JavaScript Plugin

Tony Voss

Version 0.2 - plugin history here

Contents Page

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.

The basics

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

Click on the icon to open the plugin console. 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

which 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 link is given whereby you can access the code which you can copy.

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

- Comments
- JavaScript key words[1]
- Strings

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.

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

In this early release there is a test button **Test A**. Please ignore it.

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

JavaScript and the embedded engine

A useful guide/tutorial on JavaScript can be found here. The engine fully supports ECMAScript E5.1 with partial support for E6 and E7. (See the Duktape page here for details, but note that not all features discussed on that site are available.)

Note that the embedded engine does not support:
- for/of loops
- classes
- arrow functions

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

JavaScript plugin extensions

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

I have implemented a few extensions to provide some output capability.

`print(arg1, arg2…)`

The print function displays a series of arguments in the results pane. Each argument can be of type *string*, a *number* or *boolean* value. Example:

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

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

`alert(arg1, arg2…)`

This is similar to `print` but the output is displayed in an alert box. The final newline is less necessary here. While the box is displayed, other screen functions are not available.

An argument cannot be an array or object. To display either of these, first turn it into a JSON string.

---

[1] The following JavaScript keywords are coloured but not supported and should be avoided (apart from inside strings) as they may cause compile errors or other untoward effects:
`abstract byte char class debugger double enum export extends final finally float goto implements import instanceof int interface of let long native short static super synchronized throws transient volatile with yield`

```
var boats = ["Yacht", "Dinghy", "Tender"];
print(boats, "\n");    // this will fail as boats is an array
print(JSON.stringify(boats), "\n");  // prints the JSON string of boats
```

readTextFile(fileString)

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

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

require(moduleName)

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

Understanding the result

After a script has run, the *result* is displayed in the output window after any other output, such as from print statements. The result is the result of the last executed statement, so for

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

This will display "Hi there!" And the result is undefined as the print function does not run a result.

In this example

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

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

If there is no statement returning a result, it is undefined, which is very often the case.

OpenCPN APIs

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

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

Basic APIs and event handling

Often it is necessary to set up a response to an OpenCPN event. These functions set up a callback to a function you supply and have names starting with *OCPNon.* The first argument is the name of the function to be called on the event. This sets up one call-back only. If you want the function to be called repeatedly, it needs set up the next call within it. When the OCPNon function is executed, a check is made that the nominated function exists within your code. Usually any error will be reported on compilation. However, where a call is made within a called-back function, the error can only be discovered at that time.

Herewith the currently implemented APIs.

OCPNpushNMEA(sentence)

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

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

OCPNgetMessageNames()

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

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

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

OCPNsendMessage(messageName[, message])

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

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

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

OCPNonSeconds(functionName, seconds[, parameter])

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

```
OCPNonSeconds(timesUp, 15, "15 seconds gone");

function timesUp(what){
      print(what, "\n");
      }
```

This would display the message `15 seconds gone` after 15 seconds.

Unlike other call-backs, you may set up any number of timed call-backs to different functions or the same function but each call-back is fulfilled only once. If multiple call-backs are due at the same time, they are fulfilled in reverse order with the most recently set up obeyed first.

OCPNonNMEAsentence(functionName)

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

```
OCPNonNMEAsentence(processNMEA);

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

OCPNonMessageName(functionName, messageName)

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

```
routeGUID = "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 set up the call-back before sending the request to be sure the call-back is in place when the message arrives.

- If the route GUID does not exist, the print will fail, so I am using JavaScript's `try & catch` to handle this.

OCPNcancelAll()

Cancels all outstanding call-backs and cleans up.

Call-backs are obeyed only once and are, therefore, self-cancelling unless renewed. But sometimes a call-back may never be fulfilled. This function cancels any outstanding call-backs and cleans up ensuring the Stop button returns to Run.

OCPNgetNavigation()

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

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

Example use:

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

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

OCPNgetNavigationK();

If you want to explore this, use `JSON.stringify()` to see the structure.

OCPNgetARPgpx()

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

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

Get code

OCPNgetNewGUID()

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

OCPNgetWaypointGUIDs()

Returns an array of the waypoint GUIDs. Example:

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

APIs for waypoint and route handling

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

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

OCPNgetSingleWaypoint(GUID)

Returns a waypoint object for the given GUID. Returns `false` if the waypoint does not exist. Example:

```
myWaypoint = OCPNgetSingleWaypoint("137eecdd-
e3e0-4eea-9d72-6cec0e500dbe");
if (!myWaypoint){
      print("No waypoint with that GUID\n");
      }
else{
      print("Waypoint name is ", myWaypoint.markName, "\n");
      }
```

Get code

OCPNdeleteSingleWaypoint(GUID)

Deletes a single waypoint, given the GUID.

Returns true if the waypoint is found and deleted, else `false`. Example:

```
if (OCPNdeleteSingleWaypoint("6aaded39-8163-43ff-9b6d-13ad729c7bb1")){
      print("Waypoint deleted\n");
      }
else print("Waypoint not found\n");
```

GUID = OCPNaddSingleWaypoint(waypoint)

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

If `waypoint.GUID` contains a GUID, that will be used. Otherwise a new GUID will be obtained for you. This function returned `the` GUID used if successful and `false` if it fails, such as when the supplied GUID is already in use. If you do not supply a GUID, you need to save the returned one if needed.

Example:

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

Get code

OCPNupdateSingleWaypoint(waypoint)

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

This function returned `true` if successful and `false` if it fails, such as when there is no existing waypoint with the given GUID.

OCPNgetRoute(GUID)

Returns a route object for the given GUID, complete with an array of the waypoints. Returns `false` if the route does not exist. Example:

```
myRoute = OCPNgetroute("137eecdd-e3e0-4eea-9d72-6cec0e500abc");
if (! myRoute){
      print("No route with that GUID\n");
      }
else{
      print("Route name is ", myRoute.name, "\n");
      }
```

OCPNdeleteRoute(GUID)

Deletes a route, given the GUID.

Returns true if the route is found and deleted, else `false`. Example:

```
if (OCPNdeleteRoute("6aaded39-8163-43ff-9b6d-13ad729c7abc")){
      print("Route deleted\n");
      }
else print("Route not found\n");
```

GUID = OCPNaddRoute(route)

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

If `route.GUID` contains a GUID, that will be used. Otherwise a new GUID will be obtained for you. This function returned `the` GUID used if successful and `false` if it fails, such as when the supplied GUID is already in use. If you do not supply a GUID, you need to save the returned one if needed.

Example:

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

OCPNupdateSingleRoute(route)

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

This function returned `true` if successful and `false` if it fails, such as when there is no existing route with the given GUID.

OCPNrefreshCanvas()

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

Objects and methods

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

You use the `require` function to load the constructor from the built-in library. You then need to construct one or more objects using the `new` statement, as shown in the following examples.

Position(lat, lon)

This constructs a `position` object as follows:

To load constructor	require("Position")	Note: constructor can take optional latitude, &longitude myposition = new Position(60, -1.5);
Attributes	.latitude	latitude in degrees
	.longitude	longitude in degrees
	.fixTime	time of position fix if recorded, else 0
Properties	.formatted	returns the position formatted for the human eye
	.nmea	returns the position formatted as used in NMEA sentences. You need to add the comma before and after, if required.
Methods	.NMEAdeode(sentence, n)	decodes the NMEA sentence and sets the position to the nth position in the sentence
	.latest()	Sets the position to the latest position available from OpenCPN and .time 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.NMEA, "\n");          // displays 5830.000,N,0030.000,E
```

Example 2: Decode an NMEA string and print the second position for the human eye:

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

`Waypoint()`

This constructs a `waypoint` object as follows:

Attributes	.description	The text to be displayed
	.link	The URL to which the text is to be linked

To load constructor	require("Waypoint")	
Attributes	.postion	As described for Position
	.GUID	
	.markName	
	.iconName	
	.isVisible	`true` if the mark is visibly displayed, else `false`
	.creationDateTime	A timestamp from when the waypoint was first created in OpenCPN recorded as seconds since 1st January 1970.
	.description	Free text description
	.hyperlinkList	Array of hyperlinks (see Hyperlinks)
Methods	.add(GUID)	Adds the waypoint into OpenCPN using the optional GUID, which must not already exist. If GUID is omitted, a new GUID will be obtained. Returns the GUID if successful, else false (GUID already exists). You must save the GUID if needed. If .creationDateTime is undefined, it is set to the present time. Bug: as of OpenCPN v5.2, the .creationDateTime attribute is ignored when adding a waypoint.
	.get(GUID)	Gets the waypoint from OpenCPN and sets the object to it. If GUID is supplied, that is the waypoint loaded. If GUID is omitted, the GUID in `waypoint.GUID` is used. Returns the GUID if successful, else false (no waypoint with the GUID exists). 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. Returns `true` if successful else `false`, such as when there is no matching GUID.
	.delete(GUID)	Deletes the waypoint in OpenCPN with GUID. If GUID is omitted, uses the GUID in `waypoint.GUID`. Returns `true` if successful, else false (waypoint with GUID does not exist).

Route()

This constructs a `route` object as follows:

To load constructor	require("Route")	
Attributes	.name	The route name
	.GUID	
	.from	The from text
	.to	The to text
	.waypoints	An array of the waypoints in the route, each being a waypoint object.
Methods	.add(GUID)	Adds the route into OpenCPN using the optional GUID, which must not already exist. If GUID is omitted, a new GUID will be obtained. Returns the GUID if successful, else false (GUID already exists). You must save the GUID if needed.
	.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 false (no waypoint with the GUID exists). 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. Returns `true` if successful else `false`, such as when there is no matching GUID.
	.delete(GUID)	Deletes the route in OpenCPN with GUID. If GUID is omitted, uses the GUID in `route.` Returns `true` if successful, else false (route with GUID does not exist).
	.purgeWaypoints()	Deletes all waypoints within the route object, including the waypoint's hyperlinks.

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);
```

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

Modules

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

You can also load code from your own file space using the `require` function. If the argument is a simple name without a suffix or file path separator, `require` looks in the library included with the plugin. Otherwise it looks for a matching file relative to your home directory. You cannot use an absolute path starting with '/' but you can use `..` notation to move up from your home directory. (This is true for MacOS and Unix-based systems - others to be confirmed.)

Loading your own functions

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

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

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

Writing and loading your own object constructors

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

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

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

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

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

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.

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(sentence){
      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 have var to insert
                  {
                  splut[10] = vardegs; splut[11] = varEW;
                  splut[0] = "$JSRMC";
                  result = splut.join(",");
                  OCPNpushNMEA(result);
                  }
            }
      OCPNonNMEAsentence(processNMEA);
      };
```

Getcode

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

Locate and edit waypoint, inserting hyperlinks

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

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

Get code

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

Diagnostic and confidence tester

This script tests the APIs and what OCPN functioning it can and also tests the included object constructors.

This is an evolving script which will develop with time as the plugin and OpenCPN evolve.

It can also be used to check the functioning of your build.

The script is thorough and the user is not expected to understand it all.

Download tester

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