through getFileString and so can be a file name or fileString and can start with '?' or '??'.

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

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

The whole file will be loaded into memory. This is not wise for large files. For these see the File object below.

If `file` commences "https:" or "http:", it is taken to be a URL and the file will be read from that location. NB OpenCPN will be blocked while the file is read.

[2] These modes are those supported by wxWidget's wxFile class

writeTextFile(text, file, access)

Writes the text to the `file`.

Access	Mode used
0	`WRITE_EXCL`
1	`WRITE`
2	`APPEND`

Example to append a sentence to a file:

```
sentence = "$PCDIN,01F802,000C8286,09,3AFC8CCA0500FFFF";
writeTextFile(sentence, "/Users/Tony/myfile.txt", 2);
```

The File object

The file object provides a flexible way to work with large files, whether text or binary. The File object constructor must be loaded and a file object constructed for each file being used.

```
File = require("File");     // load the constructor
myFile = new File("myData.txt" [, mode]);
```

The file name string follows the rules for `getTextFile()`, so if it commences with '?', a dialogue will be opened. The file object has the following properties

Property	Description	Notes
`.length()`	Returns length of the file in bytes	
`.eof()`	`true` if the current position is at file end	
`.seek(offset)`	Sets the current position to `offset` and returns the new offset.	The maximum offset is length - 1. If `offset > length-1`, offset is left at end of file and that offset is returned.
`.tell()`	Returns the present offset.	
`.getAllText( )`	Returns all data as text	Not suitable for large files
`.getTextLine( )`	Each successive call returns one line of text, starting from the beginning of the file.	Excludes any newline character \n
`.getText(count)`	Returns `count` characters starting from present position	The number of characters returned will be less than count if the end of file is encountered.
`.getBytes(count)`	As for .getText but returns a Uint8Array (array of unsigned 8-bit bytes).	The length of the array will be less than count if the end of file is encountered.
`.writeText(string)`	Writes the string starting at the current offset.	
`.writeBytes(Uint8Array)`	Writes a byte array starting at the current offset.	
`.fileString`	The fileString for the file.	Read-only attribute

`.getTextLine()` accesses the file in blocks and can be used to read large files without using large amounts of memory. You could process a large text file with

```
while (!myFile.eof()){
      textLine = myFile.getTextLine();
      // do something useful with textLine
      }
```

The first call to `getNextLine()` returns the first line in the file. Successive calls return the text line-by-line. Any call to `seek()` causes `getNextLine()` to start again at the beginning, regardless the offset given.

The File object can be used for complex file operations. The following example replaces bytes at offsets 100 & 101:

```
File = require("File");
myFile = new File("testWriteData.txt", READ_WRITE);
data = new Uint8Array(2);
data[0] = 111; data[1] = 222;
myFile.seek(100);
myFile.writeBytes(data);
```

MacOS user: For security reasons, Apple restricts ad hoc access to iCloud disk space.

Scripts will be unable to write to this area unless you grant permission for OpenCPN to access it. In that case, you could scribble over any file. It is good to place your files outside iCloud.

8. The _remember variable

Sometimes you may want to remember information from one script run to another - such as some preference choices.

The plugin has a special variable `_remember`, which is remembered from one script run to another and across OpenCPN restarts. The `_remember` variable is unique to each console.

You can set `_remember` to anything you like - a number, string or object and it will be available to the next script run. By making it an object, you can remember several different things.

This example script remembers OpenCPN's navigation object when it wraps up.

On each run, it prints the position from the previous run after checking _remember has been set.

```
onExit(wrapUp);
if (_remember != undefined)
      print("Last position for previous run was ",
            _remember.position, "\n");

function wrapUp(){
      _remember = OCPNgetNavigation();
      }
```

This example uses `_remember` to remember a log file name. The first time it runs, it will open a save dialogue. On subsequent runs it will use the same file without asking again.

```
File = require("File");
_remember = getFileString("?" + _remember);
logFile = new File(_remember, APPEND);
logFile.writeText("Data to be logged\n");
logFile.writeText("More data to be logged\n");
```

You might use the same console for a different purpose and _remember might be left over from a previous use. It can be useful to store a flag for the particular script by which you can check for a first time run.

```
if ((_remember == undefined)
      || (!_remember.hasOwnProperty("myScriptFlag"))){
      // first time actions
      ...
      _remember.myScriptFlag = true;
      }
```

9. 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.

It is wise to always display the error message. The error might not be what you think it is and without that information you could look for the problem in the wrong place.

Throwing an error from a script

If a script detects that an error situation should terminate it, you can use the standard JavaScript `throw` statement. Consider the following script:

```
outer();

function outer(){
      middle();
      }

function middle(){
      inner();
      }

function inner(){
      bullseye();
      }

function bullseye(){
      throw("Inside bullseye");
      }
```

When the function bullseye is executed, the script will throw the error and display

```
Inside bullseye
```

As the script does not terminate normally, there is no result to display.

The above, however gives no indication where the throw is and how it came to be thrown. This would not matter in a simple script but in a complex script it might be difficult to understand how it came about.

You can get much better tracing information by throwing an Error object, which includes traceback information. So, in the above example, if we change the throw to

```
Throw Error("Inside bullseye");
```

We get the much more informative

```
inner line 12 Error: Inside bullseye
called from middle line 8
called from outer line 4
called from line 1
```

10. 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

11. 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(handler, dialogue)
```

```
onAllDialogue(handler, dialogue)
```

where `handler` is 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.

Starting with plugin v4, you can construct multiple dialogues, each with their own handler.

`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 handler is given two arguments:

1. 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.
2. The label of the button selected

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.

type	Purpose	Other attributes Grey items are optional	Explanation
		sufix:"text"	Suffix text to be displayed after the field, e.g. "°T"
		fieldStyle	See below on styling
"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. If a button name starts with "*" that is omitted from the name and it becomes the default button. 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 In the returned structure, value is the selected value.
		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. In the returned structure, value is the selected value.

type	Purpose	Other attributes Grey items are optional	Explanation
		label:"text"	Text to form label for the spinner. Omit to suppress label.
"hLine"	Horizontal line	None	Adds a horizontal line as a separator.
"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.
"position"	Set position for this dialogue	x:<number>, y:<number>	Specifies the position in Device Independent Pixels for the dialogue. If not specified, the default position is that of the previous default position.
"colours"	Change the colour of the text or background		The colours are specified as strings like "RED", "GOLD". The colours available are as in the table below.
		text:<colour>	Sets the colour of the text from this point forward in the constructed dialogue. You can change the colour used at different points.
		background:<colour>	If you set the background to a dark colour, you will need to set the text to a light colour as the default black will not show against a dark background.

Colour names

AQUAMARINE	FIREBRICK	MEDIUM FOREST GREEN	RED
BLACK	FOREST GREEN	MEDIUM GOLDENROD	SALMON
BLUE	GOLD	MEDIUM ORCHID	SEA GREEN
BLUE VIOLET	GOLDENROD	MEDIUM SEA GREEN	SIENNA
BROWN	GREY	MEDIUM SLATE BLUE	SKY BLUE
CADET BLUE	GREEN	MEDIUM SPRING GREEN	SLATE BLUE
CORAL	GREEN YELLOW	MEDIUM TURQUOISE	SPRING GREEN
CORNFLOWER BLUE	INDIAN RED	MEDIUM VIOLET RED	STEEL BLUE
CYAN	KHAKI	MIDNIGHT BLUE	TAN
DARK GREY	LIGHT BLUE	NAVY	THISTLE
DARK GREEN	LIGHT GREY	ORANGE	TURQUOISE
DARK OLIVE GREEN	LIGHT STEEL BLUE	ORANGE RED	VIOLET
DARK ORCHID	LIME GREEN	ORCHID	VIOLET RED
DARK SLATE BLUE	MAGENTA	PALE GREEN	WHEAT
DARK SLATE GREY	MAROON	PINK	WHITE
DARK TURQUOISE	MEDIUM AQUAMARINE	PLUM	YELLOW
DIM GREY	MEDIUM BLUE	PURPLE	YELLOW GREEN

Simple example:

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

function action(dialogue, label){
      if (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 individual parts of 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. As a special case, if the font name is set to "monospace", a monospaced font will be used, which is useful when displaying tabular data over multiple lines.	✓
italic:<bool>	e.g. italic:true	✓
bold:<bool>	e.g. bold:true	✓

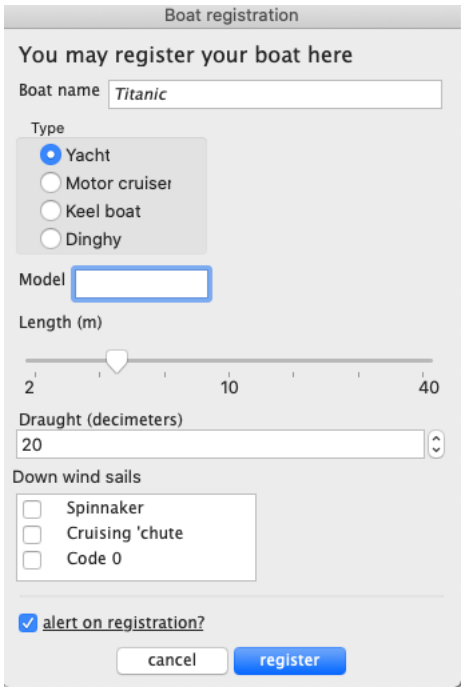
underline:<bool>	e.g. underline:true	

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.



Cancelling a dialogue

As with other callbacks, `onDialogue` and `onAllDialogue` return a callback ID. This can be used to cancel the dialogue.

In this example, the dialogue is cancelled after 10 seconds:

```
dialog = [{type:"text", value:"This is some text"}];
id = onDialogue(function(outcome){print(outcome, "\n");}, dialog);
onSeconds(function(){onDialogue(id);}, 10);
```

Modal dialogues

Using a callback can make for complicated code, especially if a sequence of dialogues is required. An alternative is to create a model dialogue, which suspends the script until the dialogue is dismissed.

```
outcome = modalDialogue(dialogue)
```

The returned outcome is the dialogue as passed to a handler.

Cancelling a modal dialogue

Like non-modal dialogues, modal dialogues use the unified callback mechanism, so it is possible to cancel them. To do this, you need to find the callback ID. Consider the following script:

```
dialog = [{type:"text", value:"This is some text"}];
onSeconds(function(){
      ids = onDialogue("?");
      onDialogue(ids[ids.length-1]);
      }, 10);
outcome = modalDialogue(dialog);
onSeconds();
if (outcome == undefined) print("dialogue timed out\n");
else print("Outcome: ", outcome, "\n");
```

This creates a modal dialogue. On return from the modal, any timer is cancelled and the outcome printed.

Before the dialogue is created, a 10 second timer is started. If this fires, it fetches the array of callback IDs. The last one in the array will be that for the just-created modal dialogue and it uses this to cancel the model dialogue. The script resumes with a return from the modal call. In this case, the returned outcome is undefined.

Caution when using modal dialogues

Modal dialogues suspend the script until the dialogue is dismissed or cancelled as above.

Meanwhile, other callbacks will be processed, as can be seen from the `onSeconds` callback in the above example.

If a callback throws an error while the modal dialogue is open, the dialogue will be left ownerless. This may crash OpenCPN or leave it in modal mode with no modal window to dismiss. Make sure that any callbacks do not throw errors.

If a modal dialogue is open, `stopScript()` will not stop the script but display an error message.

12. Automatically running scripts

It is possible to arrange for a script to be automatically loaded and run when the plugin is started, 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 find the offending console name and change `AutoRun=1` to `AutoRun=0`. This will stop the script running automatically on launch.

13. Working with multiple scripts in one console

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.

14. 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

Working with multiple consoles in scripts

For the APIs in this section, you need to load them 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.

You cannot close the console running this script step.

consoleGetOutput(consoleName)

Returns the contents of the output pane of the named console.

consoleClearOutput(consoleName)

Clears the contents of the output pane of the console.

If the argument is omitted, it clears its own output.

consoleLoad(consoleName, script)

If script ends with .js, it will load the script from a file of that name. Otherwise, script is taken to be a JavaScript and that is loaded.

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(handler, consoleName [, 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 timed out 2 other script completed normally 4 other script executed a scriptStop( ) step 8 other script's console was closed by another script
.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.

Notes:

If you use `onConsoleResult` to run a script which chains itself to a further script or scripts, the result returned is that from the final script in the chain, unless an error is encountered, when the error is returned.

For compatibility with previous plugin versions, the alternative argument order
`onConsoleResult(consoleName, handler [, brief]`
is accepted but deprecated.

15. 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(handler)`

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

The handler is called during the wrapping up process. Some actions within such a handler are meaningless. For example, if call-backs are set up they will have no effect. If the onExit handler itself 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 handler in a situation where the console remains visible.

The following calls are not allowed in the handler and will throw an error:

- require()
- stopScript()
- Any function setting up a callback, such as OCPNonSeconds

The handler will not be called when a script terminates because

- An error has been thrown
- The console has been closed, including closed by another script
- The plugin is deactivated, including when OpenCPN is shut down

17. Working with SignalK

OpenCPN processes certain SignalK messages. If you wish, you can process SignalK messages in a JavaScript, which has a natural affinity with JSON.

The following script shows how to use the OpenCPN message interface to receive SignalK messages. This illustration script simply prints out certain information.

```
// Listen out for SignalK messages and display
Position = require("Position");
OCPNonMessageName(received, "OCPN_CORE_SIGNALK");

function received(message){
      signalK = JSON.parse(message);
//    uncomment next line to pretty-print object
//    print(JSON.stringify(signalK, null, "\t"), "\n\n");
      for (u = 0; u < signalK.updates.length; u++){
      update = signalK.updates[u];
      timeStamp = update.timestamp;
      sentence = update.source.sentence;
      values = update.values;
      for (v = 0; v < values.length; v++){
            what = values[v].path;
            value = values[v].value;
            }
      switch (sentence){
            case "GLL":
                  position = new Position(value);
                  print("Position at\t\t", timeStamp, " is\t",
                        position.formatted, "\n");
                  break;
            case "VTG":
                  cog = values[1].value;
                  sog = values[2].value;
                  print("Over ground at\t\t", timeStamp, " is\tCOG:",
                        cog, "\tSOG:", sog,  "\n");
                  break;
            case "VHW":
                  hdt = values[0].value;
                  stw = values[1].value;
                  print("Through water at\t", timeStamp, " is\tHDT:",
                        hdt, "\tSTW:", stw,  "\n");
                  break;
            default:
            }
      OCPNonMessageName(received, "OCPN_CORE_SIGNALK");
      };
```

Get Code

Sample output:

```
Position at      2023-11-13T13:39:36.000Z is 49° 42.903'N 003° 35.039'W
Over ground at   2023-11-13T12:51:05.808Z is COG:123.4  SOG:2.34
Through water at 2023-11-13T12:51:05.814Z is HDT:125.2  STW:2.86
```

18. Working with NMEA2000

An NMEA2000 message type is known as its PGN (Parameter Group Number).

OpenCPN includes decoders for some 50 PGNs and uses these to handle navigational data. Plugins such as Dashboard use them to display a variety of data.

The JavaScript plugin can independently decode and give access to any message for which there is a known description - presently 373.

Background

Unlike NMEA0183 and Signal K, NMEA2000 messages are complex and coded in binary. How they are encoded is proprietary and available only to NMEA2000 licensees, who are bound by a non-disclosure agreement. However, a few projects have worked out how the various PGNs are encoded.

The canboat project has published a list of 'descriptors' - computer readable descriptions of the PGNs and created utilities written in C++ to process messages according to these descriptors.

A daughter project canboatjs has replicated the code in JavaScript. However, it is a large complex suite of scripts using the latest JavaScript concepts not available to the JavaScript plugin and is not usable for our purpose here.

The JavaScript plugin can independently decode NMEA2000 messages using the canboat descriptors.

The NMEA2000 object

The NMEA2000 object has 'beta' status. More extensive testing could reveal issues. There could be changes. Decoding binary has only had limited testing and is limited to 32 bits.

A script can construct any NMEA2000 object for which there is a canboat descriptor and can be used to decode NMEA messages into JavaScript objects. An example with comments illustrates this.

```
Nmea2000 = require("NMEA2000");
// by convention, constructors are given an initial capital letter
// to remind us it is a constructor and not an object

// construct an NMEA2000 object for 129029 (GNSS Position Data)
NMEA129029 = new Nmea2000(129029);

// the bare object has the attributes of PGN 129029 but the values are
// undefined. You could pretty-print it using
print(JSON.stringify(NMEA129029, null, "\t"), "\n");
```

Decoding received data

The object's decode method will decode a message's data. Example:

```
Pos = require("Position"); // constructor so we can format positions
OCPNonNMEA2000(handle129029, 129029); // listen for message

function handle129029(payload, pgn, source){
      fix = NMEA129029.decode(payload);     //decodes data in the payload
      position = new Position(fix.latitude, fix.longitude);
      print("Position: ", position.formatted, "\n");
      print("Fix time: ",  fix.date + " " + fix.time, "\n");
      print("using ", fix.numberOfSvs, " satellites\n");
      }
```

Sample output:

```
Position: 52° 37.143'N 001° 28.919'E
Fix time: 2023.10.22 17:34:59
using 12 satellites
```

Shortcut

The above code constructs the NMEA129029 object once at the start and can use it for repeatedly decoding PGN129029 messages. If you are not repeatedly using it, you could construct and decode in one step by including the data for the constructor:

```
fix = new Nmea2000(message.pgn, message.data);
```

Encoding

Having set the attributes of an NMEA2000 object, you can obtain the encoded data thus:

```
data = fix.encode();
```

Sending data

If you wish to send the data out, you can do so directly with the object's push method. This encodes the object into an NMEA2000 payload and sends it out in one step.

```
Nmea2000 = require("NMEA2000");
nmeaSend = new Nmea2000(59904);  // construct object
// set desired attributes of nmeaSend...
nmeaSend.push(handle);
```

If the handle of the output stream is omitted, the push method will look for a single NMEA2000 stream and use that and remember it for the next push. So it is only necessary to specify this if there is more than one NMEA2000 stream.

The descriptors

The plugin contains a library of descriptors copied from canboat project. The NMEA2000 constructor loads the appropriate descriptor. Should you wish to view the descriptor, it is available as the .descriptor. attribute. You could pretty-print it by

```
print(JSON.stringify(NMEA129029.descriptor, null, "\t"), "\n");
```

One of the descriptor's attributes is `.Complete`, which is true if is believed to be a complete description. You could write:

```
print("This descriptor is ",
NMEA129029.descriptor.Complete?"complete":"doubtful", "\n");
```

You can also load a descriptor directly from the library without using the NMEA2000 constructor:

```
descriptor = require("pgnDescriptors")(129029);
```

You can load an array of all descriptors with

```
descriptors = require("pgnDescriptors")();
```

You might want to decode using a custom descriptor. Perhaps you have a descriptor not in the plugin's library or have a corrected version of a descriptor. You can construct the NMEA2000 object using a custom descriptor in place of one from the library using one of these:

```
NMEAspecial = new Nmea2000(myDescriptor);
NMEAspecial = new Nmea2000(myDescriptor, message.data);
```

Descriptor ambiguity

Some PGNs are used differently by different manufacturers. For example PGN 130824 is used differently by Maretron for Annunciator and B&G for key-value data. Attempts to load PGN 130824 will throw an error citing this ambiguity.

However, you can load an array containing all definitions of a PGN using, e.g.:

```
descriptors = require("pgnDescriptors")(130824, {"options":"returnAll");
```

You could decide which of these is correct for your installation and construct using the relevant descriptor.

Should you have the complication of having equipment from both manufacturers, you would need to construct both NMEA2000 objects and examine the manufacturer code in the data to decide which object to use on a message by message basis. The manufacturer code can be decoded by

```
code = ((data[14] << 8) & (data[13])) >> 5;
```

See here to relate the code to the manufacturer.

PGN 129541 GPS Almanac Data

This descriptor includes several Greek characters, which the JavaScript engine on Windows cannot handle. It stops all the descriptors loading. This descriptor has, therefore, been removed from the built in ones and made available here.

If you want to decode this PGN and are not running on Windows, you can follow the instructions to load it as a custom descriptor.

Concerning the descriptors

The descriptors contain an array of fields, each describing an element of the pgn. Each field includes an attribute `Order` being the serial number of the field. Confusingly, in the canboat library, Order 1 is at index 0. To simply scripting and reduce confusion, the plugin inserts an element zero at index 0 so that Order 1 is at index 1 et seq.

If you display the descriptor stored in the constructed object, it has this extra element.

If you supply a custom descriptor, the plugin will insert this element if it is not already present.

Using the canboat analyser

If you have the canboat software installed on your computer, you can use it to decode an NMEA2000 message instead of the NMEA2000 object. The canboatAnalyzer function converts the message data from that provided by OpenCPN to the pseudo-Actisense format expected by canboat analyzer, invokes the analyser and returns an NMEA2000 object. This process is significantly slower than using the decoder built in to the NMEA2000 object but might be useful if you wish to compare results. Example:

```
canboatAnalyzer = require("canboatAnalyzer"); // load the function
analyzer = "/Users/Tony/OpenCPN_project/canboat/rel/
           darwin-arm64/analyzer"; //where to find the analyser program
//use the bash shell
myObject = canboatAnalyzer(test.data, "bash", analyzer);
```

The attribute values should be the same for the two methods but the object structure differs. See Appendix A for this.

19. Sockets

Scripts can send and receive text data over UDP sockets.

id = onSocketEvent(handler, port)

Creates a socket connection on the given port. When data is received, it is passed to the given handler. The call returns a callback id.

If you set OpenCPN to send NMEA0183 data out on UDP port 10000, the following script will print the sentences.

```
id = onSocketEvent(display, 10000);
function display(error, data){
      if (!error) print(data);
      }
```

Note that the data is printed as received and has not been through OpenCPN's filtering or checking.

count = socketSend(id, text [, port, address])

Given a socket id, sends out text to the port at the address. Returns a count of the characters sent. A maximum of 512 characters is allowed because of UDP constraints - but see `socketChunker` below.

If the address is omitted, it is broadcast.

If the port is also omitted, it is taken to be a reply to the last received data and is sent only to the sender. Data must, therefore, have previously been received on the socket.

Note that, were you to send to the same port as was set up to receive, you would receive the sent data twice.

Consider this illustration:

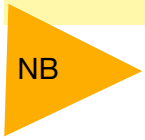
```
// Part A
id1 = onSocketEvent(report1, 10000);
back = "@";
function report1(error, data){
      print(data, "\n");
      back += "@";
      socketSend(id1, back);
      }

// Part B
id2 = onSocketEvent(report2, 20000);
out = "*";

function report2(error, data){
      print(data, "\n");
      out += "*";
      socketSend(id2, out);
      }

socketSend(id2, out, 10000);
```

Part A sets up a listener on port 10000 and each time data is received, sends a reply with an incrementing number of '@'.

Part B sets up a listener on port 20000 and, similarly, replies with an incrementing number of '*'. Part B kicks things off by sending an an initial '*'.

The two parts exchange continuously.

Both parts can be run in a single script but you could run each part in different consoles which need not be in the same OpenCPN instance. Part A must be started first so that it is listening when Part B starts things off.



If you broadcast from a socket to the same port that it is listening to, it will itself receive the sent message twice - once as loop-back as it is sent and once when it receives the broadcast.

Sending a reply by omitting both port and address avoids this because the reply is not broadcast. However, you cannot reply until a message has been received, thus recording its address.

In the illustration above, the initial single '*' is received twice.

You can avoid this problem if you know the address to which the message is to be sent - thus avoiding broadcasting the data.

This sockets support opens the way for scripts to communicate directly with OpenCPN or with scripts running on different instances of OpenCPN.

socketChunker

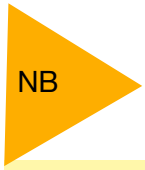
`socketSend` is restricted to sending strings of up to 512 characters - the maximum that can reasonably be sent over UDP.

The `socketChunker` object allows any size of object to be sent in chunks, which can be reassembled by the receiving `socketChunker` object. This only works between `socketChunker` objects and cannot therefore be used to send text to, e.g. , OpenCPN.

socketChunker handles a number of limitations and issues issues with UDP. See the following table.

You use the methods in socketChunker which calls the socket APIs as needed.

```
socketChunker = require("socketChunker");
chunker = new socketChunker(handleEvent, 1234); // instead of onSocketEvent
chunker.socketSend("This can be a looong message", 1234); // instead of
socketSend
function handleEvent(error, data){
      if (error) throw("socketChunker got error " + data);
      // do something with the data received
      }
```



socketChunker opens a socket using onSocketEvent. When a script terminates, any sockets will be closed. Should you want to destroy a socketChunker object or let it go out of scope, it is best to close the socket beforehand using the usual callback cancellation. For this, you need the socket id, which is available as the objects socketId attribute. Following the above code, you might have

```
onSocketEvent(chunker.socketId); // closes the socket using its id
delete chunker;                  // and then delete the chunker object
```

Comparison between onSocketEvent / socketSend and socketChunker

What	onSocketEvent / socketSend	socketChunker
Communicate with	Anyone, including OpenCPN	Other socketChunker(s) only
What can be sent	Strings only	Any JavaScript object - numbers, strings, arrays and objects such as waypoints
Maximum length	512 characters	Unlimited
Delivery sequence	No checking	Handled correctly
Transmission error checking	None	Uses SNV hash to check integrity
Duplicates	Broadcasting can cause receipt of own sends	Echoes ignored and hidden

20. Executing a terminal command

```
result = execute(command);
```

```
result = execute(command, env, errorOption);
```

The command is run as a sub-process and the script waits for the result. The result is a structure holding the output to `stdout` and `stderr`. Example:

```
result = execute("hostname");
print(result.stdout, "\n");// on my computer this prints its host name
```

A second argument `env` can be provided set the environment for the sub-process. The following illustrates all options:

```
env = {
      "PATH":"/bin:~/myPrograms",// where to search for programs
      "PWD":"~/ocpnProject,      // working directory for the sub-process
      "SHELL":"/bin/zsh"         // shell to use
      }
```

Some programs, such as the canboat analyzer, write non-error information to stderr. If you wish to see such output, it is available at `result.stderr`.

If a real error occurs, an error will be thrown. This would show you the error message but you would not get an opportunity to see what had been written to that point. You can override this by providing a third argument of `true`. In this case an error will not be thrown and the result will have an addition property `errorCode`, which will be

0	no error
-1	unable to launch process
+ve	error code

If the error code is positive, you can examine the output on `stdout` up to the error. `stderr` should include the real error message.

If you want to provide this third argument but not set the env options, you can use `null`.

```
result = execute("echo 'Hello world!'", null, true);
if (result.errorCode != 0) printOrange("after output of ", result.stdout,
      " got error ", result.stderr, "\n");
else print(result.stdout, "\n");
```

Warnings

The plugin waits for the other process to complete. You could start a long one. There is no timeout.

You could start a process that cannot complete. For example, it might try to read from the terminal input (stdin). Since there is no terminal it will wait indefinitely. There is no way to get out of this, other than to force quit OpenCPN.

You should not experiment with this while depending on OpenCPN to navigate!

Running a shell script

For non-trivial commands, such as a shell pipeline, it will be easier to use the `shell` function, which you can load if required. The following example imagines you want a list of the track files in a particular folder

```
shell = require("shell");
result = shell("cd ~/Tracks; ls | grep track");
trackFiles = result.stdout;// list of files with 'track' in their name
```

You can give the shell function optional arguments thus:

```
result = shell(pipeline, shell, errorOption);
```

`shell` is the name of the shell to use. The default is "bash".

`errorOption` is the same as for `execute` above.

21. JavaScript Plugin Tools

The tools window is accessed top right of any console or through the Preferences button in plugin's entry in the plugin manager.

The tools comprise six pages

Consoles

Allow creation of an additional console and renaming of an existing console.

A couple of user have reported they were unable to find a console. There is a button that makes all consoles visible.

Console options:

- Float windows on the OpenCPN frame (on by default). Mainly relevant to MacOS.
- Preserve the *Show consoles* toggle status in the toolbar (off by default)

Directory

Allows setting of the current directory.

NMEA

Allow allows you to simulate sending an NMEA message to the plugin for testing purposes.

Message

Allow allows you to simulate sending an OPenCPN message to the plugin for testing purposes.

Parking

Allows you to customise the parameters for parking consoles, including the park location.

Diagnostics

See the technical manual for this.

22. Updating Published Scripts

If you publish a script for others to use, you may encounter the issue of how to get an updates to the users if you develop it further or add fixes, etc. You can use the `checkForUpdates` function to propagate an updated script to users.

We assume here that the script has been made available on-line for downloading.

You need to create an additional JSON file contains the version details. This

Attribute	Purpose	Example
name	The name of the script	"PointUtility"
version	The version no released	1.4 or "1.4.1"
date	Date of release	"16 Jun 2024"
script	URL of the script. Can be omitted if it is supplied in the call to checkForUpdates	"https://raw.githubusercontent.com/antipole2/PointUtility/main/pointUtility.js"
	The following attributes are optional	
new	Short explanation of what has changed	"Reduces excessive checks if on-line"
pluginVersion	Minimum JavaScript plugin version required	3.1
comment	Any comment/reminder for this JSON file	"Don't forget to set same version number in the script too!"

Here is the actual JSON file from which the above examples are taken:

```
{     "name":"PointUtility",
      "version":1.4,
      "date":"16 Jun 2024",
      "script": "https://raw.githubusercontent.com/antipole2/PointUtility/main/
pointUtility.js",
      "new":"Reduces excessive checks if on-line",
      "pluginVersion": 3.1,
      "comment": "Don't forget to set same version number in the script too!"
}
```

The `checkForUpdates` function takes the following parameters:

1. The script name
2. The current script version
3. How often to check for updates in days
4. The location of the JSON file
5. The location of the script file. If this is omitted, the location in the JSON file will be used.

The function is loaded using `require()`. This is usually best to place near the beginning o the script. Example:

```
version = 1.3;
require("checkForUpdate")("PointUtility",  version,   5,
 "https://raw.githubusercontent.com/antipole2/PointUtility/main/version.JSON");
```

What checkForUpdates does

1. Checks if you have Internet access. If not, it does nothing.
2. Checks if the specified number of days has elapsed since the last check. If not goes no further.
3. It fetches the JSON file and checks whether it describes a later version.
4. If so, it gives the user a chance to save any modifications made to the existing script.
5. If no modifications need to be saved, it loads the new version of the script.

Notes

- `checkForUpdates` keeps track of the script name and when the last check was made using the location `_remember.versionControl`.
- If you are also using `_remember`, you need store your information in a different attribute.
- `_remember` is unique for each console. So if you load a script into a different or new console, it will check as soon as possible.
- The locations of the files are usually specified as a URL (starting `http:` or `https:`). Internally the plugin uses the same functionality as `readTextFile()` so you could specify a local file or use '??' to invoke a dialogue.

23. Tips

This section provides a few tips on working with the plugin.

Examining objects

During development of a script, it can be very helpful to examine a JavaScript object. You can do this by printing it out.

```
nav = OCPNgetNavigation();
print(nav, "\n");
```

The printed object will look something like:

```
{"fixTime":1672843366,"position"{"latitude":57.494,"longitude":-4.2344},"
SOG":null,"COG":null,"variation":0,"HDM":null,"HDT":null,"nSats":0}
```

For a large object, reading this can be tricky. In such a case it is better to use `JSON.stringify()` to transform it into a pretty JSON string. The following uses a tab character to indent the structure.

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

which would display the following equivalent, which is much easier to read.

```
{
      "fixTime": 1672843366,
      "position": {
            "latitude": 57.494,
            "longitude": -4.2344
      },
      "SOG": null,
      "COG": null,
      "variation": 0,
      "HDM": null,
      "HDT": null,
      "nSats": 0
}
```

See a JavaScript tutorial for all the capabilities of JSON transformations in JavaScript

24. Trouble-shooting character code issues

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 tools 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, “ºT\n”);
```

this works under MacOS but not under Windows. Instead you could use

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

25. 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.

There is also a library of contributed scripts available here.

A. Save script preferences for next script run

This script demonstrates one way of saving some preferences for use in a subsequent script or re-run of the same script.

The preferences are held in a structure prefs and saved to a text file in a JSON string. For this simple script to work, the preferences file must already exist and contain at least the empty JSON string { }.

The script sets up a call to saveConfig( ) on its exit in which the prefs structure is saved to the file as a JSON string.

When the script starts, it reads the file and parses it into a structure.

It then modifies some preferences.

```
prefFile = "/Tony/myFiles/prefsFile.txt";   // location of text file
onExit(saveConfig);

prefs = JSON.parse(readTextFile(prefFile));
print("Old prefs were: ", prefs);
prefs.iconName = "Circle";
prefs.scale = 5;

function saveConfig(){
          writeTextFile(JSON.stringify(prefs), prefFile, 1);
          print("Prefs saved\n");
          }
```

B. 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

When you push an NMEA sentence from within a function like this, it will itself be processed by the function. If that processing causes another matching sentence to be pushed, you could set up an infinite loop, which would cause OpenCPN to hang. In the above code, the pushed sentence is given a different sender, which is being filtered out of received sentences by OpenCPN.

C. 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

D. 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.

```
// Add hyperlinks to an existing waypoint with markName of 'lunch stop'
wpName = "lunch stop";
guids = OCPNgetWaypointGUIDs();
foundIt = false;
for (i = 0; i < guids.length; i++){
      // look for our waypoint
      lunchWaypoint = OCPNgetSingleWaypoint(guids[i]);
      if (lunchWaypoint.markName == wpName){
            foundIt = true;
            break;
            }
      }
if (!foundIt) 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"});
OCPNupdateSingleWaypoint(lunchWaypoint);    // update OpenCPN waypoint
```

Get code

E. 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

F. 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.

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.

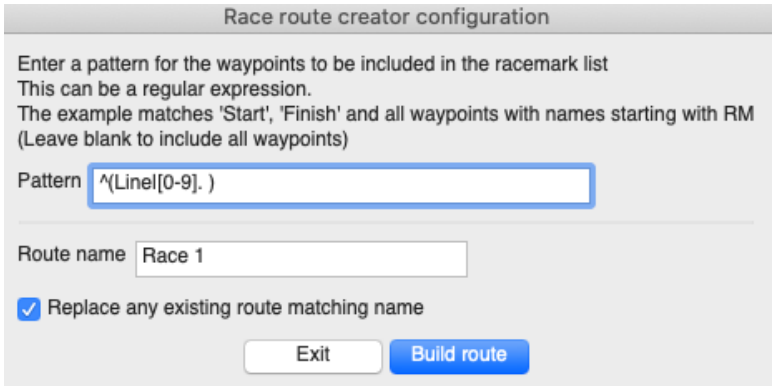
In this dialogue you select the course marks in order adding them to the course. You can indicate whether they are to be left to port or starboard.

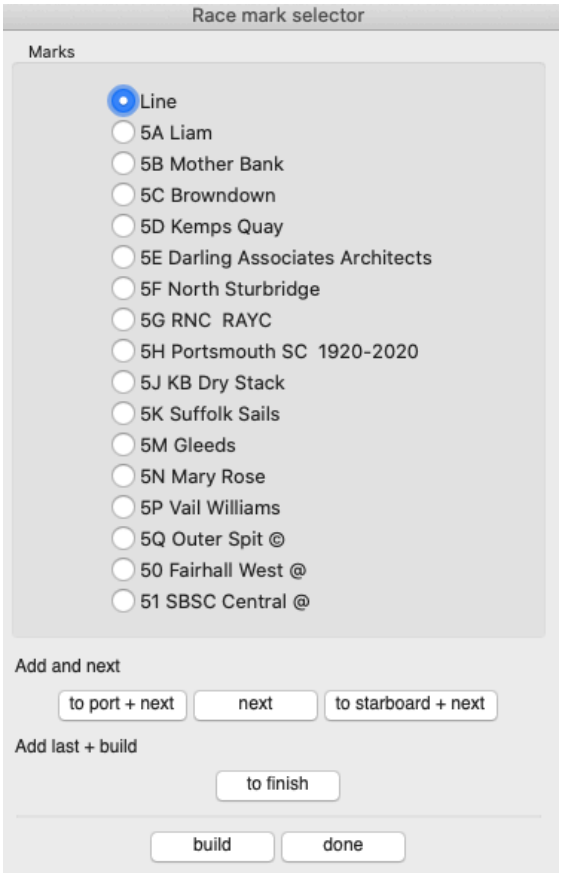
In this example, the finish is through the start line, so the final selection is *Line* and the button *to finish.*

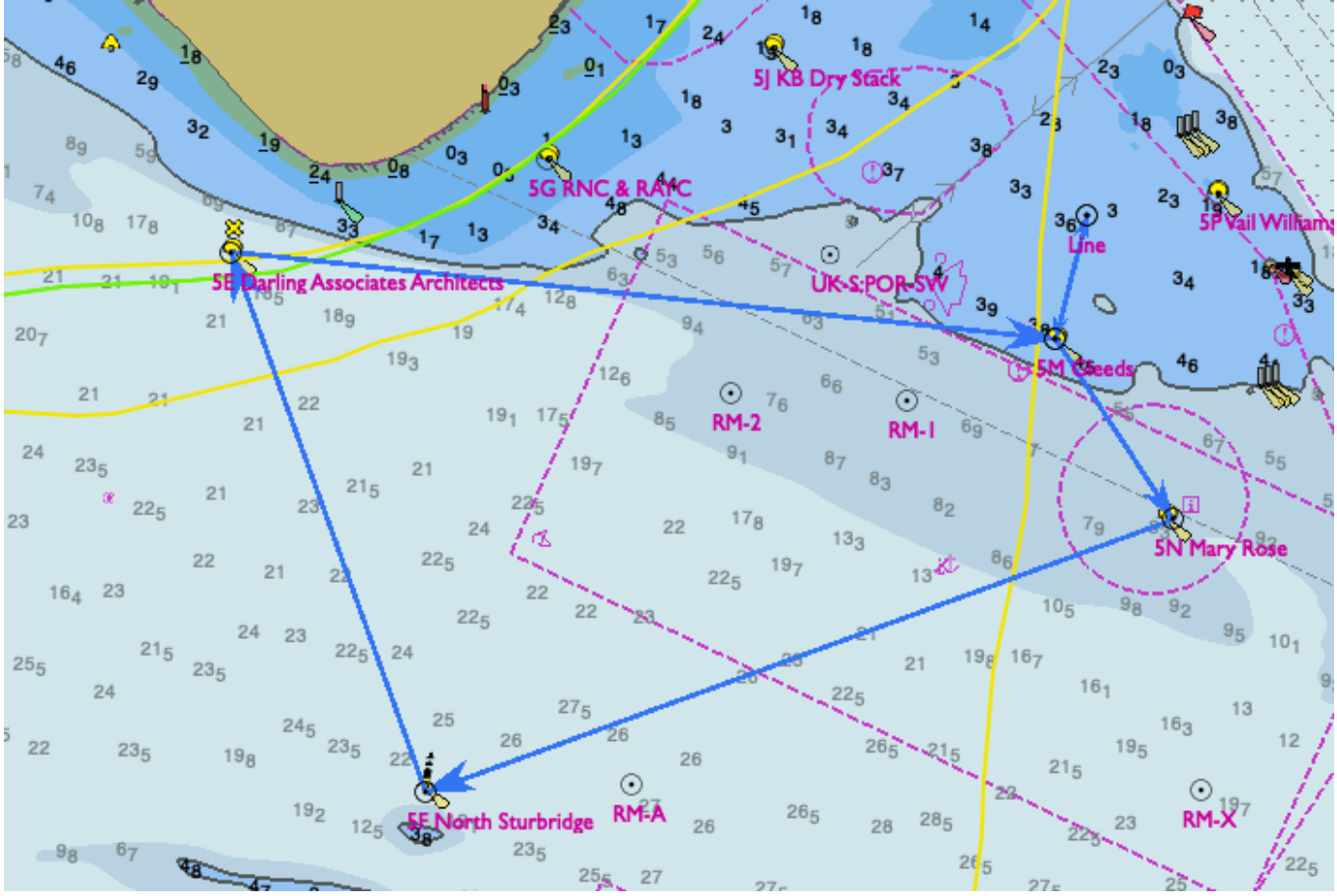
The script then builds the route in OpenCPN and also displays the route with the list of waypoints indicating the bearing and distance to each and which side to pass. The caption includes the course length.

Get code

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







G. 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

H. 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

Get code

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
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( )
0.4		• Position.NMEA precision increased from 3 to 5 decimal places

I. 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

J. Copy ship's formatted position

This script creates an extra context menu that copes the ship's latest position as a formatted string. This could be used to paste the position into a log or report.

```
Position = require("Position");
OCPNonContextMenu(doIt, "Copy position");
function doIt(){
      navData = OCPNgetNavigation();
      Shipos = new Position(navData.position);
      toClipboard(Shipos.formatted);
      OCPNonContextMenu(doIt, "Copy position");
      }
```

K. Anchor watch

This script creates an extra context menu to set or unset an anchor watch.

```
checkSeconds = 30;    // how often to check
drag = 0.001;         // drag limit in nm
OCPNonContextMenu(setAnchor, "Set Anchor");
var anchorPos;


function setAnchor(){
      navData = OCPNgetNavigation();
      anchorPos = navData.position;
      OCPNonContextMenu(unsetAnchor, "Unset anchor");
      onSeconds(checkAnchor, 30);
      alert("Anchor alarm set for ", drag, "nm");
      }

function checkAnchor(){
      nowPos = OCPNgetNavigation();
      if (OCPNgetVectorPP(anchorPos, newPos.position).distance > drag)
            OCPNsoundAlarm();
      onSeconds(checkAnchor, 30);
      }

function unsetAnchor(){
      onSeconds();     // cancel timer)
      alert(false);
      OCPNonContextMenu(setAnchor, "Set Anchor");
      }
```

Get Code

Appendix A. NMEA2000 and canboat analyser

The objects returned by these two methods of decoding NMEA message data yield the same data attribute values (with one exception noted below) but differ in the structure.

There follows a side-by-side example using PGN 129540.

1. The first seven attributes come from the Actisense header and are enumerated in a different order. This does not effect accessing them programatically.
2. OpenCPN leaves the Actisense header timestamp undefined. The canboat analyser requires a valid timestamp and so the `canboatAnalyzer` function inserts an arbitrary one. This is always the same and should be ignored.
3. After the Actisense header come the fields according to the descriptor. Canboat analyzser creates all these as attributes of an attribute `fields`. This complicates access to the data and NMEA2000 puts them directly in the NMEA2000 object.
4. For the attributes thereafter, canboat analyser uses the attribute description from the descriptor as the attribute name. These may include spaces, which makes for invalid attributes when accessing the data. You cannot write
   `sats = object.GNSS Sats in View;`
   NMEA2000 uses the attribute id so you can write
   `sats = object.gnssSatsInView;`
   The attribute ids start lower case.
5. Where an attribute has a lookup table for EnumValues, NMEA2000 returns a structure including both the value and the name.
6. Where the is a repeating field, such as for each satellite, canboat analyser creates an array `list[ ]`. If there were more than one set of repeating fields, there would be two lists with the same name. NMEA2000 creates an attribute for the count and then an array with the name of the list.
7. Many of the descriptor fields specify the units of the parameter. For example, it will tell you whether a temperature is in °C or °K. Navigation parameters such as headings or courses are in degrees. Satellite elevation and azimuth are in radians, as recorded in the descriptor. Canboat analyser arbitrarily converts these to degrees, despite what the descriptor says. This is the only such case I have found. NMEA2000 leaves them in radians.

NMEA2000


```
{
        "pgn": 129540,
        "id": "gnssSatsInView",
        "description": "GNSS Sats in View",
        "timestamp": "undefined",
        "prio": 3,
        "dst": 255,
        "src": 1,
        "sid": 13,
        "rangeResidualMode": "invalid",
        "satsInViewCount": 12,
        "satsInView": [
                {
                        "prn": 25,
                        "elevation": "1.3089",
                        "azimuth": "4.6949",
                        "snr": 31,
                        "rangeResiduals": 0,
                        "status":
                                {value:2, name:"Used"}
                },
                {
                        "prn": 28,
                        "elevation": "0.523500",
                        "azimuth": "5.288300",
                        "snr": 28
                        "rangeResiduals": 0,
                        "status":
                                {value:2, name:"Used"}
                },
```


canboatAnalyzer.js


```
{
        "timestamp":"2023-11-24-22:42:04.388",
        "prio": 3,
        "src": 1,
        "dst": 255,
        "pgn": 129540,
        "description": "GNSS Sats in View",
        "fields": {
                "SID": 13,
                "Sats in View": 12,
                "list": [
                        {
                                "PRN": 25,
                                "Elevation": 75,
                                "Azimuth": 269,
                                "SNR": 31,
                                "Range residuals": 0,
                                "Status": "Used"
                        },
                        {
                                "PRN": 28,
                                "Elevation": 30,
                                "Azimuth": 303,
                                "SNR": 28,
                                "Range residuals": 0,
                                "Status": "Used"
                        },
```

Appendix B. Updating built-in Scripts

The plugin includes several built-in script functions, loaded through the `require()` API. See Modules. These files are installed along with the plugin.

This appendix explains how you can update these scripts without waiting for a new plugin version.

Installing a new script version

Get the new or revised script into a console script window.

You can load it or copy it and paste it in. If it is available on-line, you can copy the URL to your clipboard and then use `Load` and select `URL on clipboard button`.

Now hold down the shift and option (Windows Alt) keys while clicking on Save as... This will open a `Save` dialogue with the build-in scripts folder as the location. Save the script, replacing any pre-existing version.

Patching a built-in script

To modify or patch an existing built-in script, click on `Load` and then, while holding down the shift and option (Windows Alt) keys, click on the `File...` button. This will open the built-in scripts folder allowing you to select the script to be changed.

Make the desired changes and then click on `Save` to update the script.

Checking the script

It is prudent check that all is well before saving a script to the built-in folder. You can attempt to run it. Pure functions should compile without error and give `Result: undefined`.

Plugin re-installations

The build in scripts are completely replaced during plugin installations. A later release of the plugin should contain the latest version of the scripts. If you make any other changes, these will be lost when the plugin is next re-installed.

Appendix C. 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
0.5		Functional changes: Waypoints • Enhanced: waypoint APIs have extended attributes allowing more control over waypoint details. • Additional attribute to distinguish between a free-standing waypoint and a routepoint • Additional attribute giving a count of the number of routes including this waypoint • New: get active waypoint GUID Routes • New: get all route GUIDs • New: get active route GUID Tracks (all new) • get all track GUIDS • get/update/delete track • New: get active leg information • New: printUnderlined Behind the scenes The plugin options have been moved within the opencpn.ini file from [Settings] to [Plugins]. The plugin will move settings to the new location as required. Numerous changes for the move from wxWidgetsv512 to v515
0.6		Functional changes Script and output panes Now soft-fold text at window boundary Dialogues Text field styles: If the font is set to "monospace" a monospaced font is used.
1.0		Re-issue to accompany v1.0

Version	Date	
1.1		Output pane • The output pane is now scrolled, so that the last output is always visible. • The output length is now limited to 100,000 characters to avoid a rogue script exhausting memory. If the output exceeds this, text is deleted from the top of the pane and a message inserted to indicate that this has happened. Fixes 1. Sending a message to OCPN_DRAW_PI was causing a crash. Because of the way OD processes messages, this cannot be supported. It will now throw an error. 2. A check is now made to avoid a called back function being invoked while any other JavaScript code is being executed. This is generalised protection against situations such as 1 above. 3. Some dialogue elements require or accept a list of values. An empty list could lead to a crash. Now an error is thrown. 4. Dialogue multi-line text fields now soft-wrap so long lines are readable. 5. When adding or updating a route, a routepoint without a GUID was not being allocated a new GUID. Now it is. 6. onExit( ) was not being lexed blue in the script pane. User Guide • Error in documentation of track structure corrected. • Added that with functions such as `consoleClearOutput()`, if the console name is omitted, it performs the action on the script's own console.

Version	Date	
2.0	18 Jan 2023 (	Load script The plugin now keeps a list of the ten most recently accessed script files. Load now displays a dialogue in which you can select a file from the lists or you can choose `Other...` which opens a file selection dialogue. You can also add a recent file to a list of favourites, which will always be offered, or you can remove a favourite. Consoles ☆The script and output panes are now more flexible and better optimise their space, including when resized. ☆The splitter sash between the script and output pane now shows adjustments while being moved. ☆Consoles can now be reduced to a minimal size just showing the console name. ☆ Consoles can be parked in the frame top bar using the Park button. A script can park its own console or another console. ☆The parking page of the tools allows you to set your own parameters for console parking, including the location of the park. ☆The consoles page of the tools now allows you to change the name of a console. Extended APIs ☆`readTextFile(fileString)` – fileString can now be a URL and text will be read from that location, if OpenCPN is on-line. New APIs ☆ `onContextMenu()` - create context menu item and handle with a function ☆ `consoleName()` - set the console name from a script ☆ `consolePark()` - park this or another console ☆ `messageBox()` - display message ☆ `OCPNsoundAlarm()` - sound alarm Other ☆ JavaScript engine updated from v2.5.0 to 2.7.0. Performance improvements and bug fixes only. ☆ OCPNpushNMEA( ) now checks the sentence starts "$....," ☆ Extensive rationalisation of plugin code, especially regularising processing after execution of some JavaScript. When an onExit script is not called has changed and is now documented.

Version	Date	
2.1	14 Apr 2023	Console Parking Consoles are now parked by screen position rather than by the OpenCPN frame. Getting GUIDs You can now select whether to get GUIDs for ordinary objects, objects in a layer or both.. Routes These now include the `description` and the `isVisible` attributes Windows scaled displays Supported Console options Options to choose *Float on Top* and to preserve toolbar toggle status

Version	Date	
3.0	28 Mar 2024	Clipboard Added toClipboard and fromClipboard functions Drivers Support for accessing and using the new input/output driver handles Navigation data Added `OCPNonNavigation` & `OCPNonAllNavigation` NMEA0183 OCPNonNMEA ➙ OCPNonNMEA0183 NMEA0183 data can now be written via a specific driver handle OCPNonNMEAsentence now supports receipt by specific NMEA sentence type NMEA2000 Added `OCPNonNMEA2000` & `OCPNonAllNMEA2000` Built in scripts gain OCPN2000, pgnDescriptors & canboatAnalyzer objects. Enduring callbacks Many callback APIs now have an `All` variant that invokes the handler function repeatedly rather than once only. Timers Internally, the timer functionality has been completely rewritten to be more efficient and accurate. You can now specify timer intervals to less than 1 second (0.5, 2.5 etc.). You can now cancel individual timers as well as all timers. You can set a repeating timer. Running other processes Added execute and shell function APIs File handling Added built in constructor for the File object that allows more flexible file handling. _remember Added a variable that endures between script runs Parking Parked consoles now remember where they were before being parked and will re-park in their own parking space. The close button can be used to unpark a console. Parking spaces are adjusted for console name changes and compacted when a console is deleted. onCloseButton( ) You can set a function to be called when the close button is clicked. OCPNsoundAlarm( ) You can now specify a sound file to be played. OCPNgetCanvasView( ) Fetches certain attributes of the current view port

Version	Date	
		cleanString( ) text filter to convert non-ASCII characters to nearest ASCII equivalent. NMEA0183checksum(sentence( ) Calculates the NMEA0183 sentence checksum. Load You can now load a script from a URL on the clipboard. Position constructor This can now parse a wide range of text formatted positions in degrees, degrees & minutes or degrees minutes & seconds.

3.1	2024	New
		`OCPNonAIS()` & `OCPNonAllAIS()` APIs added
		`OCPNisOnline() API added`
		`keyboardState() API added`
		require("pluginVersion") added to check the plugin version
		require("checkForUpdate") added.
		Shift key down while unpacking console vacates parking space.
		Tools->Consoles has gained a button to find all consoles
		
		Fixes
		In a script-created dialogue, hLine produced a line so thin it was not visible. Now it is visible.
		Internally, a macro is now used to cope with degree symbols under Windows. The overhead of this is now avoided on other platforms.
		One build was missing from the on-line catalogue in plugins updater. Now all 15 included.

3.2	2025	New Support for OpenCPN notifications `Tools-> Diagnostics -> Dump` includes the wxWidgets version for both the the plugin compile-time and run-time. API to convert latitude or longitude to formatted string API to parse a latitude or longitude to a number Fixes OCPNgetPluginConfig( ) reports the run-time wxWidgets version rather than the plugin compile-time version. Other When Position constructor is given a position as a text string, it uses the built-in OpenCPN parser. `Position.formatted` now formats according to the user's choice of units The processing of receiving OpenCPN messages has been changed to use the more efficient shared listeners. You can now cancel a specific messages callback as well as all.
4.0	2026	See the Release Notice

C. 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
0.5	06 Dc 2021	To accompany plugin v0.5
0.6	23 Jan 2022	To accompany plugin v1.0
2.0	03 Jan 2023	To accompany plugin v2.0
2.1	14 April 2023	To accompany plugin v2.1
3.0	28 Mar 2024	To accompany plugin v3.0
3.0.6	11 Jun 2024	Corrections to accompany plugin v3.0.6
3.1	8 Jan 2025	Updates to accompany v3.1
4.0	31 Jan 2026	Updated to accompany v4.0