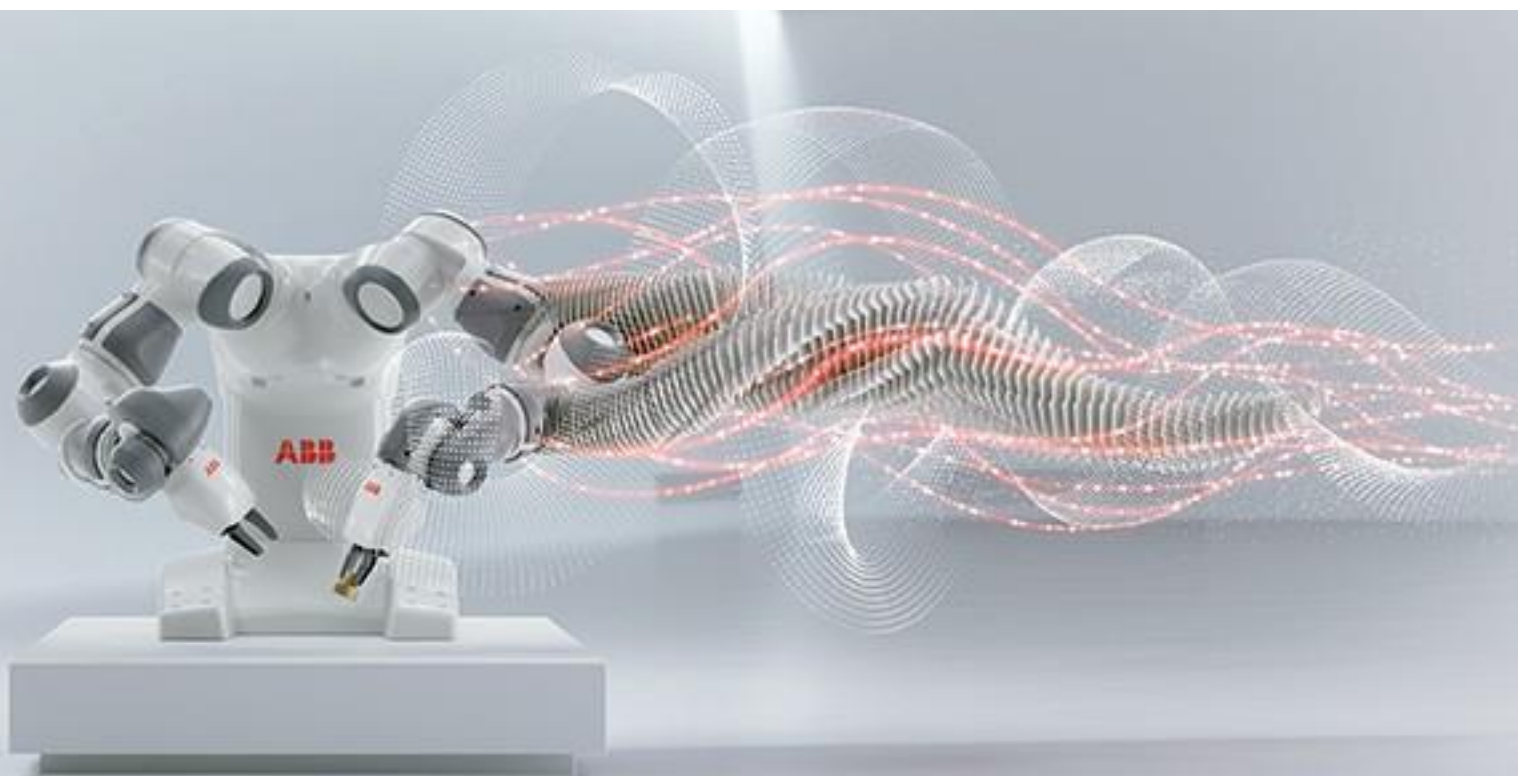


ROBOTICS

Application manual

OmniCore App SDK and AppMaker 1.4.1



This page is intentionally left blank

Application manual
OmniCore App SDK and AppMaker 1.4.1
Document revision: A, en

The information in this manual is subject to change without notice and should not be construed as a commitment by ABB. ABB assumes no responsibility for any errors that may appear in this manual.

Except as may be expressly stated anywhere in this manual, nothing herein shall be construed as any kind of guarantee or warranty by ABB for losses, damages to persons or property, fitness for a specific purpose or the like.

In no event shall ABB be liable for incidental or consequential damages arising from use of this manual and products described herein.

This manual and parts thereof must not be reproduced or copied without ABB's written permission.

Additional copies of this manual may be obtained from ABB.

The original language for this publication is English. Any other languages that are supplied have been translated from English.

© Copyright 2020-2024 ABB. All rights reserved.

ABB AB, Robotics

Robotics and Discrete Automation

Hydrovägen 10

SE-721 68 Västerås

Sweden

Table of contents

1	About This Manual	7
2	Introduction	8
3	Setting Up	10
4	Robot Web Services Client Library	14
4.1	App.Activation Namespace.....	16
4.2	RWS.Controller Namespace.....	18
4.2.1	Monitor Object.....	25
4.3	RWS.Rapid Namespace.....	28
4.3.1	Monitor Object.....	37
4.3.2	Task Object.....	40
4.3.3	ServiceRoutine Object.....	45
4.3.4	Module Object.....	46
4.3.5	Data Object.....	47
4.4	RWS.IO Namespace.....	57
4.4.1	Network Object.....	59
4.4.2	Device Object.....	60
4.4.3	Signal Object.....	62
4.5	RWS.CFG Namespace.....	66
4.5.1	Domain Object.....	71
4.5.2	Type Object.....	74
4.5.3	Instance Object.....	76
4.6	RWS.FileSystem Namespace.....	78
4.6.1	Directory Object.....	80
4.6.2	File Object.....	83
4.7	RWS.Elog Namespace.....	86
4.7.1	Event Object.....	88
4.7.2	Domain Object.....	89
4.8	RWS.UAS Namespace.....	92
4.9	RWS.Mastership Namespace.....	94
5	Components Library	96
5.1	Introduction.....	96
5.2	Common Resources.....	97
5.3	Standard Colors.....	98

1 About This Manual

5.4 Button.....	100
5.5 Checkbox	102
5.6 Dropdown.....	104
5.7 Input.....	108
5.8 Radio Button	111
5.9 Switch Button	113
5.10 Toggle	115
5.11 Slider.....	119
5.12 Context Menu.....	123
5.13 Digital Indicator.....	128
5.14 Analog Level Meter.....	130
5.15 Line Chart.....	132
5.16 Pie Chart	140
5.17 Menu	147
5.18 Fold-in	154
5.19 Popup Dialog.....	156
5.20 File System Dialog.....	161
5.21 Tab Container.....	164
5.22 Hamburger Menu.....	169
5.23 On-screen Keyboard.....	173
5.24 Console Log Overlay	176
6 AppMaker	179
6.1 Introduction	179
6.2 Installation	180
6.3 Accessing AppMaker	180
6.4 Workflow	181
6.5 Using The Visual Editor	183
6.6 Advanced - Adding Code	186

1 About This Manual

About this manual

This manual describes how to build and deploy custom OmniCore FlexPendant user interfaces. Custom user interfaces are developed using common web technology, i.e. HTML, CSS and JavaScript.

This manual also describes how to connect your custom user interface to the OmniCore controller using Robot Web Services (RWS) through a JavaScript RWS client library supplied by ABB.

There is also a components library for creating “widgets” (e.g., buttons, drop-downs, switches, ...) for a look-and-feel that is similar in appearance to what is used in the native FlexPendant shell. This library is also documented in this manual.

Finally, the manual covers the basics of using *AppMaker*, an optional visual editing tool for quickly designing and generating simple user interfaces.

Who should read this manual?

This manual is intended for:

- System integrators
- Function package developers
- Advanced end-customers

Prerequisites

The reader should have a basic knowledge of:

- General web development (HTML, CSS, JavaScript)
- The OmniCore controller

References

References	Document ID
-	-

2 Introduction

General information

It is common for various robot cell function packages and solutions to add their own user interfaces to the robot controller's teach pendant. ABB's previous-generation IRC5 controller had a C#/.NET SDK for building such user interfaces. The OmniCore FlexPendant takes a step forward and introduces the possibility to create user interfaces in the form of web applications, using common web technologies (HTML, CSS, JavaScript).

In practice, this means that each custom user interface on OmniCore FlexPendant is a web page/application running in an embedded browser within the FlexPendant shell.

Since the embedded browser is a general web browser, there are few limitations on third party frameworks or libraries used for developing the application. Since most FlexPendant applications are relatively small and simple, some application developers prefer to use plain HTML/CSS/JavaScript – which has grown quite powerful over the years. Others may opt to use one or more of the many tools available from the web development community.

It is beyond the scope of this document to describe general web development. It is assumed that the reader has knowledge in these areas. If not, it is recommended to start by going through one of the many introductions available from various sources on the Internet.

RobotWare support

Support for web-based user interfaces and OmniCore App SDK was added in RobotWare 7.1 and the corresponding FlexPendant software.

Browser engine

The embedded browser is using the legacy pre-Chromium Microsoft Edge engine, version 18.19044.

The engine may be replaced in future versions of the FlexPendant software.

Robot Web Services Client Library

The application will most likely need to communicate with the controller. This is done by using the *Robot Web Services* REST API, commonly referred to as *RWS*, hosted on the controller.

While it is fully possible for the application to use the built-in capabilities of JavaScript to communicate directly with RWS, a client library is also available for more user-friendly access.

The Robot Web Services Client Library is distributed as JavaScript files that should be deployed together with the application itself. More on this in section 4.

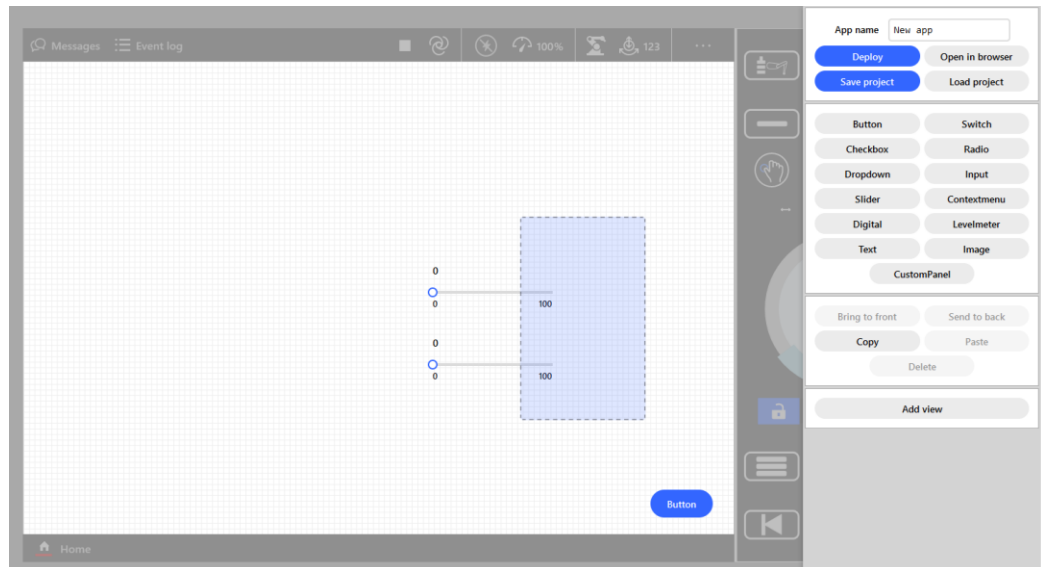
Components Library

The Components Library is a collection of re-usable interface components, such as buttons, input fields, menus, etc.

The Components Library is distributed as JavaScript/CSS files that should be deployed together with the application itself. More on this in section 5.

AppMaker

AppMaker is an optional visual editing tool for designing and generating OmniCore App SDK-based user interfaces.



AppMaker can be used for creating simple user interfaces, complete with bindings to various resources on the controller, such as RAPID or I/O.

Another possibility is to use AppMaker to quickly generate a prototype that can be extended with manually written code.

More on this in section 6.

3 Setting Up

Two ways forward

Setting up the structure for a new OmniCore App SDK-based app can be done manually, or by using the *AppMaker* visual editor. For more information on AppMaker, see section 6.

This section describes the minimal requirements for the anatomy of a app running on the OmniCore FlexPendant, and some other general information such as accessing the app from an external browser.

Application file structure

Each application needs to meet a few requirements in order to be properly installed. The source files for a very simple application named `MyApp` could look like this:

```
$HOME
└─ WebApps
    └─ MyApp
        ├── appinfo.xml
        ├── myapp.html
        └─ icon.png
```

Each application can be deployed in a separate subdirectory under `$HOME/WebApps/`. Hence, if another application is added, the file structure would be

```
$HOME
└─ WebApps
    └─ MyApp
        ├── appinfo.xml
        ├── myapp.html
        └─ icon.png
    └─ MyOtherApp
        ├── appinfo.xml
        ├── myotherapp.html
        └─ icon.png
```

The FlexPendant will automatically discover and add launchers for available applications, providing they are located in a correct directory path on the controller. See more below on how to package the application within a RobotWare Addin.

In the examples above, each application consists of three files.

The `myapp.html` file is the HTML page implementing the application. An application usually contains many more files than this file, it could be other HTML pages, CSS files, JavaScript files, or images.

If the RWS Library or the Components Library is used, they are placed here as well. More information about these files can be found in the relevant sections in this document.

Each application should also contain an icon image file that will be used for its launcher in the FlexPendant Applications menu. This image will be rendered at 100x100 pixels resolution and will automatically be scaled.

The `appinfo.xml` file is the application manifest file. For the simple application above, it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<WebApp>
    <name>MyWebApp</name>
    <icon>myicon.png</icon>
    <path>myappl.html</path>
</WebApp>
```

The `name` element is used to give the application a name, this name is used in the FlexPendant Application menu.

The `icon` element is used to refer to the image file that should be used for the launcher.

The `path` element is a reference to where the start page of the application is located.

Both the icon and start page paths are relative to the directory where the `appinfo.xml` file is located.

Distributing the app within a RobotWare Add-in

For advanced users, it is also possible to place a `WebApps` directory in an option directory of a RobotWare Add-In. This directory can host applications in the same way as when located under `$HOME`. I.e. `<option>/WebApps/<app>`.

Information about the general structure of RobotWare Addins is beyond the scope of this manual. However, to register an option directory, use the following line in the add-in's `install.cmd` file:

```
register -type option -description my_app -path $BOOTPATH
```

In this example, the FlexPendant will search in `$BOOTPATH/WebApps/` for app directories, provided the `WebApps` directory exists.

The web-root

The *web-root* is the base directory for the application when loading additional resources from the application. For each application, the web-root is the directory on the controller that contains the `appinfo.xml` file.

The HTML image tag

```
</img>
```

will load the SomePicture.png image from the root directory on the controller.

Using an external browser

It is possible to use an external browser (such as Chrome, Edge or Firefox) for connecting to a controller and running an application. This is especially useful during development, since most modern browsers have a plethora of powerful development tools.

To connect to a virtual controller on the same machine, assuming there is a web application at `$HOME/WebApps/MyApp/myapp.html`, use the following address:

```
https://127.0.0.1:80/fileservice/$home/WebApps/MyApp/myapp.html
```

The network port (80) is not guaranteed on a virtual controller, since the port may be in use by another virtual controller or other process on the same machine, or otherwise blocked by a firewall or operating system policy.

From RobotWare 7.3 and forward, if port 80 cannot be used for any reason, the virtual controller will instead try to find an available port from the following list:

- 5466
- 9403
- 9805
- 11622
- 19985
- 31015
- 34250
- 40129
- 45003

Since the virtual controller is using a self-signed certificate, the browser may issue a warning, and a permission to proceed anyway must be given.

The browser will ask for user credentials, which are the same as when logging into the system normally. Unless UAS (*User Authorization System*) has been configured, the default user is **"Default User"** and the corresponding password is **"robotics"**.

If using a real controller, replace `127.0.0.1` with the IP address of the controller. The port is **443**, which can be omitted since it is the default port for HTTPS communication.

Note that due to limitations in Robot Web Services and RobotWare, it is not supported to use several tabs in the same browser for connecting to the same controller.

Example WebApps

The OmniCore App SDK distribution contains a set of simple application examples.

One such example is the `components-example` application, which showcases how to use some components from the Components Library. Another example, `rws-example`, shows how to use the RWS Library to set up simple read and write communication with RAPID data and I/O signals, as well as use these values to update the user interface of an application.

Note that each example has a readme.txt file which contains setup instructions.

4 Robot Web Services Client Library

General Information

The Robot Web Services Library provides a JavaScript API to access the controller through use of Robot Web Services (RWS.) It uses pure JavaScript and does not require any knowledge of other frameworks. It does not preclude the use of other frameworks or techniques, that is left entirely up to the app developer.

The Robot Web Services Library uses only JavaScript functionality that is compatible with Microsoft's Edge browser as that is used by the WebView component on the FlexPendant. That is a limitation that also should be adhered to by users of the library.

The API uses JavaScript promises for handling asynchronous calls. It is strongly recommended that the user of the API knows how to best use the advantages of promises, however, most examples in this chapter use the `await` instruction to make it easier to understand the API.

Terminology

Throughout this chapter we will use the term namespace. In the Robot Web Services library, we build hierarchies by nesting objects. To visualize such hierarchies, we refer to them in the format; `RWS.Rapid`, this syntax we call namespaces. As a namespace refers to an object it may have methods as any other object in JavaScript.

RWS supports subscriptions to events that occur on the controller. Most subscriptions are related to a specific object, e.g. a IO signal or RAPID variable, but some are general and have no objects created. To facilitate subscriptions to such events there are special objects implemented in the namespaces. These objects are called Monitors and they are implemented specifically for the namespaces and thus have different behavior depending of the functionality of the subscriptions they support.

Installation

The Robot Web Services Library files should be placed in a subdirectory under the application web root (where the `appinfo.xml` file is located) named `rws-api`.

omnicore-rws.js

The omnicore-rws script is required and contains the RWS namespace. This is where all essential functionality regarding controller interaction via Robot Web Services resides.

To use the script, it must be imported to the JavaScript files it is being used in, as:

```
var script = document.createElement('script');
script.src = "rws-api/omnicore-rws.js";
document.head.appendChild(script);
```

or to the HTML files in the head, as;

```
<script src="rws-api/omnicore-rws.js"></script>
```

rapiddata-rws.js

This script contains rapid data conversion functionality. It is intended to be used internally by the library, and as such has no public methods documented in this manual. The script needs to be imported to the project in the same manner as the `omnicore-rws.js` file.

omnicore-app.js

The App script is optional. It contains the App namespace and is imported in the same way as the `omnicore-rws.js` file in the previous section, but with the source file changed to `omnicore-app.js`.

This script contains functionality that can improve the behavior and performance of the app. However, simple apps can skip using this functionality.

Return Values

Typically methods that will take time to execute, e.g. methods that communicate with the controller through RWS, will return a Promise. This allows the API user to execute them asynchronously, or synchronously by using `await`. It is strongly recommended that the API user knows how Promises in JavaScript work and can select the appropriate way to handle them. Running asynchronously will result in significantly improved UI performance.

When a Promise is resolved the returned data will be as defined for the specific method in this manual, however when a Promise is rejected a status object will be returned. This object will simply be referred to as Status in this document. The object will at least contain a message string but may also contain additional information (note that the user is responsible to check if the additional information exists before use.)

Structure of the status object:

```
status = {  
    (string) message,           descriptive text  
    ({} ) httpStatus,          HTTP status object, see description below  
    ({} ) controllerStatus     controller status object, see description below  
}
```

```
httpStatus = {  
    (number) code,              HTTP status, e.g. 400  
    (string) text               HTTP text, e.g. 'BadRequest'  
}
```

```
controllerStatus = {  
    (string) name,              status name, e.g.  
                                'SYS_CTRL_E_RAPID_SYNTAX_ERROR'  
    (string) code,              status code, e.g. '-1073442803'  
    (string) severity,          'Error' or 'Success'  
    (string) description        status description, e.g.  
                                'Operation not allowed due to syntax error(s).'
```

Example

```
var x = 42;
RWS.Rapid.setDataValue('T_ROB1', 'TestModule', 'TestData', x.toString())
    .catch(status => {
        let m = status.message;
        let h = '';
        if (status.hasOwnProperty('httpStatus') === true){
            let code = status.httpStatus.code;
            let text = status.httpStatus.text;
            h = `${code} ${text}`;
        }
        let c = '';
        if (status.hasOwnProperty('controllerStatus') === true){
            c = status.controllerStatus.description;
        }
        console.log(`${m}\n${h}\n${c}`);
    });
```

4.1 App.Activation Namespace

Description

The activation namespace, `App.Activation`, contains functionality to get notified by the OmniCore FlexPendant when the app is getting or losing focus.

The namespace does not contain any user callable methods, instead it relies on the user implementing methods that will be called by the framework when events occur.

User Implemented Methods

`appActivate`

Signature: `appActivate()`

The method will be called from an event handler in the `App.Activation` namespace when an activation event occurs. This method is expected to return a Promise that resolves if the execution of the method was successful and rejects otherwise. The contents of the promise are not relevant as the return value that is sent back to the FlexPendant will always be a boolean, true or false, depending on if the promise resolves or rejects.

This method can be used to enable user-controlled subscriptions and update the GUI if any state has changed while the app was deactivated.

Return value

A Promise with a boolean indicating success (true) or failure (false).

`appDeactivate`

Signature: `appDeactivate()`

The method will be called from an event handler in the `App.Activation` namespace when a deactivation event occurs. This method is expected to return a Promise that

resolves if the execution of the method was successful and rejects otherwise. The contents of the promise are not relevant as the return value sent back to the FlexPendant will always be a boolean, true or false, depending on if the promise resolves or rejects.

This method can be used to disable user-controlled subscriptions and persist any volatile state that otherwise can be lost if the FlexPendant shuts down.

Return value

A Promise with a boolean indicating success (true) or failure (false).

Example

This is an example of how the activation methods can be implemented in a HTML file.

```
<head>
...
    <script src="omnicore-app.js"></script>
    <script src="omnicore-rws.js"></script>
</head>
<body>
    <script>
        var userElog = null;
        window.addEventListener("load", () => {
            // Initialize the app
            userElog = RWS.Elog.getDomain(RWS.Elog.DomainId.user);
            userElog.addCallbackOnChanged((newValue) => {
                processUserEvent(newValue);
            });
        });

        var appActivate = function() {
            // Activate subscriptions
            try{
                userElog.subscribe();
                return Promise.resolve(true);
            } catch(error) {
                return Promise.reject(false);
            }
        }

        var appDeactivate = function() {
            // Deactivate subscriptions
            try{
                userElog.unsubscribe();
                return Promise.resolve(true);
            } catch(error) {
                return Promise.reject(false);
            }
        }
    </script>
</body>
```

```

...

</script>

...

</body>

```

4.2 RWS.Controller Namespace

Description

The controller namespace, `RWS.Controller`, contains basic functionality for manipulating the controller, e.g. get system information, create/restore backup, set motors on/off, restart controller, etc. To facilitate easy subscription to controller states there is a Monitor object.

Methods

`getMonitor`

Signature: `getMonitor(string resource)`

`resource` the resource to subscribe

Creates a new Monitor (see section 4.2.1) for subscribing to the provided resource. Valid resources are:

<code>'controller-state'</code>	changes to controller's state
<code>'operation-mode'</code>	changes to the controller's operation mode

The enum `RWS.Controller.MonitorResources` provides an easy way to safely use supported resource strings.

The returned Monitor has no callbacks set nor subscription enabled.

Return value

A Monitor object.

`isVirtualController`

Signature: `isVirtualController()`

Gets the controller's type. I.e., whether it is a virtual or physical controller.

Return value

A promise with a boolean, *true* for virtual controller, *false* for physical controller.

getControllerState

Signature: `getControllerState()`

Gets the controller's system state.

Return value

A promise with a string containing the controller state. Possible states are:

'initializing'	the controller is starting up
'motors_on'	motors on state
'motors_off'	motors off state
'guard_stop'	stopped due to safety
'emergency_stop'	emergency stop state
'emergency_stop_resetting'	emergency stop state resetting
'system_failure'	system failure state

Example

```
var state = await RWS.Controller.getControllerState();
if (state === 'emergency_stop' || state === 'system_failure'){
    mySetErrorState();
}
```

setMotorsState

Signature: `setMotorsState(string state)`

state the new state of the motors

Sets the controller's motor state. Possible states are:

'motors_on'	motors on state
'motors_off'	motors off state

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

getOperationMode

Signature: `getOperationMode()`

Gets the controller's operation mode.

Return value

A promise with a string containing the operation mode. Possible modes are:

'initializing'	controller is starting up, but not yet ready
----------------	--

'automatic_changing'	automatic mode requested
'manual_full_changing'	manual full speed mode requested
'manual_reduced'	manual reduced speed mode
'manual_full'	manual full speed mode
'automatic'	automatic mode
'undefined'	undefined

setOperationMode

Signature: `setOperationMode(string mode)`

mode the new operation mode

Sets the controller's operation mode. Possible modes are:

'automatic'	automatic mode
'manual'	manual mode
'manual_full'	manual full speed mode

NOTE! Automatic and manual full speed mode must be acknowledged by the user. This is done in a message box that is presented on the FlexPendant.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

restartController

Signature: `restartController(string mode)`

mode the restart mode, default is 'restart'

Restarts the controller. The parameter mode indicates what kind of restart is requested. Possible modes are:

'restart'	normal warm start
'shutdown'	shut down the controller
'boot_application'	start boot application
'reset_system'	reset system
'reset_rapid'	reset Rapid
'revert_to_auto'	revert to last auto-save

NOTE! An enum `'RWS.Controller.RestartModes'` with the valid values is provided for ease of use.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

getEnvironmentVariable

Signature: `getEnvironmentVariable(string variable)`

`variable` the environment variable to get

Gets any of the predefined system environment variables, e.g. "\$HOME", "\$TEMP", etc.

Return value

A promise with a string containing the contents of an environment variable.

getTime

Signature: `getTime()`

Gets the system time. The time is returned as a string formatted as 'year-month-day T hour:minute:second', e.g. '2020-06-11 T 15:04:33'.

Return value

A promise with a string containing the system time.

Example

```
var timeString = await RWS.Controller.getTime();
try {
    // Convert controller time string to JavaScript Date
    var temp = timeString.replace(/[T]/g, '-');
    temp = temp.replace(/[ ]/g, '');
    temp = temp.replace(/[:]/g, '-');
    var s = temp.split('-');
    var t = new Date(s[0], s[1] - 1, s[2], s[3], s[4], s[5]);
    var h = t.getHours();
    if (h >= 6 && h < 9) {
        console.log('Good morning!');
    }
} catch(error) {
    console.error(`not a valid date: ${error}`);
}
```

getTimezone

Signature: `getTimezone()`

Gets the system time zone.

NOTE! Not supported on virtual controller.

Return value

A promise with a string containing the system time zone.

getIdentity

Signature: `getIdentity()`

Gets the controller's identity (name.)

Return value

A promise with a string containing the controller's identity.

getNetworkSettings

Signature: `getNetworkSettings()`

Gets the controller's network settings.

NOTE! Not supported on virtual controller.

Return value

A promise with a list of objects containing the network settings for each adapter, defined as:

<code>(string) id</code>	adapter identification string
<code>(string) logicalName</code>	logical name of adapter
<code>(string) network</code>	network interface name
<code>(string) address</code>	IP address
<code>(string) mask</code>	network interface mask
<code>(string) primaryDNS</code>	primary DNS IP address
<code>(string) secondaryDNS</code>	secondary DNS IP address
<code>(string) DHCP</code>	use DHCP, valid values: 'enabled' or 'disabled'
<code>(string) gateway</code>	gateway IP address

getNetworkConnections

Signature: `getNetworkConnections()`

Gets the controller's network connections. This collection of information will provide information on which interfaces are currently available.

NOTE! Not supported on virtual controller.

Return value

A promise with a list of objects containing the network connections:

<code>(string) id</code>	adapter identification string
<code>(string) MACAddress</code>	adapter MAC address
<code>(boolean) connected</code>	adapter media status
<code>(boolean) enabled</code>	adapter execution status

(string) speed adapter network speed, e.g. '100 Mbps'

verifyOption

Signature: `verifyOption(string option)`

option the name of the option

Verifies if an option is available on the system.

NOTE! The option string is case sensitive.

Return value

A promise with a boolean.

createBackup

Signature: `createBackup(string path, number timeout)`

path the path where the backup will be created, including the name of the backup

timeout the maximum time (in seconds) to wait for completion, default is 60 s

Creates a backup of the system. The Promise returned by the function will settle when the backup is either completed or fails. As the process of creating a backup can take a long time, and the time is heavily dependent of the system, there is a possibility that the wait for completion is terminated before completion. By adjusting the timeout value; it is possible to ensure that the process gets a reasonable time to finish.

NOTE! Even if the wait for completion is aborted, the controller can still continue the process and the backup may or may eventually be completed.

Return value

A promise with the status code from the backup process or error status text.

verifyBackup

Signature: `verifyBackup(string path, {})`

path the path to the backup

{ } options for the verify process, containing:

(string) ignoreMismatches	ignore mismatches between system and backup, valid values: 'all', 'system-id', 'template-id' or 'none'
---------------------------	--

(boolean) includeControllerSettings	include controller settings, true or false
-------------------------------------	--

(boolean) includeSafetySettings	include safety settings, true or false
---------------------------------	--

(string) include	included components, 'cfg', 'modules' or 'all'
------------------	--

Verifies a backup. The return value is a Promise containing a list of errors. If the list is empty the verification did not find any faults and backup will be usable for the given properties.

There are two enum objects that can be used to set values to the `ignoreMismatches` and `includeParameters`; `BackupIgnoreMismatches` and `BackupInclude`. These enums can be used to avoid spelling errors and other use of illegal values.

Return value

A promise with a list of objects containing errors. An error object contains:

<code>(string) status</code>	the status of the verification, possible values:
	<code>'ok'</code>
	<code>'system_id_mismatch'</code>
	<code>'template_id_mismatch'</code>
	<code>'file_or_directory_missing'</code>
	<code>'cfg_file_corrupt'</code>
<code>(string) path</code>	optionally a path to the file or folder with issues

Example

```
await RWS.Controller.createBackup('$BACKUP/myBackup', 120);
var statuses = await RWS.Controller.verifyBackup('$BACKUP/myBackup');
if (statuses.length > 0){
    for (let i = 0; i < statuses.length; i++){
        if (statuses[i].path !== ''){
            var s = statuses[i].status;
            var p = statuses[i].path;
            console.log(`${s} in '${p}'`);
        } else {
            console.log(statuses[i].status);
        }
    }
} else {
    console.log('No problems.');
```

restoreBackup

Signature: `restoreBackup(string path, {})`

`path` the path to the backup
`{}` options for the restore process, containing:

<code>(string) ignoreMismatches</code>	ignore mismatches between system and backup, valid values: <code>'all'</code> , <code>'system-id'</code> , <code>'template-id'</code> or <code>'none'</code>
--	--

(boolean) <code>deleteDir</code>	indicates if backup directory should be deleted afterwards, true or false
(boolean) <code>includeSafetySettings</code>	include safety settings, true or false
(string) <code>include</code>	included components, 'cfg', 'modules' or 'all'

Restores a backup with the given settings.

There are two enum objects that can be used to set values to the `ignoreMismatches` and `include` parameters; `BackupIgnoreMismatches` and `BackupInclude`. These enums can be used to avoid spelling errors and other use of illegal values.

NOTE! This operation will restart the controller.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

saveDiagnostics

Signature: `saveDiagnostics(string path, number timeout)`

`path` the path where the diagnostics will be placed, including file name

`timeout` the maximum time (in seconds) to wait for completion, default is 60 s

Creates a system diagnostics log. The Promise returned by the function will settle when the process is either completed or fails. As the process of saving a diagnostics log can take a long time, and the time is heavily dependent of the system, there is a possibility that the wait for completion is terminated before completion. By adjusting the timeout value; it is possible to ensure that the process gets a reasonable time to finish.

NOTE! Even if the wait for completion is aborted, the controller can still continue the process and the diagnostics may or may not be completed.

NOTE! Not supported on virtual controller.

Return value

A promise with the status code from the process or an error status text.

4.2.1 Monitor Object

Description

The Monitor object is used to subscribe to resources that have no explicit object that can be used for subscription. The Monitor supports subscriptions to the controller's state and operation mode.

To create a Monitor object use the factory method `RWS.Controller.getMonitor(resource)`.

Each instance of the Monitor object can subscribe to one resource.

Resources

controller-state

Changes to the state of the controller, e.g. guard stop, motors on, etc. The event data is a string and can be one of the following values:

'initializing'	the robot is starting up
'motors_off'	standby, no power to motors
'motors_on'	ready to move
'guard_stop'	stopped due to safety run chain is open
'emergency_stop'	stopped due to emergency stop active
'emergency_stop_resetting'	emergency stop deactivated but not yet confirmed
'system_failure'	system failure state

operation-mode

Changes to the operation mode of the controller, e.g. auto, man, man full speed, etc. The event data is a string and can be one of the following values:

'initializing'	the robot is starting up
'automatic_changing'	automatic mode requested
'manual_full_changing'	manual full speed mode requested
'manual_reduced'	manual reduced speed mode
'manual_full'	manual full speed mode
'automatic'	automatic mode
'undefined'	undefined

Methods

getTitle

Signature: `getTitle()`

Gets the title (part of the URL) of the subscribed resource.

Return value

A string containing the title.

getResourceString

Signature: `getResourceString()`

Gets a URL that is unique for the resource and used for subscription.

Return value

A string containing a unique resource URL (as used by RWS.)

addCallbackOnChanged

Signature: `addCallbackOnChanged(function callback)`

callback a callback function to run when event occurs

Adds a callback function which will be called when the subscribed resource's state changes.

The callback routine can have one in-parameter, which is the new value of the subscribed resource. Contents of the value depends on the resource subscribed to.

Return value

No return value.

Example

```
var controllerMonitor = RWS.Controller.getMonitor('controller-state');
controllerMonitor.addCallbackOnChanged((eventData) => {
    console.log(`Controller state has changed to '${eventData}'.`);
});
```

onchanged

This is a function for internal use only. Do not call from external scripts.

subscribe

Signature: `subscribe(boolean raiseInitial = false)`

raiseInitial flag indicating whether an initial event is raised when subscription is registered

Starts the subscription to the resource defined when the Monitor object was created.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var controllerMonitor = RWS.Controller.getMonitor('operation-mode');
try {
    controllerMonitor.addCallbackOnChanged((eventData) => {
        console.log(`Operation mode changed to '${eventData}'.`);
    });
    await controllerMonitor.subscribe();
} catch(error) {
    console.error(`Subscription to 'operation-mode' failed. >>> ${error}`);
}
```

unsubscribe

Signature: `unsubscribe()`

Stops the subscription to the resource defined when the Monitor object was created.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.3 RWS.Rapid Namespace

Description

The Rapid namespace, `RWS.Rapid`, contains methods to manipulate Rapid data and execution of programs and service routines.

The namespace contains the following objects; Monitor, Task, Module and Data. These are described closer later in this section.

Methods

`getMonitor`

Signature: `getMonitor(string resource, string taskName)`

`resource` the resource to subscribe

`taskName` the name of the task

Creates a new Monitor (see section 4.3.1) for subscription on the provided resource. Valid resources are:

<code>'execution'</code>	changes to the Rapid program execution state
<code>'program-pointer'</code>	changes to the program pointer
<code>'motion-pointer'</code>	changes to the motion pointer
<code>'uiinstr'</code>	events from a Ui instruction such as TPWrite

NOTE! The parameter `taskName` is not valid for the 'execution' resource as it only applies for all tasks simultaneously.

NOTE! The returned Monitor has no callbacks set nor is the subscription enabled.

Return value

A Monitor object.

`getTasks`

Signature: `getTasks()`

Gets the available Rapid tasks on the controller.

Return value

A promise with a list of Task (see section 4.3.2) objects.

getTask

Signature: `getTask(string taskName)`

`taskName` the name of the task

Gets the Task object for Rapid task named `taskName`.

Return value

A promise with a Task object.

getProgramInfo

Signature: `getProgramInfo(string taskName)`

`taskName` the name of the task

Gets the program information for a task. The function returns an object with the information.

Return value

A promise with an object containing:

<code>(string) name</code>	program name
<code>(string) entrypoint</code>	name of function that is the program entry point

getModuleNames

Signature: `getModuleNames(string taskName)`

`taskName` the name of the task

Gets module names for a task. This will return both program modules and system modules.

Return value

A Promise with an object containing:

<code>([string]) programModules</code>	list of the names of all program modules
<code>([string]) systemModules</code>	list of the names of all system modules

Example

```
var modules = await RWS.Rapid.getModuleNames('T_ROB1');
if (modules.hasOwnProperty('programModules') === true){
    for (let i = 0; i < modules['programModules'].length; i++){
        console.log(modules['programModules'][i]);
    }
}
if (modules.hasOwnProperty('systemModules') === true){
    for (let i = 0; i < modules['systemModules'].length; i++){
        console.log(modules['systemModules'][i]);
    }
}
```

```
}
```

getModule

Signature: `getModule(string taskName, string moduleName)`

<code>taskName</code>	the name of the task
<code>moduleName</code>	the name of the module

Gets a module. This will retrieve the properties for the module from the controller and initialize the object.

Return value

A Promise with an initialized Module (see section 4.3.4) object.

getData

Signature: `getData(string taskName, string moduleName, string symbolName)`

<code>taskName</code>	the name of the task
<code>moduleName</code>	the name of the module
<code>symbolName</code>	the name of the symbol

Gets a Data object for symbol. This will retrieve the properties for the symbol from the controller and initialize the object.

Return value

A Promise with an initialized Data (see section 4.3.5) object.

setDataValue

Signature: `setDataValue(string taskName, string moduleName, string symbolName, string value, ...number indexes)`

<code>taskName</code>	the name of the task
<code>moduleName</code>	the name of the module
<code>symbolName</code>	the name of the symbol
<code>value</code>	the new value
<code>indexes</code>	an optional list of numbers that specifies an element in arrays and records

Sets the value of a symbol. This requires the value string to be on the RWS format, and as that can be a bit complicated it is recommended that the `setValue` function on an instance of `RWS.Rapid.Data` is used. However, as this call is slightly faster in some cases, it can be used when timing is essential.

By providing indexes, in the in-parameter `indexes`, a specific element in an array (multidimensional) or record can be set independent of the other elements in the structure. The indexes are 1-based as in Rapid.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var x = 42;
await RWS.Rapid.setDataValue('T_ROB1', 'TestModule', 'TestData', x.toString());
```

Example

Lets assume that a Record is declared as follows:

```
Record TestRecord
    num Num1;
    num Num2;
    string String1;
EndRecord

PERS TestRecord TestSimple := [0,1,"OK"];
```

Then its string component can be set as follows:

```
var str1 = "Testing raw write";
await RWS.Rapid.setDataValue('T_ROB1', 'TestModule', 'TestSimple', str1, 3);
```

Example

Lets assume that an Array is declared as follows:

```
PERS num TestArray25{2,5} := [[0,0,0,0,0], [0,0,0,0,0]];
```

Then the element in position [2,3] can be set as follows:

```
var e23 = '100';
await RWS.Rapid.setDataValue('T_ROB1', 'TestModule', 'TestArray25', e23, 2, 3);
```

getSharedData

Signature: getSharedData(string symbolName)

symbolName the name of the symbol

Gets a Data object for symbol that is declared in a module that is shared across all tasks. This will retrieve the properties for the symbol from the controller and initialize the object.

Return value

A Promise with an initialized Data object.

setSharedDataValue

Signature: setSharedDataValue(string symbolName, string value)

`symbolName` the name of the symbol

`value` the new value

Sets the value of a symbol that is declared in a module that is shared across all tasks. This requires the value string to be on the RWS format, and as that can be a bit complicated it is recommended that the `setValue` function on an instance of `RWS.Rapid.Data` is used instead.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

`getExecutionState`

Signature: `getExecutionState()`

Gets the current execution state of the controller.

Return value

A Promise with a string containing the execution state.

Possible values: 'running' or 'stopped'

`resetPP`

Signature: `resetPP()`

Moves the program pointer to Main (defined entrypoint) in all tasks.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

`startExecution`

Signature: `startExecution({})`

`{}` object containing;

(string) `regainMode`

regain mode, valid values: 'continue', 'regain', 'clear' or 'enter_consume'

(string) `executionMode`

execution mode, valid values: 'continue', 'step_in', 'step_over', 'step_out', 'step_backwards', 'step_to_last' or 'step_to_motion'

(string) `cycleMode`

cycle mode, valid values: 'forever', 'as_is' or 'once'

(string) `condition`

condition, valid values: 'none' or 'callchain'

(boolean) `stopAtBreakpoint`

stop at breakpoint

(boolean) enableByTSP all tasks according to task selection panel

Starts the Rapid execution with the settings given in the parameter object. All or any of the defined parameters can be supplied, if a value is omitted a default value will be used. The default values are:

```
regainMode = 'continue'
execMode = 'continue'
cycleMode = 'forever'
condition = 'none'
stopAtBreakpoint = true
enableByTSP = true
```

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
try {
  await RWS.Rapid.resetPP();
  await RWS.Controller.setMotorsState('motors_on');
  await RWS.Rapid.startExecution({
    regainMode: 'continue',
    executionMode: 'continue',
    cycleMode: 'once',
    condition: 'none',
    stopAtBreakpoint: false,
    enableByTSP: true
  });
} catch(error) {
  myErrorHandler(error);
}
```

stopExecution

Signature: stopExecution({})

{ } object containing

(string) stopMode	stop mode, valid values: 'cycle', 'instruction', 'stop' or 'quick_stop'
(string) useTSP	use task selection panel, valid values: 'normal' or 'all_tasks'

Stops the Rapid execution with the settings given in the parameter object. All or any of the defined parameters can be supplied, if a value is omitted a default value will be used. The default values are:

```
stopMode = 'stop'
useTSP = 'normal'
```

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var state = await RWS.Rapid.getExecutionState();
if (state === 'running'){
    await RWS.Rapid.stopExecution({
        stopMode: 'cycle',
        useTSP: 'normal'
    });
    await setTimeout(() => {}, 2000);
    state = await RWS.Rapid.getExecutionState();
    if (state === 'stopped'){
        console.log('We stopped Rapid.');
```

```
}
```

```
}
```

searchSymbols

Signature: searchSymbols({} properties, string dataType, string regexp)

properties	search properties (see getDefaultSearchProperties below)
dataType	the name of the data type to look for, can be empty
regexp	regular expression, can be empty

Searches for Rapid symbols as defined in the search parameters. The method can be used to search for variables, methods and data types.

If the data type is declared in a module that is installed or shared, only the data type's name should be provided in the `dataType` parameter, otherwise the full path is required. The flags "isInstalled" and "isShared" in the `properties` parameter must also be set accordingly.

Return value

A Promise with list of objects. Each object contains:

(string) name	name of the data symbol
([string]) scope	symbol scope
(string) symbolType	type of the symbol, e.g. 'pers'
(string) dataType	type of the data, e.g. 'num'

Example

```
var properties = RWS.Rapid.getDefaultSearchProperties();
properties.searchURL = 'RAPID/T_ROB1/TestModule';
properties.types = RWS.Rapid.SymbolTypes.rapidData;
```

```
properties.isInUse = false;

var hits = await RWS.Rapid.searchSymbols(properties);
if (hits.length > 0){
    for (let i = 0; i < hits.length; i++){
        if (hits[i].name === 'TestData'){
            console.log(JSON.stringify(hits[i]));
        }
    }
}
```

Example

```
var properties = RWS.Rapid.getDefaultSearchProperties();
properties.searchURL = 'RAPID/T_ROB1/TestModule';
properties.types = RWS.Rapid.SymbolTypes.rapidData;
properties.isInUse = false;

var hits = await RWS.Rapid.searchSymbols(properties, 'num', 'Test[a-zA-Z]*');
if (hits.length <= 0) {
    console.log('Could not find test data.');
```

getDefaultSearchProperties

Signature: `getDefaultSearchProperties()`

Returns an object with a default set of properties for searching Rapid symbols.

NOTE! Symbols in a shared module will reside directly under 'RAPID', i.e. use 'Rapid' as searchURL.

NOTE! Symbols in an installed module require the flag "#SYS" instead of module name, i.e. use a string like 'Rapid/T_ROB1/#SYS' as searchURL.

The fields are:

(SearchMethods) method	search methods (see enum SearchMethods)
(string) searchURL	URL to the starting point
(SymbolTypes) types	symbol type (see enum SymbolTypes)
(boolean) isInstalled	is the module containing the datatype installed or loaded, true if an installed module
(boolean) isShared	is the module containing the searched datatype declared in a shared module, true if a shared module
(boolean) recursive	search recursively, i.e. nested symbols like components for a RECORD type will be returned
(boolean) skipShared	skip all results that are declared in shared modules

(boolean) `isInUse`

only return symbols that are used in a Rapid program, i.e. a type declaration that has no declared variable will not be returned when this flag is set true.

The default values are:

```
method = SearchMethods.block
searchURL = 'RAPID'
types = SymbolTypes.any
isInstalled = false
isShared = false
recursive = true
skipShared = false
isInUse = false
```

Return value

An object with default search properties.

Enums

SearchMethods

Supported search method in `searchSymbols`. Specifies the direction of the search, which can be block (down) or scope (up). This means that if the search uses “block” all symbols matching the search criteria in the path declared in “searchURL” and forward will be returned. If using “scope”, all symbols matching the search criteria that can be accessed from the path declared in “searchURL” will be returned.

The fields are as follows:

<code>block: 1</code>	block mode (down from searchURL)
<code>scope: 2</code>	scope mode (up from searchURL)

SymbolTypes

Valid symbol types to search for using `searchSymbols`. This is a flag style enum, i.e. fields can be combined by adding the values of several fields.

NOTE! Not all combinations are relevant, but no errors will be issued if the value is senseless.

The fields are as follows:

<code>undefined: 0</code>	value not set
<code>constant: 1</code>	Rapid constant
<code>variable: 2</code>	Rapid var declared variable
<code>persistent: 4</code>	Rapid pers declared variable
<code>function: 8</code>	Rapid function

procedure: 16	Rapid procedure
trap: 32	Rapid trap
module: 64	Rapid module
task: 128	Rapid task
routine: 8 + 16 + 32	function, procedure or trap
rapidData: 1 + 2 + 4	constant, variable or persistent
any: 255	any of the above

4.3.1 Monitor Object

Description

Monitor is an object used to subscribe to resources that have no other object that can be subscribed.

To create an Monitor object use the factory method `getMonitor(resource, task)` in the `RWS.Rapid` namespace.

The Monitor object can subscribe to changes in rapid execution, program pointer, motion pointer and UI instructions.

Each instance of the Monitor object can subscribe to one resource.

NOTE! The program pointer and motion pointer are due to limitations in other parts of the controller software not usable to follow program execution when Rapid is executing.

Resources

execution

Changes to the state of the Rapid execution. The events will return a string as:

'running'	Rapid is running
'stopped'	Rapid is stopped

program-pointer

Changes to the program pointer.

NOTE! Due to limitations in the underlying architecture events for all changes may not be received. Changes faster than 200 ms will be filtered.

The events will return a program pointer object with the properties:

(string) moduleName	name of the module
(string) routineName	name of the routine
(string) beginPosition	position of start in program file, e.g. '89,9' - row = 89 and column = 9
(string) endPosition	position of end in program file
(boolean) hasValue	flag indicating if data in this object is valid

motion-pointer

Changes to the motion pointer.

NOTE! Due to limitations in the underlying architecture events for all changes may not be received. Changes faster than 200 ms will be filtered.

The events will return a motion pointer object with the properties:

(string) moduleName	name of the module
(string) routineName	name of the routine
(string) beginPosition	position of start in program file, e.g. '89,9' - row = 89 and column = 9
(string) endPosition	position of end in program file
(boolean) hasValue	flag indicating if data in this object is valid

uiinstr

Events when UI instructions' states change. UI instructions are all RAPID instructions that present information in the operator messages view, and may or may not require operator responses. Typical UI instructions are *TPWrite*, *TPReadFK*, *UIMessageBox*, etc.

The events will return an object with the properties:

(string) instruction	instruction type
(string) event	event type, valid values: 'send' 'post' 'abort'
(string) task	RAPID task that executed the instruction
(string) message	message of the instruction
(string) executionLevel	execution level, valid values: 'user' 'normal'
(string) id	unique id of the instruction call, a 'send' event can be matched with an 'abort' event using this id
({}) parameters	an object containing the instructions parameters, which varies depending of instruction and what parameters are provided in the instruction call

Methods

getTitle

Signature: getTitle()

Used internally. Gets the title (part of a URL) of the subscribed resource.

Return value

A string containing the title.

`getResourceString`

Signature: `getResourceString()`

Gets a unique resource URL for subscription.

Return value

A string containing a unique resource URL (as used by RWS.)

`addCallbackOnChanged`

Signature: `addCallbackOnChanged(function callback)`

`callback` a callback function to run when event occurs

Adds a callback function which will be called when the subscribed resource's state changes.

The callback routine can have one in-parameter, which is the new value of the subscribed resource. Contents of the value depends on the resource subscribed to (see section 4.3.1).

Return value

No return value.

Example

```
var counter = 0;
var executionMonitor = RWS.Rapid.getMonitor('execution');
executionMonitor.addCallbackOnChanged(eventData => {
    console.log(`New execution state: ${eventData}.`);
});
```

`onchanged`

This is a function for internal use only. Do not use from external scripts.

`subscribe`

Signature: `subscribe(boolean raiseInitial = false)`

`raiseInitial` flag indicating whether an initial event is raised when subscription is registered

Starts the subscription to the resource defined at instantiation.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var counter = 0;
var executionMonitor = RWS.Rapid.getMonitor('execution');
executionMonitor.addCallbackOnChanged((newValue) => {
    var c = (counter++).toString();
    console.log(`Execution state changed: ${newValue} (${c}).`);
});

try {
    await executionMonitor.subscribe();
} catch(error) {
    console.error(`Subscription to 'execution' failed. >>> ${error}`);
}
```

unsubscribe

Signature: `unsubscribe()`

Stops the subscription to the resource defined at instantiation.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.3.2 Task Object

Description

The Task object corresponds to a Rapid task on the controller. It provides functionality to access modules, data and service routines.

To get a Task object use the factory method `getTask()` directly in the `RWS.Rapid` namespace.

Nested Objects

The Task object contains a nested object, `ServiceRoutine`, which is used to handle service routines.

Methods

getName

Signature: `getName()`

Gets the name of the task.

Return value

A string containing the name of the task.

getProperties

Signature: `getProperties()`

Gets a set of properties of the task. The function returns an object with the properties.

Note that all properties are not necessarily available at all times. The user should check if the property exists (see example.)

Return value

A promise with an object containing:

<code>(string) name</code>	task name
<code>(string) taskType</code>	task type, valid values: 'normal' 'static' 'semistatic' 'unknown'
<code>(string) taskState</code>	task state, valid values: 'empty' 'initiated' 'linked' 'loaded' 'uninitialized'
<code>(string) executionState</code>	execution state, valid values: 'ready' 'stopped' 'started' 'uninitialized'
<code>(string) activeState</code>	active state, valid values: 'on' 'off'
<code>(boolean) isMotionTask</code>	flag indicating if task is a motion task
<code>(string) trustLevel</code>	trust level, valid values: 'sys_fail' 'sys_halt' 'sys_stop' 'none'
<code>(number) id</code>	identifier
<code>(string) executionLevel</code>	current execution level, valid values: 'none'

	'normal'
	'trap'
	'user'
	'unknown'
(string) executionMode	current execution mode, valid values:
	'continous'
	'step_over'
	'step_in'
	'step_out_of'
	'step_back'
	'step_last'
	'stepwise'
	'unknown'
(string) executionType	current execution type, valid values:
	'none'
	'normal'
	'interrupt'
	'external_interrupt'
	'user_routine'
	'event_routine'
	'unknown'
(string) progEntrypoint	program entry point (the configured routine that will be activated by PP to Main calls)
(boolean) bindRef	flag indicated if task is bound
(string) taskInForeground	name of task in foreground

Example

```
var task = await RWS.Rapid.getTask('T_ROB1');
var properties = task.getProperties();
if (properties.hasOwnProperty('executionType')) {
    console.log(`Execution type = ${properties['executionType']}`);
}
```

getServiceRoutines

Signature: `getServiceRoutines()`

Gets a list of `ServiceRoutine` (see 4.3.3) objects for the service routines on the task.

Return value

A Promise with a list of available ServiceRoutine objects.

Example

```
// Move program pointer to first service routine
var task = await RWS.Rapid.getTask('T_ROB1');
var serviceRoutines = await task.getServiceRoutines();
if (serviceRoutines.length > 0) {
    serviceRoutines[0].setPP();
}
```

getData

Signature: `getData(string moduleName, string symbol)`

`moduleName` the name of the module

`symbol` the name of the symbol

Gets the Data object (described later in this chapter) for a symbol. The call will try to get information from the controller to initialize the object.

Return value

A Promise with a Data object

Example

```
var task = await RWS.Rapid.getTask('T_ROB1');
var data = await task.getData('TestModule', 'TestData');
data.setValue('Test String');
```

getProgramInfo

Signature: `getProgramInfo()`

Gets program information, program name and entrypoint, for the task.

Return value

A promise with an object containing:

(string) `name` program name

(string) `entrypoint` name of function that is currently the entry point
(i.e. the routine where execution will start)

getModuleNames

Signature: `getModuleNames()`

Gets module names for the task. This will return both program modules and system modules.

Return value

A Promise with an object containing:

<code>([string]) programModules</code>	list of the names of all program modules
<code>([string]) systemModules</code>	list of the names of all system modules

getModule

Signature: `getModule(string moduleName)`

`moduleName` the name of the module

Gets a module on the task.

Return value

A Promise with a Module object

getPointers

Signature: `getPointers()`

Gets information about the program pointer and the motion pointer. The pointers may not have values and the corresponding `hasValue` flag should always be checked before use.

Return value

A Promise with an object containing:

<code>({}) programPointer</code>	an object with pointer information
<code>({}) motionPointer</code>	an object with pointer information

Pointer information contains:

<code>(string) moduleName</code>	name of the module
<code>(string) routineName</code>	name of the routine
<code>(string) beginPosition</code>	position of start in program file, e.g. '89,9' - row = 89 and column = 9
<code>(string) endPosition</code>	position of end in program file
<code>(boolean) hasValue</code>	flag indicating if value in this object is valid

Example

```
var task = await RWS.Rapid.getTask('T_ROB1');
var pointers = await task.getPointers();
if (pointers.programPointer.hasValue === true) {
    var r = pointers.programPointer['routineName'];
    console.log(`PP is in routine '${r}'`);
}
```

movePPToRoutine

Signature: `movePPToRoutine(string routineName, boolean userLevel = false, string moduleName = undefined)`

<code>routineName</code>	the name of the routine
<code>userLevel</code>	flag indicating if execution is to be on user level
<code>moduleName</code>	optional module name

Moves the program pointer for the task to a routine. Optionally the execution level can be set to “user”, i.e., execute as a service routine.

Optionally a module name can be specified.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

abortServiceRoutine

Signature: `abortServiceRoutine()`

Aborts the current execution level of the task, under the condition that the execution level is “user”, i.e., a service routine.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.3.3 ServiceRoutine Object

Description

The ServiceRoutine object is nested in the Task object. It provides a basic interface to facilitate executing service routines.

ServiceRoutine objects cannot be explicitly created, they are returned as results of other methods.

The execution of the service routine is controlled with methods in the `RWS.Rapid` namespace, i.e. `startExecution` and `stopExecution`.

Methods

getName

Signature: `getName()`

Gets the name of the service routine.

Return value

A string containing the name of the service routine.

getUrl

Signature: `getUrl()`

Gets an RWS URL for the service routine. Used internally by api.

Return value

A string containing the URL.

setPP

Signature: `setPP()`

Set program pointer to the service routine.

After the program pointer has been set; use `RWS.Rapid.startExecution()` to run the routine.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.3.4 Module Object

Description

The Module object corresponds to a Rapid module on the controller. It provides functionality to access Rapid data.

To get an Module object use the factory method `getModule()` in the `RWS.Rapid` namespace.

Methods

getName

Signature: `getName()`

Gets the name of the module.

Return value

A string containing the module name.

getTaskName

Signature: `getTaskName()`

Gets the name of the task that contains the module.

Return value

A string containing the task name.

getProperties

Signature: `getProperties()`

Gets the properties of the module.

Return value

A Promise with an object containing:

<code>(string) taskName</code>	task's name
<code>(string) moduleName</code>	module's name
<code>(string) fileName</code>	the name of the file the module is saved as
<code>(string) attributes</code>	attributes

Example

```
var module = await RWS.Rapid.getModule('T_ROB1', 'TestModule');
var properties = await module.getProperties();
console.log(`Module file name is '${properties.fileName}'`);
```

getData

Signature: `getData(string symbolName)`

`symbolName` the name of the symbol

Gets the Data object for a symbol. The call the controller to initialize the object with current properties. This will verify that the symbol exists. Note that the value of the is not read and must be read explicitly.

Return value

A Promise with a Data object.

Example

```
var module = await RWS.Rapid.getModule('T_ROB1', 'TestModule');
var data = await module.getData('TestData');
var testValue = data.getValue();
```

4.3.5 Data Object

Description

The Data object corresponds to a Rapid data on the controller. It provides functionality to access read, write and subscribe to the data.

To get an Data object use the factory method `getData()` in the `RWS.Rapid` namespace, or the Task and Module objects.

Methods

`getTitle`

Signature: `getTitle()`

Used internally. Gets the title (part of a URL) of the subscribed resource.

Return value

A string containing the title.

`getProperties`

Signature: `getProperties()`

Gets the basic properties of the data. A call to this method will refresh the contents of the object.

NOTE! Calling the individual property accessors will return the values from the last call to this method and are consequently not necessarily the actual values.

Return value

A Promise with an object containing:

<code>(string) taskName</code>	task's name
<code>(string) moduleName</code>	module's name
<code>(string) symbolName</code>	symbol's name
<code>(string) dataType</code>	symbol's data type
<code>(string) symbolType</code>	the declaration type of the data, valid values: ' <code>constant</code> ' ' <code>variable</code> ' ' <code>persistent</code> '
<code>([number]) dimensions</code>	list of dimensions for arrays
<code>(string) scope</code>	the data's scope, valid values: ' <code>local</code> ' ' <code>task</code> ' ' <code>global</code> '
<code>(string) dataTypeURL</code>	RWS URL to the data's type symbol

Example

```
var data = await RWS.Rapid.getData('T_ROB1', 'TestModule', 'TestData');
var properties = await data.getProperties();
console.log(`Data type is '${properties.dataType}'`);
```


getName

Signature: `getName()`

Gets the name of the symbol.

Return value

A string containing the symbol name.

getModuleName

Signature: `getModuleName()`

Gets the name of the module containing the symbol.

Return value

A string containing the module name.

getTaskName

Signature: `getTaskName()`

Gets the name of the task containing the symbol.

Return value

A string containing the task name.

getDataType

Signature: `getDataType()`

Gets the data type of the symbol, i.e. Rapid data types *num*, *string*, etc.

Return value

A promise with a string containing the data type.

getSymbolType

Signature: `getSymbolType()`

Gets the symbol declaration type of the data.

Return value

A promise with a string containing the symbol type.

Possible values: `'constant'`, `'variable'` or `'persistent'`

getDimensions

Signature: `getDimensions()`

Gets the dimensions of the symbol, if the symbol is an array or multi-dimensional matrix. The method returns a list of number which corresponds to the dimensions in order. If the symbol is atomic (i.e. has no dimensions) the return value will be an empty list.

Return value

A promise with a list of numbers.

getScope

Signature: `getScope()`

Gets the data's scope. The value indicates how accessible the data is, i.e. local, global or task.

Return value

A promise with a string.

Possible values: 'local', 'task' or 'global'

getTypeURL

Signature: `getTypeURL()`

Gets the RWS URL to the data type's definition of the symbol. For complex data types, e.g. Record types, this is the entry point for when evaluating the value of the symbol.

Return value

A promise with a string containing the data type URL.

getValue

Signature: `getValue()`

Gets the value of the symbol. This will return the latest retrieved value. If value has not been retrieved yet, it will be retrieved by this call. Use the `fetch` function to update the value before subsequent calls to this method.

The return value will be parsed and returned as a JavaScript value corresponding to the Rapid value, i.e. a num will be returned as number, an array will be returned as an JavaScript array (`[]`), a Record will be returned as an object with properties named as the record components in Rapid, etc.

Return value

A promise with a parsed data value.

Example

Lets assume that a Record is declared as follows:

```
Record TestRecord
    num Num1;
```

```
    num Num2;  
    string String1;  
EndRecord
```

Then it can be read as follows:

```
var data = await RWS.Rapid.getData('T_ROB1', 'TestModule', 'TestRecord');  
var testValue = await data.getValue();  
if (testValue.Num1 > 17) {  
    console.log(`Num1 = '${testValue.Num1}' is greater than 17.`);  
}  
console.log(testValue.String1);
```

getArrayItem

Signature: `getArrayItem(...number indexes)`

`indexes` a parameter list of numbers

Gets the value of an element in an array specified by the provided indexes. If the array is multidimensional an index can be provided for each dimension. If the number of indexes or the indexes are greater than the arrays size in that dimension, the call will be rejected.

The indexes are 1-based as in Rapid.

Return value

A promise with a parsed data value.

Example

Lets assume that an array is declared as follows:

```
PERS num TestArray52{5,2} := [[0,1],[0,4],[0,9],[1,2],[2,5]];
```

Then its elements can be read as follows:

```
var array = await RWS.Rapid.getData('T_ROB1', 'TestModule', 'TestArray52');  
var number = await array.getArrayItem(3,2);  
if (number === 9) console.log('Correctly read number from array.');
```

```
var tuple = await array.getArrayItem(3);  
if (tuple === [0,9]) console.log('Correctly read tuple from array.');
```

getRecordItem

Signature: `getRecordItem(...string components)`

`components` a parameter list of component names

Gets the value of an element in a record specified by the provided component names. If the record contains nested records, several names can be provided to get a value deeper in the hierarchy. If the list of component names cannot be traversed, the call will be rejected.

Return value

A promise with a parsed data value.

Example

Lets assume Records declared as follows:

```
RECORD TestRecord
    num Code;
    num State;
    string Status;
ENDRECORD

RECORD NestedTestRecord
    TestRecord InnerTest;
    TestRecord OuterTest;
ENDRECORD

PERS NestedTestRecord TestNested := [[0,1,"OK"],[0,2,"Error"]];
```

Then the components can be read as follows:

```
var rec = await RWS.Rapid.getData('T_ROB1','TestModule',' TestNested ');
var status = await rec.getRecordItem('InnerTest', 'Status');
if (status === 'OK') console.log('Inner test is OK.');
```

```
status = await rec.getRecordItem('OuterTest', 'Status');
if (status === 'OK') console.log('Outer test is OK.');
```

getRawValue

Signature: `getRawValue()`

Gets the RWS string value of the symbol. This will return the latest retrieved value. If the value has not been retrieved yet, it will be retrieved by this call. Use the `fetch` function to update the value before subsequent calls to this method.

NOTE! Use this function only if you know the format of the RWS string and need to optimize for speed.

Return value

A promise with a string containing the data value.

fetch

Signature: `fetch()`

Retrieves the latest value from the controller. The value is stored in the object and can be accessed using a call to `getValue()`.

NOTE! If the Data object has an active subscription to the value, it is not necessary to call `fetch`. The value will be updated by the events.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var testValue1 = '';
var testValue2 = '';
var data1 = await RWS.Rapid.getData('T_ROB1', 'TestModule', 'TestData1');
var data2 = await RWS.Rapid.getData('T_ROB1', 'TestModule', 'TestData2');

...

async function updateValues() {
    try {
        await data1.fetch();
        await data2.fetch();
        testValue1 = await data1.getValue();
        testValue2 = await data2.getValue();
    } catch (error) {
        console.error(`We have a problem. ${error}`);
    }
}
```

setValue

Signature: setValue({} value)

value object or atomic with the new value

Sets the value of the symbol to the value provided.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

Lets assume that a Record is declared as follows:

```
Record TestRecord
    num Num1;
    num Num2;
    string String1;
EndRecord
```

Then it can be set as follows:

```
var data = {
    Num1: 17,
```

```

        Num2: 42,
        String1: 'Some random numbers'
    };
    var record = await RWS.Rapid.getData('T_ROB1','TestModule','TestRecord');
    await record.setValue(data);

```

setArrayItem

Signature: `setArrayItem(any value, ...number indexes)`

value object or atomic with the new value

indexes a parameter list of numbers

Sets the value of an element in an array specified by the provided indexes. If the array is multidimensional an index can be provided for each dimension. If the number of indexes or the indexes are greater than the arrays size in that dimension, the call will be rejected.

The indexes are 1-based as in Rapid.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

Lets assume that an array is declared as follows:

```
PERS num TestArray25{2,5} := [[0,1,0,4,0],[9,1,2,2,5]];
```

Then its elements can be written as follows:

```

var array = await RWS.Rapid.getData('T_ROB1','TestModule','TestArray25');
await array.setArrayItem(42,2,3); // will write the value 42 to position 2,3

```

setRecordItem

Signature: `setRecordItem(any value, ...string components)`

value object or atomic with the new value

components a parameter list of component names

Sets the value of an element in a record specified by the provided component names. If the record contains nested records, several names can be provided to get a value deeper in the hierarchy. If the list of component names cannot be traversed, the call will be rejected.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

Lets assume Records declared as follows:

```
RECORD TestRecord
```

```

        num Code;
        num State;
        string Status;
    ENDRECORD

    RECORD NestedTestRecord
        TestRecord InnerTest;
        TestRecord OuterTest;
    ENDRECORD

    PERS NestedTestRecord Test := [[0,1,"OK"],[0,2, "Error"]];

```

Then the components can be written as follows:

```

var rec = await RWS.Rapid.getData('T_ROB1','TestModule','Test');
await rec.setRecordItem('OK', 'OuterTest', 'Status');
await rec.setRecordItem(7, 'OuterTest', 'State');
var inner = {Code:0, State: 2, Status: 'Warning'};
await rec.setRecordItem(inner, 'InnerTest');

```

setRawValue

Signature: setRawValue(string value, ...number indexes)

value the new value

indexes an optional list of numbers that specify an element in arrays and records

Sets the value of the symbol to the value provided. This requires the value string to be on the RWS format, and as that can be a bit complicated it is recommended that the `setValue` function is used instead.

By providing indexes, in the in-parameter `indexes`, a specific element in an array (multidimensional) or record can be set independent of the other elements in the structure. The indexes are 1-based as in Rapid.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

getResourceString

Signature: getResourceString()

Gets a URL to the resource. This URL is used to identify the resource when subscribing.

Return value

A string containing a unique resource URL (as used by RWS.)

addCallbackOnChanged

Signature: addCallbackOnChanged(function callback)

callback a callback function to run when event occurs

Adds a callback function which will be called when the subscribed symbol's value changes.

The callback can have one argument, which will contain the data value. The value will be the same as received when `getRawValue` is called. This means that the value will be in raw RWS format. To get a more friendly parsed value, use the `getValue` function available on the data object from within the callback function.

Return value

No return value.

onchanged

This is a function for internal use only. Do not use from external scripts.

subscribe

Signature: `subscribe(boolean raiseInitial = false)`

raiseInitial flag indicating whether an initial event is raised when subscription is registered

Subscribes to changes on the value of the symbol.

IMPORTANT! Note that only persistent data variables (PERS) support subscriptions.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var data = await RWS.Rapid.getData('T_ROB1', 'TestModule', 'TestData');
data.addCallbackOnChanged((newValue) => {
    console.log(`Testdata value changed: ${newValue}.`);
});

try {
    await data.subscribe();
} catch(error) {
    var resource = data.getResourceString();
    console.error(`Subscription to '${resource}' failed. >>> ${error}`);
}
```

unsubscribe

Signature: `unsubscribe()`

Stops the subscription to changes on the value of the symbol.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.4 RWS.IO Namespace

Description

The IO namespace, `RWS.IO`, contains methods to manipulate I/O signals.

The namespace contains the following objects; Network, Device and Signal. These are described closer in this section.

Methods

`getNetwork`

Signature: `getNetwork(string networkName)`

`networkName` the name of the network

Gets a Network object. The presence of the network is verified and if it does not exist the call will be rejected.

Return value

A Promise with a Network object.

`getDevice`

Signature: `getDevice(string networkName, string deviceName)`

`networkName` the name of the network

`deviceName` the name of the device

Gets a Device object. The presence of the device is verified and if it does not exist the call will be rejected.

Return value

A Promise with a Device object.

`getSignal`

Signature: `getSignal(string signal)`

`signal` the name of the signal

Gets a Signal object. The presence of the signal is verified and if it does not exist the call will be rejected. The basic properties for the signal are retrieved from the controller, however the value is not retrieved until any of the methods `getValue()` or `fetch()` is called.

Return value

A Promise with a Signal object.

Example

```
var signal = await RWS.IO.getSignal('TestDI');
var testValue = await signal.getValue();
if (testValue === 1){
    mySetSomeIndicator(1);
}
```

setSignalValue

Signature: setSignalValue(string signal, number value)

signal the name of the signal

value the new value of the signal

Sets a value on a signal. The call will fail if the value string is not valid for the signal type.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

searchSignals

Signature: searchSignals({} filter)

filter an object with filter information:

(string) name	signal name
(string) device	device name
(string) network	network name
(string) category	category string
(string) category-pon	
(string) type	type of signal, valid values: 'DI', 'DO', 'AI', 'AO', 'GI' or 'GO'
(boolean) invert	inverted signals
(boolean) blocked	blocked signals

Search system for signals matching the filter. The signal `name` does not have to be an exact match, any signal which name begins with the string will be returned.

The `type` filter can contain several types by grouping the values within '[' and ']'. The signal types should be separated by comma characters, e.g. '[DI,DO]'.

Return value

A Promise with a list of Signal objects that match the filters.

Example

```
var filter = {
    name: 'TestDI',
```

```
}  
var signals = await RWS.IO.searchSignals(filter);  
if (signals.length > 0){  
    for (let i = 0; i < signals.length; i++){  
        var testValue = await signals[i].getValue();  
        console.log(`${signals[i].getName()} = ${testValue}`);  
    }  
}
```

Example

```
var filter = {  
    name: 'Test',  
    type: '[DI,DO]'  
}  
var signals = await RWS.IO.searchSignals(filter);  
if (signals.length > 0){  
    for (let i = 0; i < signals.length; i++){  
        var testValue = await signals[i].getValue();  
        console.log(`${signals[i].getName()} = ${testValue}`);  
    }  
}
```

4.4.1 Network Object

Description

The Network object represents an IO network and can be used to check the states of the network and access signals.

Network contains two states, physical state and logical state, whose values are not updated unless explicitly ordered by calling `fetch()`. The first time any of the states are accessed, `fetch()` is called implicitly and both states are updated.

Methods

`getName`

Signature: `getName()`

Gets the name of the Network.

Return value

A string containing the network's name.

`getPhysicalState`

Signature: `getPhysicalState()`

Gets the physical state of the Network. If the Network instance has not been accessed before, it will retrieve current states from the controller and then return the physical state. If the Network instance has been accessed, the `fetch()` function must be called first to ensure that the states are up to date.

Return value

A promise with a string containing the physical state.

Possible values: 'halted', 'running', 'error', 'startup', 'init' or 'unknown'

Example

```
var device = await RWS.IO.getDevice('TestNet', 'TestDevice');
var state = await device.getPhysicalState();
if (state !== 'running') {
    myHandleDeviceState('TestDevice', state);
}
```

getLogicalState

Signature: `getLogicalState()`

Gets the logical state of the Network. If the Network instance has not been accessed before, it will retrieve current states from the controller and then return the logical state. If the Network instance has been accessed, the `fetch()` function must be called first to ensure that the states are up to date.

Return value

A promise with a string containing the logical state.

Possible values: 'stopped', 'started' or 'unknown'

fetch

Signature: `fetch()`

Retrieves the current states from the controller. The values are stored in the object and can be accessed using calls to `getLogicalState()` or `getPhysicalState()`.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.4.2 Device Object

Description

The Device object represents an IO device and can be used to check the states of the device and get signals that are configured on the device.

Device contains two states, physical state and logical state, whose values are not updated unless explicitly ordered by calling `fetch()`. The first time any of the states are accessed, `fetch()` is called implicitly and both are values updated.

Methods

`getName`

Signature: `getName()`

Gets the name of the Device.

Return value

A string containing the device's name.

`getNetworkName`

Signature: `getNetworkName()`

Gets the name of the Network on which the Device resides.

Return value

A string containing the network's name.

`getPhysicalState`

Signature: `getPhysicalState()`

Gets the physical state of the Device. If the Device instance has not been accessed before, it will retrieve current states from the controller and then return the physical state. If the Device instance has been accessed, the `fetch()` function must be called first to ensure that the states are up to date.

Return value

A promise with a string containing the physical state.

Possible values: 'deact', 'running', 'error', 'unconnect', 'unconfig', 'startup', 'init' or 'unknown'

`getLogicalState`

Signature: `getLogicalState()`

Gets the logical state of the Device. If the Device instance has not been accessed before, it will retrieve current states from the controller and then return the logical state. If the Device instance has been accessed, the `fetch()` function must be called first to ensure that the states are up to date.

Return value

A promise with a string containing the logical state.

Possible values: 'disabled', 'enabled' or 'unknown'

getNetwork

Signature: `getNetwork()`

Gets an object for the parent network.

Return value

A Network object.

getSignal

Signature: `getSignal(string signalName)`

`signalName` the name of the signal

Gets a Signal object with properties retrieved from the controller. The signal must reside on the current device. The value is not retrieved until one of the methods `getValue()` or `fetch()` is called.

Return value

A promise with a Signal object.

4.4.3 Signal Object

Description

The Signal object represents an IO signal and can be used to read and set signals value as well as checking some properties.

The signal's properties and value are not updated unless explicitly ordered by calling `fetch()`. The first time any property or value are accessed, `fetch()` is called implicitly and all is updated.

Methods

getName

Signature: `getName()`

Gets the name of the Signal.

Return value

A string containing the signal's name.

getPath

Signature: `getPath()`

Gets the part of a URL that is the signal's path.

Return value

A string containing the signal's path.

getNetworkName

Signature: `getNetworkName()`

Gets the name of the network the Signal resides on.

Return value

A string containing the network's name.

getDeviceName

Signature: `getDeviceName()`

Gets the name of the device the Signal resides on.

Return value

A string containing the device's name.

getTitle

Signature: `getTitle()`

Used internally. Gets the title (part of a URL) of the subscribed resource.

Return value

A string containing the signal's title.

getIsSimulated

Signature: `getIsSimulated()`

Gets the flag indicating whether the signal is simulated. This will update the value at the first call, but subsequent calls must call `fetch()` first.

Return value

A Promise with a boolean.

getQuality

Signature: `getQuality()`

Gets the quality of the last update of the signal.

Return value

A Promise with a string.

Possible values: 'bad', 'good' or 'unknown'

getCategory

Signature: getCategory()

Gets the category field of the signal, as defined in the configuration database.

Return value

A Promise with a string.

getAccessLevel

Signature: getAccessLevel()

Gets the access level field of the signal, as defined in the configuration database.

Return value

A Promise with a string.

getWriteAccess

Signature: getWriteAccess()

Gets the write access field of the signal, as defined in the configuration database.

Return value

A Promise with a string.

getSafeLevel

Signature: getSafeLevel()

Gets the safe level field of the signal, as defined in the configuration database.

Return value

A Promise with a string.

getType

Signature: getType()

Gets the type field of the signal, as defined in the configuration database.

Return value

A Promise with a string.

getDevice

Signature: `getDevice()`

Gets the Device object for the device this signal resides on.

NOTE! This will create a new object and not return an existing one if one exists.

Return value

A Promise with a Device object.

getValue

Signature: `getValue()`

Gets the signals value. This will update the value at the first call, but subsequent calls must call `fetch()` first.

Return value

A Promise with a number.

setValue

Signature: `setValue(number value)`

`value` the new value

Sets the value of the signal to the value provided.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

addCallbackOnChanged

Signature: `addCallbackOnChanged(function callback)`

`callback` a callback function to run when event occurs

Adds a callback function which will be called when the subscribed signal's value changes.

The callback can have one argument, which will contain the signal value (same as is received when `getValue` is called.)

Return value

No return value.

onchanged

This is a function for internal use only. Do not use from external scripts.

subscribe

Signature: `subscribe(boolean raiseInitial = false)`

`raiseInitial` flag indicating whether an initial event is raised when subscription is registered

Starts the subscription to changes on the value of the signal.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

unsubscribe

Signature: `unsubscribe()`

Stops the subscription to changes on the value of the signal.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var signal = await RWS.IO.getSignal('TestDI');
signal.addCallbackOnChanged((newValue) => {
    console.log(`Test signal value changed: ${newValue}.`);
});

try {
    await signal.subscribe();
} catch(error) {
    var resource = signal.getResourceString();
    console.error(`Subscribe to '${resource}' failed. >>> ${error}`);
}

...

try {
    await signal.unsubscribe();
} catch(error) {
    var resource = signal.getResourceString();
    console.error(`Unsubscribe to '${resource}' failed. >>> ${error}`);
}
```

4.5 RWS.CFG Namespace

Description

The `RWS.CFG` namespace, contains methods to read and write to the configuration database (the controller's system parameters.)

The namespace contains the following objects; Domain, Type and Instance. These are described closer in this section.

There are also a set of methods that are available directly in the `RWS.CFG` namespace.

NOTE! The identifiers for the domains, types, instances and attributes can easily be determined by creating an instance and then saving that domain to file. In that file the identifiers can be found by looking at the instance created.

Methods

`getDomains`

Signature: `getDomains()`

Gets a list of valid domains in the configuration database.

Return value

A Promise with a list of Domain objects.

`saveConfiguration`

Signature: `saveConfiguration(string domain, string filePath)`

`domain` the domain name, e.g. 'EIO', 'SYS', 'MOC', etc.

`filePath` the path to the file, including file name

Saves the configurations for a domain to file.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

`verifyConfigurationFile`

Signature: `verifyConfigurationFile(string filePath, string action = 'add')`

`filePath` the path to the file, including file name

`action` the validation method, valid values: 'add', 'replace' and 'add-with-reset'.

Verifies a configuration file. The validation method must match the intended load method.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected. The promise is rejected both in case of an invalid and/or missing file.

`loadConfiguration`

Signature: `loadConfiguration(string filePath, string action = 'add')`

`filePath` the path to the file, including file name

`action` the validation method, valid values: 'add', 'replace' and 'add-with-reset'.

Loads a configuration file. The validation method indicates in which way duplicates should be handled.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

getTypes

Signature: `getTypes(string domain)`

`domain` the name of the domain

Gets all types on a domain.

Return value

A Promise with a list of Type objects.

getInstances

Signature: `getInstances(string domain, string type)`

`domain` the name of the domain

`type` the name of the type

Gets all instances on a type.

Return value

A Promise with a list of Instance objects.

getInstanceByName

Signature: `getInstanceByName(string domain, string type, string name)`

`domain` the name of the domain

`type` the name of the type

`name` the name of the instance

Gets a specific instance on a type. The instance is identified by its name.

NOTE! Not all instances have names, use `getInstanceById()` or `getInstances()` in that case.

Return value

A Promise with an Instance object.

getInstanceById

Signature: `getInstanceById(string domain, string type, string id)`

domain the name of the domain
type the name of the type
id the identifier of the instance

Gets a specific instance on a type. The instance is identified by the id created by RWS.

Return value

A Promise with an Instance object.

createInstance

Signature: `createInstance(string domain, string type, string name = '')`

domain the domain
type the type
name the name of the instance

Creates a new instance on the controller. The instance will be blank and setting it's attributes by calling `updateAttributesByName()` or `updateAttributesById()` is expected. If the Type does not contain a name attribute the name parameter can be omitted.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

updateAttributesByName

Signature: `updateAttributesByName(string domain, string type, string name, {} attributes)`

domain the domain
type the type
name the name of the instance
attributes a collection of key-value pairs

Updates the instance's attributes. The attributes Object should only contain valid members, i.e. attributes valid for the instance's specific Type, and the corresponding values.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var attr = {};  
attr['Label'] = 'Test IO Label';  
attr['SignalType'] = 'DI';  
attr['Access'] = 'All';
```

```
await RWS.CFG.updateAttributesByName('EIO', 'EIO_SIGNAL', 'TestDI', attr);
```

updateAttributesById

Signature: `updateAttributesById(string domain, string type, string id, {} attributes)`

<code>domain</code>	the domain
<code>type</code>	the type
<code>id</code>	the identifier of the instance
<code>attributes</code>	a collection of key-value pairs

Updates the instance's attributes. The attributes Object should only contain valid members, i.e., attributes valid for the instance's specific Type, and the corresponding values.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

deleteInstanceByName

Signature: `deleteInstanceByName(string domain, string type, string name)`

<code>domain</code>	the domain
<code>type</code>	the type
<code>name</code>	the name of the instance

Deletes an instance on the controller.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

deleteInstanceById

Signature: `deleteInstanceById(string domain, string type, string id)`

<code>domain</code>	the domain
<code>type</code>	the type
<code>id</code>	the identifier of the instance

Deletes an instance on the controller.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.5.1 Domain Object

Description

The Domain object represents a configuration domain in the configuration database, e.g. the I/O System domain, Motion domain, etc. Domain objects are used to access configuration information on the specific domain it was instantiated for. Domain objects cannot be explicitly created, they are retrieved with the `getDomains()` function in the `RWS.CFG` namespace.

Methods

`getName`

Signature: `getName()`

Gets the name of the domain.

Return value

A string containing the name of the domain.

`getTypes`

Signature: `getTypes()`

Gets a list of all types in the domain.

Return value

A Promise with a list of Type objects.

`getInstances`

Signature: `getInstances(string type)`

`type` the name of the type which instances to get

Gets all instances on a type.

Return value

A Promise with a list of Instance objects.

`getInstanceByName`

Signature: `getInstanceByName(string type, string name)`

`type` the name of the type

`name` the name of the instance

Gets a specific instance on a type. The instance is identified by its name.

NOTE! Not all instances have names, use `getInstanceById` or `getInstances` in that case.

Return value

A Promise with an Instance object.

getInstanceById

Signature: `getInstanceById(string type, string id)`

`type` the name of the type

`id` the identifier of the instance

Gets a specific instance on a type. The instance is identified by the id created by RWS.

Return value

A Promise with an Instance object.

createInstance

Signature: `createInstance(string type, string name = '')`

`type` the type

`name` the name of the instance

Creates a new instance on the controller. The instance will be blank and setting its attributes by calling `updateAttributesByName` or `updateAttributesById` is expected. If the Type does not contain a name attribute the name parameter can be omitted.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

updateAttributesByName

Signature: `updateAttributeByName(string type, string name, {} attributes)`

`type` the type

`name` the name of the instance

`attributes` a collection of key-value pairs

Updates the instance's attributes. The attributes Object should only contain valid members, i.e. attributes valid for the instance's specific Type, and the corresponding values. The function returns a Promise with the updated Instance object.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

updateAttributesById

Signature: `updateAttributeById(string type, string id, {} attributes)`

type	the type
id	the identifier of the instance
attributes	a collection of key-value pairs

Updates the instance's attributes. The attributes Object should only contain valid members (attributes for the specific instance Type) and the corresponding values. The function returns a Promise with the updated Instance object.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

deleteInstanceByName

Signature: `deleteInstanceByName(string type, string name)`

type	the type
name	the name of the instance

Deletes an instance on the controller.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

deleteInstanceById

Signature: `deleteInstanceById(string type, string id)`

type	the type
id	the identifier of the instance

Deletes an instance on the controller.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

saveToFile

Signature: `saveToFile(string filePath)`

filePath	the path to the file, including file name
----------	---

Saves the configuration of the domain to a file.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.5.2 Type Object

Description

The Type object represents a configuration type in the configuration database, e.g. Signal or System Input in the I/O System domain. Type objects cannot be explicitly created, they are retrieved with the `getTypes()` function in the `RWS.CFG` namespace or a Domain object.

Methods

`getName`

Signature: `getName()`

Gets the name of the type.

Return value

A string containing the name of the type.

`getDomainName`

Signature: `getDomainName()`

Gets the name of the type's parent domain.

Return value

A string containing the name of the domain.

`getDomain`

Signature: `getDomain()`

Gets the type's parent domain.

Return value

A Domain object.

`getInstances`

Signature: `getInstances()`

Gets all instances on the type.

Return value

A Promise with a list of Instance objects.

`getInstanceByName`

Signature: `getInstanceByName(string name)`

`name` the name of the instance

Gets a specific instance on the type. The instance is identified by its name.

NOTE! Not all instances have names, use `getInstanceById()` or `getInstances()` in that case.

Return value

A Promise with an Instance object.

`getInstanceById`

Signature: `getInstanceById(string id)`

`id` the identifier of the instance

Gets a specific instance on the type. The instance is identified by the id created by the configuration database.

Return value

A Promise with an Instance object.

`createInstance`

Signature: `createInstance(string name = '')`

`name` the name of the instance

Creates a new instance on the controller. The instance will be blank and setting its attributes by calling `updateAttributesByName()` or `updateAttributesById()` is expected. If the Type does not contain a name attribute the name parameter can be omitted.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

`updateAttributesByName`

Signature: `updateAttributeByName(string name, {} attributes)`

`name` the name of the instance

`attributes` a collection of key-value pairs

Updates the instance's attributes. The attributes object should only contain valid members, i.e. attributes valid for the instance's specific Type, and the corresponding values. The function returns the updated Instance object.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

`updateAttributesById`

Signature: `updateAttributeById(string id, {} attributes)`

`id` the identifier of the instance

`attributes` a collection of key-value pairs

Updates the instance's attributes. The attributes object should only contain valid members, i.e. attributes valid for the instance's specific Type, and the corresponding values. The function returns the updated Instance object.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

deleteInstanceByName

Signature: `deleteInstanceByName(string name)`

`name` the name of the instance

Deletes an instance on the controller.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

deleteInstanceById

Signature: `deleteInstanceById(string id)`

`id` the id of the instance

Deletes an instance on the controller.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.5.3 Instance Object

Description

The Instance object represents a configuration instance in the configuration database, e.g. a signal instance, a task instance, etc. Instances belong to a configuration type, e.g. signal instances belong to the "Signal" type, which in this API are represented by the Type (see 4.5.2) object.

Methods

getInstanceId

Signature: `getInstanceId()`

Gets the instance identifier. The identifier is read-only and set by the configuration database when a new instance is created.

Return value

A string containing the identifier.

getInstanceName

Signature: `getInstanceName()`

Gets the name of the instance.

Return value

A string containing the name of the instance.

getTypeName

Signature: `getTypeName()`

Gets the name of the instance's parent type.

Return value

A string containing the name of the type.

getType

Signature: `getType()`

Gets the instance's parent Type object.

Return value

A Type object.

getAttributes

Signature: `getAttributes()`

Gets the instance's attributes.

Return value

An object containing the attributes as properties.

Example

updateAttributes

Signature: `updateAttributes({} attributes)`

`attributes` a collection of key-value pairs

Updates the instance's attributes. The attributes object should only contain valid members, i.e. attributes valid for the instance's specific Type, and the corresponding values. The function returns a Promise with the updated Instance object.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var attr = {};
attr['Label'] = 'Test IO Label';
attr['SignalType'] = 'DI';
attr['Access'] = 'All';
try{
    var x = await RWS.CFG.getInstanceByName('EIO', 'EIO_SIGNAL', 'TestDI');
    await x.updateAttributes(attr);
} catch(error) {
    console.log(`Failed to write settings.\n${error}`);
}
```

delete

Signature: delete()

Deletes the instance on the controller. After calling this method, the Instance object is no longer valid.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.6 RWS.FileSystem Namespace

Description

The `RWS.FileSystem` namespace contains methods to access and modify files and directories on the controller's filesystem.

The namespace contains the Directory and File objects. These are described closer in this section.

There are also some methods available directly in the `RWS.FileSystem` namespace.

Methods

getDirectory

Signature: getDirectory(string directoryPath)

directoryPath the complete path to the directory

Gets a Directory object for a path. The contents of the Directory will be retrieved when this method is called and can be accessed by calling the `getContents` method on the object.

NOTE! The contents are only refreshed when the `fetch` method is called. If the contents change on the file system after the Directory object was created, it will not be reflected in the results received by calling `getContents` until `fetch` has been called.

Return value

A Promise with a Directory object.

Example

```
const pathModules = '$HOME/modules';
var directory = await RWS.FileSystem.getDirectory(pathModules);
var contents = await directory.getContents();
console.log(`Contents of '${pathModules}'`);
console.log('Directories:');
for (let item of contents.directories) {
    console.log(`\t${item.name}`);
}
console.log('Files:');
for (let item of contents.files) {
    console.log(`\t${item.name}`);
}
```

createDirectory

Signature: `createDirectory(string directoryPath)`

`directoryPath` the complete path to the directory

Creates a new directory in the file system at the given path. A Directory object for the new directory will be returned.

NOTE! The entire path will be created if any directory in the path is missing.

Return value

A Promise with a Directory object.

getFile

Signature: `getFile(string filePath)`

`filePath` the complete path to the file

Gets a File object for a path. The contents of the file will be retrieved when this method is called.

Return value

A Promise with a File object.

createFileObject

Signature: `createFileObject(string filePath)`

`filePath` the complete path to the file

Creates a new File object for the path. The method does not create the file on the file system. It can be done by adding content to the object by calling `setContents` and then creating the file by calling the `save` method.

NOTE! This does not verify that the file exists, however that can be done by calling the `fileExists` method.

Return value

A Promise with a File object.

4.6.1 Directory Object

Description

The Directory object encapsulates a directory in the controller's filesystem. The object contains the contents of the directory in an object with two lists, as follows;

<code>([]) directories</code>	list of dictionary property objects
<code>([]) files</code>	list of file property objects

The contents of the lists are objects containing the properties of the corresponding directory or file. A directory's property object contains;

<code>(string) name</code>	the dictionary's name
<code>(Date) created</code>	the creation date
<code>(Date) modified</code>	the last modification date

A file's property object contains;

<code>(string) name</code>	the file's name
<code>(Date) created</code>	the creation date
<code>(Date) modified</code>	the last modification date
<code>(number) size</code>	the size of the file in bytes
<code>(boolean) isReadOnly</code>	flag indicating if file is read only

Methods

getPath

Signature: `getPath()`

Gets the directory's path.

Return value

The path to the directory the Directory object is associated with.

getProperties

Signature: `getProperties()`

Gets the properties of the directory.

Return value

A Promise with an object containing the directory's properties defined as;

<code>(string) name</code>	the dictionary's name
<code>(Date) created</code>	the creation date
<code>(Date) modified</code>	the last modification date

getContents

Signature: `getContents()`

Gets the contents of the directory.

Return value

A Promise with an object containing lists for directories and files (see description 4.6.1).

create

Signature: `create(string newDirectory)`

`newDirectory` the name of the new directory

Creates a new directory in the file system in the Directory object's path.

Return value

A Promise with a Directory object for the new directory.

delete

Signature: `delete()`

Deletes the directory the object is associated with. The Directory object will be marked as deleted and subsequent method calls will be rejected. However, if any other entity creates the directory in the file system, calling the `fetch` method will reset the deleted status and the object will no longer reject methods.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

createFileObject

Signature: `createFileObject(string fileName)`

`fileName` the name of the file to create

Creates a new File object in the directory. The method does not create the file on the file system. It can be done by adding content to the object by calling `setContentts` and then creating the file by calling the `save` method.

Return value

A File object.

rename

Signature: `rename(string newName)`

`newName` the new name of the directory

Renames the directory to the name provided.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

copy

Signature: `copy(string copyPath, boolean overwrite, boolean isRelativePath = true)`

`copyPath` the path of the copy

`overwrite` flag indicating whether to overwrite existing directory

`isRelativePath` flag indicating whether path is relative to original directory or absolute

Copies the directory to the path provided. If the `overwrite` flag is set to false and a directory or file with the name specified in `copyPath` exists the call will be rejected. If the flag `isRelativePath` is set to true the `copyPath` is expected to continue from the Directory objects path, otherwise the `copyPath` will be assumed to contain an absolute path.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

fetch

Signature: `fetch()`

Refreshes the contents of the Directory object.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.6.2 File Object

Description

The File object encapsulates a file in the controller's filesystem. The object can be used to retrieve the contents of the file.

Methods

`getProperties`

Signature: `getProperties()`

Gets the properties of the directory.

Return value

A Promise with an object containing the file's properties defined as;

(string) name	the file's name
(Date) created	the creation date
(Date) modified	the last modification date
(number) size	the size of the file in bytes
(boolean) isReadOnly	flag indicating if file is read only

`getContents`

Signature: `getContents()`

Gets the contents of the file.

NOTE! After the contents have been retrieved, the content type can be determined by calling the method `getContentType`.

Return value

A Promise with an object containing the file's contents.

Example

```
const addTextToFile = async function(textToAdd){
  try{
    let myFile = await RWS.FileSystem.getFile("$HOME/logs/log.txt");
    let fileContent = await myFile.getContents();
    fileContent += `${textToAdd}\n`;
    myFile.setContents(fileContent);
    myFile.save(true);
  }
  catch(error) {
    console.log(`Failed to write log information.\n${error}`);
  }
}
```

```
}
```

getContentType

Signature: `getContentType()`

Gets the type of the contents. This is the content type string from the HTTP response.

Return value

A string with the content type, or empty string if not available.

setContents

Signature: `setContents(any newContents)`

`newContents` the new contents of the file

Sets the contents of the file object. After this has been called, the method `save` is used to store the content in the file on the file system.

Return value

A boolean with the status of the call.

fileExists

Signature: `fileExists()`

Checks if the file exists on the file system.

Return value

A Promise with a boolean with the result.

save

Signature: `save(boolean overwrite, boolean isBinary = false)`

`overwrite` flag indicating whether to overwrite existing directory

`isBinary` flag indicating whether binary or text mode should be used

Saves the contents to a file on the file system.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

delete

Signature: `delete()`

Deletes a file on the file system.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

rename

Signature: `rename(string newName)`

`newName` the new name of the directory

Renames a file on the file system to the name specified in `newName`.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

copy

Signature: `copy(string copyPath, boolean overwrite, boolean isRelativePath = true)`

`copyPath` the path of the copy

`overwrite` flag indicating whether to overwrite existing file

`isRelativePath` flag indicating whether path is relative to original file or absolute

Copies the file to the path provided. If the `overwrite` flag is set to true and a file with the name specified in `copyPath` exists that file will be overwritten. If the flag `isRelativePath` is set to true the `copyPath` is expected to continue from the File objects path, otherwise the `copyPath` will be assumed to contain an absolute path.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
try {
    var file = RWS.FileSystem.createFileObject('$HOME/logs/log1.txt');
    await file.copy('backups/log1.txt', false, true);
} catch (error) {
    console.log(`Error when creating backup.\n${error}`);
}
```

Example

```
try {
    var file = RWS.FileSystem.createFileObject('$HOME/logs/log1.txt');
    await file.copy('$HOME/logs/backups/log1.txt', true, false);
} catch (error) {
    console.log(`Error when creating backup.\n${error}`);
}
```

fetch

Signature: `fetch()`

Refreshes the File object's content and content type.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.7 RWS.Elog Namespace

Description

The `RWS.Elog` namespace contains methods to read, handle and subscribe to controller events.

The namespace contains the following objects; Event and Domain. These are described closer in this section.

There are also a set of methods that are available directly in the `RWS.Elog` namespace.

Methods

`clearElogAll`

Signature: `clearElogAll()`

Clears all Elog domains on the controller.

NOTE! All events will be purged from the controller, and the operation cannot be undone.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

`clearElog`

Signature: `clearElog(number domainNumber)`

`domainNumber` the domain number

Clears a specified Elog domain on the controller.

NOTE! All events will be purged, and the operation cannot be undone.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

`getBufferSize`

Signature: `getBufferSize(number domainNumber)`

`domainNumber` the domain number

Gets the buffer size of a domain. The buffer size is the maximum number of events possible to store for the specified domain.

Return value

A Promise with a number.

getNumberOfEvents

Signature: `getNumberOfEvents(number domainNumber)`

`domainNumber` a domain number

Gets the actual number of events for a domain.

Return value

A Promise with a number.

getEvents

Signature: `getEvents(number domainNumber, string language = 'en')`

`domainNumber` the domain number

`language` a language specifier, valid values: 'en', 'de', 'es', 'sv', ...

Gets all events for a specified domain in a specified language.

To allow faster download, the Event objects' data must be retrieved by calling their `getContents()` method.

Return value

A Promise with a dictionary (JavaScript Object) of Event objects, where the key is the sequence number of the event.

Example

```
var commonEvents = await RWS.Elog.getEvents(0, 'de');
var events = [];
for (let item in commonEvents) {
    try{
        var e = await res[item].getContents();
        events.push(e);
    } catch(error){
        console.warn(`Item '${item}' failed to load.\n${error}`);
    }
}
```

getEventsPaged

Signature: `getEventsPaged(number domainNumber, string language = 'en', number count = 50, number page = 1)`

`domainNumber` the domain number

`language` a language specifier, valid values: 'en', 'de', 'es', 'sv', ...

`count` the number of events per page

page the page to get

Gets the specified number of events for a specified domain in a specified language. The events returned will start at the page number specified, where 1 is the first page.

To allow faster download, the Event objects' data must be retrieved by calling their `getContents()` method.

Return value

A Promise with a dictionary (JavaScript Object) of Event objects, where the key is the sequence number of the event.

getEvent

Signature: `getEvent(number sequenceNumber, string language = 'en')`

sequenceNumber the sequence number of the event

language a language specifier, valid values: 'en', 'de', 'es', 'sv', ...

Creates an Event object for a specified event, setting its preferred language but without retrieving its data.

Return value

An Event object.

getDomain

Signature: `getDomain(number domainNumber)`

domainNumber the domain number

Creates a Domain object for a specified domain.

Return value

A Domain object.

4.7.1 Event Object

Description

The Event object stores the information about an event in a language specified. The data is retrieved only when the `getContents()` method is called. As the information is static subsequent calls to the `getContents()` method will not retrieve any data but return the information already held.

Methods

getContents

Signature: `getContents()`

Gets the data for an event. First time this method is called for the Event it will retrieve the data from the controller, subsequent calls returns the stored values.

Return value

A Promise with an object containing:

(number) <code>sequenceNumber</code>	events sequence number
(string) <code>eventType</code>	type of event, valid values: 'informational' 'warning' 'error'
(Date) <code>timeStamp</code>	time of event
(number) <code>code</code>	event's code field
(string) <code>title</code>	event's title field
(string) <code>description</code>	event's description field
(string) <code>consequences</code>	event's consequences field
(string) <code>causes</code>	event's causes field
(string) <code>actions</code>	event's suggested actions field
([object]) <code>arguments</code>	event's optional information, each object contains (string) <code>type</code> (string number) <code>value</code>

isValid

Signature: `isValid()`

Verifies if the data in the Event is retrieved and complete. Return true if data is ready to use, otherwise false.

Return value

A boolean.

4.7.2 Domain Object

Description

The Domain object represents an Elog domain on the controller. It is used to access information and retrieve and subscribe to events.

Methods**getTitle**

Signature: `getTitle()`

Used internally. Gets the title (part of a URL) of the subscribed resource.

Return value

A string containing the signal's title.

getDomainNumber

Signature: `getDomainNumber()`

Gets the domain number of the Domain object.

Return value

A number.

clearElog

Signature: `clearElog()`

Clears the domain on the controller.

NOTE! All events will be purged, and the operation cannot be undone.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

getBufferSize

Signature: `getBufferSize()`

Gets the buffer size of the domain. The buffer size is the maximum number of events possible to store for the specified domain.

Return value

A Promise with a number.

getNumberOfEvents

Signature: `getNumberOfEvents()`

Gets the actual number of events for the domain.

Return value

A Promise with a number.

getEvents

Signature: `getEvents(string language = 'en')`

language a language specifier, valid values: 'en', 'de', 'es', 'sv', ...

Gets all events for the domain in a specified language.

To allow faster download, the Event objects' data must be explicitly retrieved by calling their `getContents()` method.

Return value

A Promise with a dictionary (JavaScript object) of Event objects, where the key is the sequence number of the event.

`getEventsPaged`

Signature: `getEventsPaged(string language = 'en', number count = 50, number page = 1)`

`language` a language specifier, valid values: 'en', 'de', 'es', 'sv', ...

`count` the number of events per page

`page` the page to get

Gets the specified number of events for the domain in a specified language. The events returned will start at the page number specified, where 1 is the first page.

To allow faster download, the Event objects' data is not retrieved and must be retrieved by calling their `getContents()` method.

Return value

A Promise with a dictionary (JavaScript object) of Event objects, where the key is the sequence number of the event.

`getResourceString`

Signature: `getResourceString()`

Gets a unique resource URL for subscription.

Return value

A string containing a unique resource URL (as used by RWS.)

`addCallbackOnChanged`

Signature: `addCallbackOnChanged(function callback)`

`callback` a callback function to run when event occurs

Adds a callback function which will be called when the subscribed resource's state changes.

The callback can have one argument, which will contain the sequence number of the new event.

Return value

No return value.

`onchanged`

This is a function for internal use only. Do not use from external scripts.

subscribe

Signature: `subscribe()`

Starts the subscription to new events on the domain.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

Example

```
var myErrorCount = 0;
var elogDomain = null;

...

try{
    elogDomain = RWS.Elog.getDomain(RWS.Elog.DomainId.user);
    elogDomain.addCallbackOnChanged((newValue) => {
        let event = await RWS.Elog.getEvent(newValue);
        let contents = await event.getContents();
        if (contents.title === 'My Severe Error') {
            myErrorCount++;
        }
    });
    await elogDomain.subscribe();
} catch(error) {
    console.log(`Error occurred while subscribing to user elog.\n${error}`);
}
```

unsubscribe

Signature: `unsubscribe()`

Stops the subscription to the resource.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

4.8 RWS.UAS Namespace

Description

The `RWS.UAS` namespace, contains methods to read information in the user authentication system.

The namespace does not contain any objects, only a collection of methods.

Methods**getUser**

Signature: `getUser()`

Gets basic information of the user currently logged in and using the app.

Return value

A Promise with an object containing:

<code>(string) alias</code>	the user's alias
<code>(string) name</code>	the user's name
<code>(string) locale</code>	the locale, possible values; 'internal', 'local' or 'remote'
<code>(string) application</code>	the name of the application that is executing this method
<code>(string) location</code>	the physical location of the user

getGrants

Signature: `getGrants()`

Gets a list of available grants on the system.

Return value

A Promise with a dictionary (JavaScript object) where the key is the grant's name (reference below) and the value is an object containing:

<code>(string) reference</code>	a name used for referring to this grant, e.g., when using the <i>hasGrant</i> method described below
<code>(string) name</code>	a reference to the grant's name element in the localized xml file (<code>uastext.xml</code> which is part of RobotWare and installed on the system)
<code>(string) description</code>	a reference to the grant's description element in the localized xml file

hasGrant

Signature: `hasGrant(string grant)`

`grant` the grant to verify

Verifies if the user has a specific grant.

Return value

A Promise with a boolean indicating whether the user has the grant.

hasRole

Signature: `hasRole(string role)`

`role` the role to verify

Verifies if the user has a specific role.

Return value

A Promise with a boolean indicating whether the user has the role.

Example

Let's assume a role 'CanUseMyApp' exists on the controller and is required to run the app.

```
try {
  var hasRole = await RWS.UAS.hasRole('CanUseMyApp');
  if (hasRole === false) {
    var userInfo = await RWS.UAS.getUser();
    console.log(`User '${userInfo.name}' is not authorized to use app.`);
    myShutdownRoutine();
  }
} catch(error) {
  console.log(`Error while checking role, ${error}`);
  myShutdownRoutine();
}
```

4.9 RWS.Mastership Namespace

Description

The `RWS.Mastership` namespace contain methods for requesting and releasing controller mastership (write access).

Normally, other parts of the RWS client library handle mastership automatically on a per-request basis. However, in some situations, there might be need for explicitly taking mastership, performing a series of operations, and releasing mastership again.

The mastership namespace uses a counter mechanism for keeping track of the number of requests that has been made from different parts of the code. As long as there are more requests than releases, mastership is kept. This means that it is important to release mastership when done with a sequence of operations.

NOTE: The methods below only handles **edit** mastership, which means that they cannot be used for motion related mastership.

Methods

request

Signature: `request()`

Request *edit* mastership (write access) over the controller.

If mastership is already taken, the number of times the `release` method must be called to release mastership is increased by one.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

release

Signature: `release()`

Release *edit* mastership (write access) over the controller.

If mastership has been requested several times, an equal number of release calls must be performed.

Return value

A Promise which is empty if it resolves and contains a status if it is rejected.

5 Components Library

5.1 Introduction

General information

The Components Library provides reusable HTML/CSS/JavaScript graphical components that can be re-used in an application for a look-and-feel like the FlexPendant shell, and for consistency between different user applications.

The Components Library is not a framework, i.e. it consists of a JavaScript API that can be called from the application code, not the other way around, as is the case for a typical framework. This means that it can be combined with or integrated with many other HTML/JavaScript frameworks or tools.

The application developer still needs to take care of the general layout, navigation and logic of the application – using basic common web technologies or by using a framework fit for the purpose.

Usage of the Components Library is entirely optional.

Installation

The Components Library files should be placed in a subdirectory under the application web root (where the `appinfo.xml` file is located) named `fp-components`.

Working with the components

Most components (but not all) works in a similar way. First, a handle object is created using JavaScript. This handle is used for manipulating the component, both at the time of its creation, and during the lifetime of the application.

The handle object typically has an `attachToId` and an `attachToElement` function, which is used to insert the component as a sub-tree of elements under an element, preferably an empty DIV, specified by the user using an element id or a direct reference to the element.

As a basic example, here is very small but complete application using a single button.

```
<!DOCTYPE html>
<html>
<head>
  <!-- Always use UTF-8 as character set! -->
  <meta charset="UTF-8">

  <!-- Import the common resources, this should always be
  done when using the Component library, and always
  before loading any other Component library resources. -->
  <script src="fp-components/fp-components-common.js"></script>

  <!-- Import the Button component resource. -->
  <script src="fp-components/fp-components-button-a.js"></script>
```



```
<script>

    var myButton;

    // Always wait until all resources are loaded
    // before using the library.
    window.addEventListener("load", function() {

        // Create a new Button component handle.
        myButton = new FPComponents.Button_A();
        // Set the text of the button.
        myButton.text = "Click me!";
        myButton.onclick = ()=>{
            // Add code here that should
            // run when the user clicks
            // on the button..
            // Here, we update the text
            // on the button.
            myButton.text = "Clicked!";
        };
        // Insert the button as elements under
        // the "myDiv" DIV element defined below.
        myButton.attachToId("myDiv");
    });
</script>
</head>

<body>
    <!-- This DIV element will be used as the root for
    the button. -->
    <div id="myDiv"></div>
</body>
</html>
```

In general, always use an empty DIV element as the base for an component.

Depending on the component, in some cases it makes sense to set the styles of this DIV itself, especially the `width` CSS attribute.

5.2 Common Resources

Common resources

All components described in the sections below requires functionality located in a common file. This file should always be imported before any other files from the Components Library:

```
<script src="fp-components/fp-components-common.js"></script>
```

5.3 Standard Colors

Using the standard colors

The component library includes a set of predefined colors. According to design guidelines (see separate document), since a limited palette is generally desirable, try to use these colors to the largest possible extent.

The colors are defined as CSS variables and are available after loading the common resources:




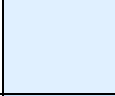












```
<script src="fp-components/fp-components-common.js"></script>
```

To use a color, use the `var` syntax in CSS. An example for setting the background color in a CSS rule:

```
background-color: var(--fp-components-GRAY-10);
```

List of available standard colors

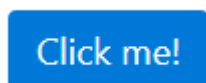
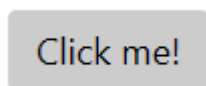
<code>--fp-color-STD-BACKGROUND</code>		White is used as the standard background color.
<code>--fp-color-PRIMARY-TEXT</code>		Use primary text color for all standard text, as well as headers.
<code>--fp-color-SECONDARY-TEXT</code>		The secondary text color can be used on a label when next to a data field.
<code>--fp-color-GRAY-10</code>		
<code>--fp-color-GRAY-20</code>		
<code>--fp-color-GRAY-30</code>		
<code>--fp-color-GRAY-40</code>		
<code>--fp-color-GRAY-50</code>		
<code>--fp-color-GRAY-60</code>		
<code>--fp-color-GRAY-70</code>		
<code>--fp-color-GRAY-80</code>		

--fp-color-GRAY-90		
--fp-color-GRAY-100		
--fp-color-BLUE-PRIMARY-HIGHLIGHT		The primary highlight color is used for highlighting. Things to highlight can be components and other elements. Same as --fp-color-BLUE-60
--fp-color-BLUE-10		
--fp-color-BLUE-20		
--fp-color-BLUE-30		
--fp-color-BLUE-40		
--fp-color-BLUE-50		
--fp-color-BLUE-60		
--fp-color-BLUE-70		
--fp-color-BLUE-80		
--fp-color-BLUE-90		
--fp-color-BLUE-100		
--fp-color-GREEN-OK		Status color: OK.
--fp-color-YELLOW-WARNING		Status color: Warning.
--fp-color-RED-DANGER		Status color: Danger / Error.

5.4 Button

Description

The Button component implements a simple clickable button. It is available in two variants – one basic (gray) and one highlighted (blue).



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-button-a.js"></script>
```

Example

```
var myButton = new FPComponents.Button_A();
myButton.text = "Click me!";
myButton.onclick = function () {
    // This function will execute when
    // the button is clicked.
};
myButton.attachToId("myDiv");
```

Handle constructor

```
var myButton = new FPComponents.Button_A();
```

Creates a handle to a new button.

Handle attributes

onclick

Type: `function ()`

Set this attribute to a callback function that should be called when the button is clicked.

enabled

Type: `boolean`

Setting this attribute to `false` will gray out the button and make it ignore any clicks. Setting this attribute to `true` (default) will enable the button, making it clickable.

text

Type: `string`

Set this attribute to the text that should be displayed on the button.

highlight

Type: `boolean`

Setting this attribute to `true` will make the button highlighted, i.e. *blue*. This is useful for marking an important button, having it stand out amongst other buttons.

Setting this attribute to `false` (default) will make the button use a normal appearance, i.e. *gray*.

icon

Type: `string`

Set this attribute to a string representing a path to an image to be used in order to display an icon to the left of the button text. If set to null (default) or empty string, the icon will not be displayed.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods**attachToId**

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.5 Checkbox

Description

The Checkbox component implements a selectable checkbox.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-checkbox-a.js"></script>
```

Example

```
var myCheckbox = new FPComponents.Checkbox_A();
myCheckbox.onclick = function (checked) {
    // This function will execute when
    // the checkbox is clicked.
    // The 'checked' argument contains a boolean
    // indicating whether the checkbox has been
    // checked (true) or unchecked (false).
};
myCheckbox.attachToId("myDiv");
```

Handle constructor

```
var myCheckbox = new FPComponents.Checkbox_A();
```

Creates a handle to a new checkbox.

Handle attributes

onclick

Type: `function (boolean checked)`

Set this attribute to a callback function that should be called when the checkbox is clicked. The function should accept an argument, *checked*, which will be set to `true` if the checkbox was checked or `false` if it was unchecked. See example above.

enabled

Type: `boolean`

Setting this attribute to `false` will gray out the checkbox and make it ignore any clicks. Setting this attribute to `true` (default) will enable the button, making it clickable.

checked

Type: `boolean`

Set this attribute to `true` to check the checkbox, or `false` to uncheck it.

Setting this attribute directly will not trigger a callback to the `onclick` function.

scale

Type: `number`

Set this attribute to change the visual scale of the component.

A value of `1.0` corresponds to the "normal" base size used by the FlexPendant shell. A value of `1.3` means 30% larger, `2.0` double the size, and so on.

desc

Type: `string`

Set this attribute to a string in order to display that string as a description label to the right of the component. Clicking the description label will act like clicking the actual checkbox. If set to null (default) or empty string, the label will not be displayed.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods**attachToId**

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

`attachToElement`

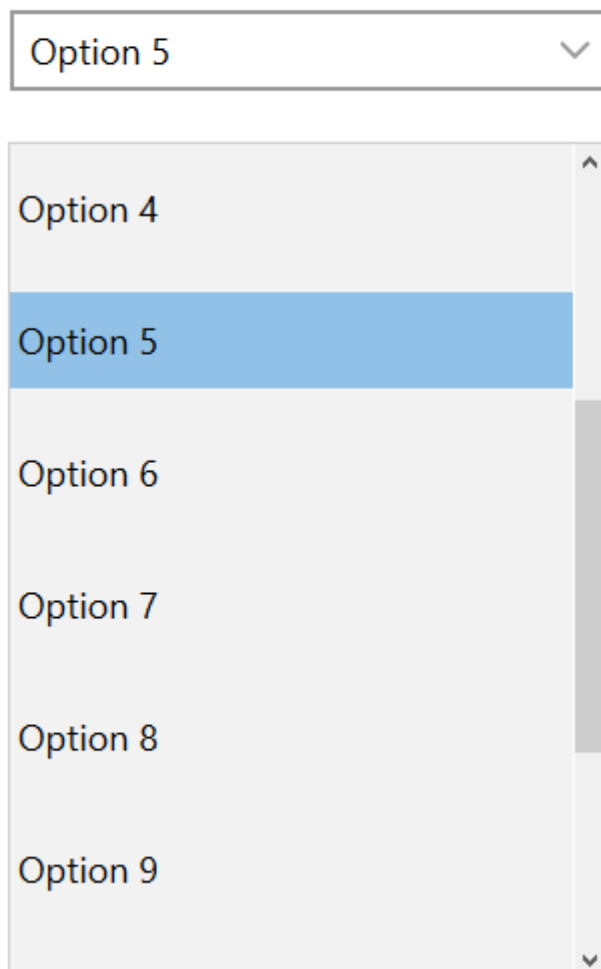
Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.6 Dropdown

Description

The Dropdown component implements a dropdown menu (also known as "combo-box"), allowing the user to select between available options.



Note: The scroll bar above is only visible on the virtual FlexPendant.

Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-dropdown-a.js"></script>
```

Example

```
var myDropdown = new FPComponents.Dropdown_A();
myDropdown.model = {
    items: [
        "Option 1",
        "Option 2",
        "Option 3",
        "Option 4",
        "Option 5",
        "Option 6",
        "Option 7",
        "Option 8",
        "Option 9",
        "Option 10",
        "Option 11",
        "Option 12"
    ]
};
myDropdown.selected = 5;
myDropdown.onselection = function (index, obj) {
    // This function will execute when
    // an item in the dropdown menu is
    // selected by the user.
}
myDropdown.attachToId("myDiv");
```

Handle constructor

```
var myDropdown = new FPComponents.Dropdown_A();
```

Creates a handle to a new dropdown menu.

Handle attributes

onselection

Type: `function (number index, object obj)`

Set this attribute to a callback function that should be called when the user selects an option in the dropdown menu. The function should accept two arguments, *index*, which will contain the selected index, and *obj*, which will contain the selected object from the model.

Indexing starts from 0.

See example above and the description for the `model` attribute.

enabled

Type: `boolean`

Setting this attribute to `false` will gray out the dropdown menu and make it ignore any clicks. Setting this attribute to `true` (default) will enable the dropdown menu, making it clickable.

leftTruncate

Type: `boolean`

If the text of the selected item does not fit within the component's width, the text will be clipped/truncated, and "..." will be added, either on the left side or the right side of the selected item's text. Setting this attribute to `false` (default) will truncate the selected item's text on the right side. Setting this attribute to `true` will truncate the selected item's text on the left side.

selected

Type: `number`

This attribute represents the currently selected index. It can be set to a new index to programmatically change selection. This will not cause the `onselection` callback to trigger.

It is possible to change selection while the menu part of the dropdown is shown. In this scenario, the selection will visibly change while the menu is kept open.

This attribute can have the value `null` when no selection is active.

model

Type: `object`

This attribute represents the available options in the dropdown menu.

The object should contain an attribute named `items`, which should contain a list of available options, where each option is represented by an object of any kind.

The string representation of each such object will be used when displaying the options within the menu. The option object will also be sent to the `onselection` callback function when the user has selected an option.

A typical option object could be a simple `string`, or a more complex custom object with a custom `toString` method.

Setting the model attribute will automatically refresh the dropdown menu options:

```
myDropdown.model = { ... some model ... }
```

... will cause a refresh.

However, simply modifying the current model:

```
myDropdown.model.items[0] = "New option title";
```

... will *not* cause a refresh, since the component will not notice that the model has changed. To force a refresh, set the model to itself after modifying it:

```
myDropdown.model.items[0] = "New option title";
```

```
myDropdown.model = myDropdown.model;
```

Example model object:

```
{
  items: [
    "My first option",
    "My second option"
  ]
}
```

desc

Type: `string`

Set this attribute to a string in order to display that string as a description label above the component. If set to null (default) or empty string, the label will not be displayed.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (`elementId`).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.7 Input

Description

The Input component implements an input field with an attached on-screen keyboard.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-input-a.js"></script>
```

Example

```
var myInput = new FPComponents.Input_A();
myInput.text = "Enter text here";
myInput.onchange = function ( text ) {
    // This function will execute when
    // the text has been modified by
    // the user.
}
myInput.attachToId("myDiv");
```

Handle constructor

```
var myInput = new FPComponents.Input_A();
```

Creates a handle to a new input field.

Handle attributes**onchange**

Type: `function (string text)`

Set this attribute to a callback function that should be called when the input field is modified by the user. The function should accept an argument, `text`, which will be set to the new text content. See example above.

enabled

Type: `boolean`

Setting this attribute to `false` will gray out the input field and make it ignore any clicks. Setting this attribute to `true` (default) will enable the input field, making it possible for the user to click on it and change the value.

text

Type: `string`

This attribute represents the text content of the input field. It can be read to get the current content. Setting it will programmatically update the content and will not trigger the `onchange` callback function to be called.

label

Type: `string`

Descriptive label string that will be visible below the editor field on the keyboard when the input field is being edited. Should describe the value that the user is editing, preferably including any input limitations.

desc

Type: `string`

Set this attribute to a string in order to display that string as a description label above the component. If set to null (default) or empty string, the label will not be displayed.

regex

Type: Regular expression object

Standard JavaScript regular expression object used for validating and allowing the input.

Example:

```
myInput.regex = /^-?[0-9]+(\.[0-9]+)?$/;
```

will only allow input of floating-point numbers or integers.

Default value is `null`.

Can be used in combination with the validator argument.

Note that if the text of the input field is set programmatically using the `text` attribute, this input restriction does not apply.

validator

Type: `function (string val)`

A callback function that will execute whenever the user is altering one or more characters in the keyboard's editable field.

The function signature should be:

`function (val)`

The `val` argument will be the current value (string) as it is being edited by the user. The callback function should return `true` when the value is acceptable and `false` when not acceptable.

Default value is `null`.

Can be used in combination with the `regex` argument.

Note that if the text of the input field is set programmatically using the `text` attribute, this input restriction does not apply.

variant

Type: `number`

Initial keyboard variant to be shown. Possible values are

`FP_COMPONENTS_KEYBOARD_ALPHA` or `FP_COMPONENTS_KEYBOARD_NUM`.

Default value is `null`.

NOTE! *This property is currently ignored and will not affect the appearance of the keyboard. This is due to limitations in built-in touch keyboard. It is possible that it will work in a future release.*

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.8 Radio Button

Description

The Radio button component implements a selectable radio button.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-radio-a.js"></script>
```

Example

The following example shows two radio buttons, connected through callbacks.

```
var myRadioButton1 = null;
var myRadioButton2 = null;

myRadioButton1 = new FPComponents.Radio_A();
myRadioButton1.onclick = function () {
    // Uncheck the other radio button
    myRadioButton2.checked = false;
};
myRadioButton1.checked = true;
```

```
myRadioButton1.attachToId("myDiv1");

myRadioButton2 = new FPComponents.Radio_A();
myRadioButton2.onclick = function () {
    // Uncheck the other radio button
    myRadioButton1.checked = false;
};
myRadioButton2.attachToId("myDiv2");
```

Handle constructor

```
var myRadioButton = new FPComponents.Radio_A();
```

Creates a handle to a new radio button.

Handle attributes

onclick

Type: `function ()`

Set this attribute to a callback function that should be called when the radio button is checked by the user. This callback function will not be called when the user clicks on a radio button that is already checked. See example above.

enabled

Type: `boolean`

Setting this attribute to `false` will gray out the radio button and make it ignore any clicks. Setting this attribute to `true` (default) will enable the radio button, making it clickable.

checked

Type: `boolean`

Set this attribute to `true` to check the radio button, or `false` to uncheck it.

Setting this attribute directly will not trigger a callback to the `onclick` function.

scale

Type: `number`

Set this attribute to change the visual scale of the component.

A value of `1.0` corresponds to the "normal" base size used by the FlexPendant shell. A value of `1.3` means 30% larger, `2.0` double the size, and so on.

descType: `string`

Set this attribute to a string in order to display that string as a description label to the right of the component. Clicking the label will act like clicking the actual radio button. If set to null (default) or empty string, the label will not be displayed.

parentType: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods**attachToId**Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElementSignature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.9 Switch Button

Description

The Switch button component implements a selectable switch button.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
```

```
<script src="fp-components/fp-components-switch-a.js"></script>
```

Example

```
var mySwitch = new FPComponents.Switch_A();  
mySwitch.onchange = function (active) {  
    // This function will execute when  
    // the switch has been clicked by  
    // the user.  
}  
mySwitch.attachToId("myDiv");
```

Handle constructor

```
var mySwitch = new FPComponents.Switch_A();
```

Creates a handle to a new switch button.

Handle attributes

onchange

Type: function (boolean active)

Set this attribute to a callback function that should be called when the switch button is modified by the user. The function should accept an argument, `active`, which will be set to `true` when the switch has been activated and `false` when it has been deactivated. See example above.

enabled

Type: boolean

Setting this attribute to `false` will gray out the switch button and make it ignore any clicks. Setting this attribute to `true` (default) will enable the switch button, making it clickable.

active

Type: boolean

Set this attribute to `true` to make the switch button activated, or `false` to deactivate it. Setting this attribute directly will not trigger a callback to the `onchange` function.

scale

Type: `number`

Set this attribute to change the visual scale of the component.

A value of `1.0` corresponds to the "normal" base size used by the FlexPendant shell. A value of `1.3` means 30% larger, `2.0` double the size, and so on.

desc

Type: `string`

Set this attribute to a string in order to display that string as a description label to the right of the component. Clicking the label will act like clicking the actual switch button. If set to null (default) or empty string, the label will not be displayed.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods**attachToId**

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

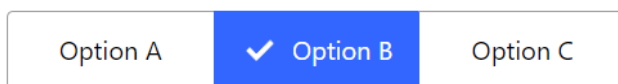
Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.10 Toggle

Description

The toggle component works as a group of buttons that allows the user to select only one or multiple values, depending on configuration of toggle component.

Each item in the toggle component can be configured separately to have different text, and different icons for both toggled and untoggled states.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-toggle-a.js"></script>
```

Example

The following example creates a toggle component with three items. The first item has no icons, the second item has one icon for untoggled and a different one for toggled state. The last item has the same icon for both untoggled and toggled states.

```
var myToggle = new FPComponents.Toggle_A();
myToggle.model = [
  { text: "Option A" },
  { text: "Option A", icon: "option.png", activeIcon: "check.png" },
  { text: "Option A", icon: "option.png" }
]
myToggle.attachToId("myDiv");
```

Handle constructor

```
var myToggle = new FPComponents.Toggle_A();
```

Creates a handle to a new toggle component.

Handle attributes

multi

Type: `boolean`

Set this attribute to `true` to allow selection of multiple options in toggle. When set to true will ignore `singleAllowNone` setting, since all toggles will be toggled independently from each other.

Set this attribute to `false` (default) to allow selection of only one option at a time.

singleAllowNone

Type: `boolean`

Set this attribute to `true` to allow removing all selections and leaving toggle component without selected option.

Set this attribute to `false` (default) to force at least one option to be selected. Only works if **multi** is set to false as well.

onclick

Type: `function (object state)`

Set this attribute to a callback function that should be called when any button in the component is toggled. This callback function will not be called when the user clicks on a button that does not change state, for example when clicking already toggled button and `singleAllowNone` is true. The function will return a state object containing which buttons changed and what they changed to, as well as current state of all buttons.

Example of state object:

```
{
  all: [false, true, false],
  changed: [ [1, true], [2, false] ]
}
```

the **all** property represents all toggle buttons and their current state, so in the example above the first button is untoggled, second is toggled and third is untoggled.

The **changed** property represents what changed when the user clicked a button, it's an array containing one array per changed button. Each button array with index of button changed, and what state it changed to. In the example above it says that button with index 1 changed to true, and button with index 2 changed to false, which can be verified in the all array as well.

model

Type: `object`

This attribute represents the buttons in the toggle component.

Contains a list of objects where each object represents a button.

The following attributes can be used on each element object:

Attribute	Description
<code>text</code>	Mandatory attribute. Text displayed on button
<code>icon</code>	Optional attribute. The path (relative to the web root, i.e. where the WebApp manifest is located) to an image to be displayed on the button. The optimal size for the icon image is 24x24 pixels, but it will be automatically resized.
<code>toggledIcon</code>	Optional attribute. Same requirements as icon attribute. When set will display the icon only when button is in toggled state. Will override icon attribute when button is toggled if both are set.

Methods

`attachToId`

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

`attachToElement`

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

`setToggled`

Signature: `setToggled (number index, boolean toggled, boolean noLimitations=false)`

Will set button with specified index to the provided toggled state. This operation will not cause onclick callbacks to be fired when used.

By default, this method will follow the limitations on toggled state (`multi`, `singleAllowNone`). Setting the optional `noLimitations` argument to `true` will force the toggle regardless of usage limitations.

getToggledList

Signature: `getToggledList ()`

Gets array containing state of all buttons in order.

isToggled

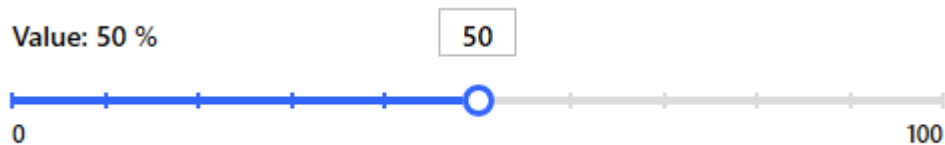
Signature: `isToggled (number index)`

Returns true if button with provided index is toggled and returns false otherwise. if index is not valid (out of range, or not a number) will return undefined.

5.11 Slider

Description

The Slider component implements an analog slider that displays and lets the user set a value. It is highly configurable and has support for labels and tick marks.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-slider-a.js"></script>
```

Example

```
var slider = new FPComponents.Slider_A();
slider.tickStep = 10;
slider.displayTicks = true;
slider.min = 0;
slider.max = 100;
slider.value = 20;
slider.enabled = true;
slider.label = "Value: ";
```

```
slider.unit = " %";
slider.width = 200;
slider.attachToId("slider-container-2");
slider.ondrag = function(value) {
  // Do something
}
slider.onrelease = function(value) {
  // Do something
}
slider.attachToId("slider-placeholder");
```

Handle constructor

```
var mySlider = new FPComponents.Slider_A();
```

Creates a handle to a new Slider.

Handle attributes

max

Type: `number`

Set this attribute to the upper bound of the slider.

Default value is 100.

min

Type: `number`

Set this attribute to the lower bound of the slider.

Default value is 0.

label

Type: `string`

This attribute represents the base content of the built-in label.

Default value is an empty string.

displayLabel

Type: `boolean`

his attribute decides if the built-in label should be displayed.

Default value is `true`.

displayValue

Type: `boolean`

Setting this attribute to `true` will display the current value in the built-in label.

Default value is `true`.

unit

Type: `string`

Set this attribute to the unit to be displayed after values.

This attribute is only used if the `displayValue` attribute is set to `true`.

Default value is an empty string.

value

Type: `number`

Set this attribute to change the current value of the slider.

Default value is `0`.

tickStep

Type: `number`

This attribute sets the frequency of the tick marks on the slider.

Default value is `1`.

displayTicks

Type: `bool`

Setting this attribute to `true` will display tick marks on the slider.

The frequency of the tick marks is set on the `tickStep` attribute.

Default value is `false`.

numberOfDecimals

Type: `number`

This attribute controls the number of decimals to displayed by the slider component.

Default value is `0`.

enabled

Type: `bool`

Setting this to `true` will enable the slider for input.

Setting this to `false` will disable the slider for input and indicate that it is disabled visually.

Default value is `true`.

ondrag

Type: `function (number value)`

Set this attribute to a callback function that should be called when thumb is moved.

The function should accept an argument, value which is the current value of the slider.

onrelease

Type: `function (number value)`

Set this attribute to a callback function that should be called when thumb is released.

The function should accept an argument, value which is the current value of the slider.

width

Type: `number`

Set this attribute to the desired width of the slider component, in pixels.

Note that this value is including space required for labels. This is handled automatically by the component.

Default value is 200.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

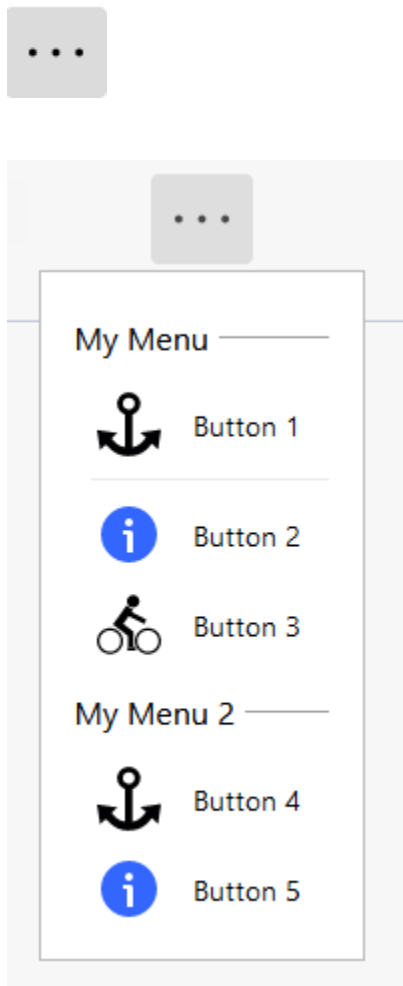
Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.12 Context Menu

Description

The Context menu component implements a context menu presenting the user with a configurable set of buttons.

The component takes the form of a single button that, when clicked, brings up the menu itself in close vicinity of the button.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-contextmenu-a.js"></script>
```

Example

```
var myContextMenu = new FPComponents.Contextmenu_A();
myContextMenu.model = {
  content: [
    {
      type: "label",
      label: "My Menu"
    },
    {
      type: "button",
      label: "Button 1",
      icon: "anchor.svg",
      onclick: function () {
        // This function will execute when
        // the button is pressed by
        // the user.
      },
      enabled: true
    },
    {
      type: "gap"
    },
    {
      type: "button",
      label: "Button 2",
      icon: "info.svg",
      onclick: function () {
        // This function will execute when
        // the button is pressed by
        // the user.
      },
      enabled: true
    },
    {
      type: "button",
```

```
        label: "Button 3",
        icon: "cyclist.svg",
        onclick: function () {
            // This function will execute when
            // the button is pressed by
            // the user.
        },
        enabled: true
    },
    {
        type: "label",
        label: "My Menu 2"
    },
    {
        type: "button",
        label: "Button 4",
        icon: "anchor.svg",
        onclick: function () {
            // This function will execute when
            // the button is pressed by
            // the user.
        },
        enabled: true
    },
    {
        type: "button",
        label: "Button 5",
        icon: "info.svg",
        onclick: function () {
            // This function will execute when
            // the button is pressed by
            // the user.
        },
        enabled: true
    },
]
};
```

```
myContextMenu.attachToId("MyDiv");
```

Handle constructor

```
var myContextMenu = new FPComponents.Contextmenu_A();
```

Creates a handle to a new Context Menu.

Handle attributes

model

Type: object

This attribute represents the available options in the context menu.

The object should contain an attribute named `content`, which should contain a list of menu items, where each item is represented by an object.

Setting the model attribute will automatically refresh the context menu options:

```
myContextMenu.model = { ... some model ... }
```

... will cause a refresh.

However, simply modifying the current model:

```
myContextMenu.model.content[0] = {type: "gap"};
```

... will *not* cause a refresh, since the component will not notice that the model has changed. To force a refresh, set the model to itself after modifying it:

```
myContextMenu.model.content[0] = {type: "gap"};
```

```
myContextMenu.model = myContextMenu.model;
```

Example model object:

```
{
  content: [
    {
      type: "label",
      label: "My Menu"
    },
    {
      type: "button",
      label: "Button 1",
      icon: "someimage.png",
      onclick: function () {
        // This function will execute when
```

```
        // the button is pressed by
        // the user.

    },
    enabled: true
}

]
```

The available attributes for the objects are:

Attribute	Description
type	Mandatory attribute. Can be either "label", "button" or "gap".
label	Valid for types "label" and "button" only. The text that should be displayed on the item.
onclick	Valid for type "button" only. A callback function that should be called when the button is clicked.
icon	Valid for type "button" only. The path (relative to the web root, i.e. where the WebApp manifest is located) to an image to be displayed as an icon on the button. The optimal size for the icon image is 32x32 pixels, but it will be automatically resized. Can be omitted.
enabled	Valid for type "button" only. If set to <code>true</code> (default), this attribute has no effect. If set to <code>false</code> , the button will be disabled, i.e. it will be "grayed-out" and cannot be clicked. Default value is <code>true</code> .

enabled

Type: `boolean`

Setting this attribute to `false` will gray out the button that brings up the context menu and make it ignore any clicks. Setting this attribute to `true` (default) will enable the button, making it possible to click.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.13 Digital Indicator

Description

The Digital indicator component implements an indicator that can show a digital state, typically for envisaging the state of a digital I/O signal.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-digital-a.js"></script>
```

Example

```
var myDigital = new FPComponents.Digital_A();
myDigital.onclick = function () {
    // This function will execute when
    // the digital indicator has been clicked by
```



```
// the user.  
myDigital.active = !(myDigital.active);  
}  
myDigital.attachToId("myDiv");
```

Handle constructor

```
var myDigital = new FPComponents.Digital_A();
```

Creates a handle to a new Digital indicator.

Handle attributes

onclick

Type: `function ()`

Set this attribute to a callback function that should be called when the digital indicator is clicked.

active

Type: `boolean`

Set this attribute to `true` to make the digital indicator activated (1), or `false` to deactivate it (0).

desc

Type: `string`

Set this attribute to a string in order to display that string as a description label to the right of the component. Clicking the label will act like clicking the actual digital indicator. If set to null (default) or empty string, the label will not be displayed.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.14 Analog Level Meter

Description

The Analog level meter component implements a value meter that can show an analog state, typically for envisaging the state of an analog I/O signal or other non-digital value sources.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-levelmeter-a.js"></script>
```

Example

```
var myLevelmeter = new FPComponents.Levelmeter_A();
myLevelmeter.min = 0;
myLevelmeter.max = 100;
myLevelmeter.lim1 = 50;
myLevelmeter.lim2 = 70;
myLevelmeter.unit = "°C";
myLevelmeter.width = 250;
myLevelmeter.value = 33;
myLevelmeter.attachToId("myDiv");
```

Handle constructor

```
var myLevelmeter = new FPComponents.Levelmeter_A();
```

Creates a handle to a new Analog level meter.

Handle attributes**width**

Type: `number`

Set this attribute to the desired component width in pixels.

Default is 200 pixels.

max

Type: `number`

Set this attribute to the upper bound of the meter.

Default is 100.

min

Type: `number`

Set this attribute to the lower bound of the meter.

Default is 0.

lim1

Type: `number`

Set this attribute to the value limit for the first (yellow) warning section.

Default is 100.

lim2

Type: `number`

Set this attribute to the value limit for the second (red) warning section.

Default is 100.

unit

Type: `string`

Set this attribute to the unit to be displayed after values.

Default is empty string.

value

Type: `number`

Set this attribute to the current value to be displayed.

Default is 0.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

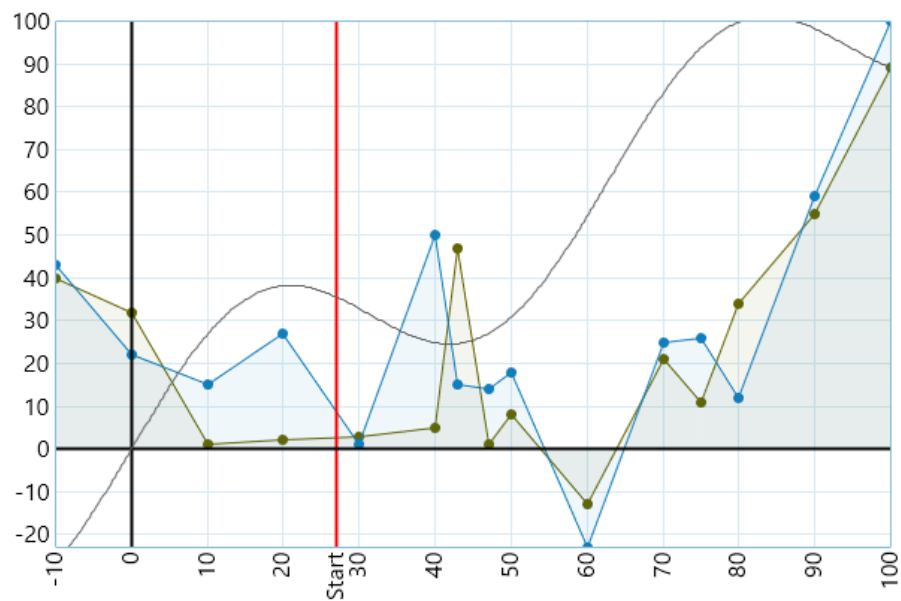
Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.15 Line Chart

Description

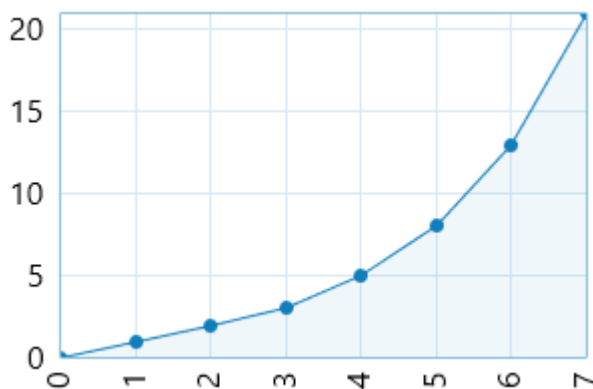
The Line chart component implements a configurable line chart with support for multiple 2-axis data series and mathematical functions.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-linechart-a.js"></script>
```

Example



```
var myLineChart = new FPComponents.Linechart_A();
myLineChart.width = 300;
myLineChart.height = 400;

myLineChart.model = [
```

```

{
    points: [
        [0, 0],
        [1, 1],
        [2, 2],
        [3, 3],
        [4, 5],
        [5, 8],
        [6, 13],
        [7, 21]
    ]
}

];

myLineChart.attachToId("myDiv");

```

Handle constructor

```
var myLineChart = new FPCComponents.Linechart_A();
```

Creates a handle to a new line chart.

Handle attributes

width

Type: `number`

Set this attribute to the desired width of the line chart component, in pixels.

Note that this value is including space required for labels. This is handled automatically by the component.

Default value is 640.

height

Type: `number`

Set this attribute to the desired height of the line chart component, in pixels.

Note that this value is including space required for labels. This is handled automatically by the component.

Default value is 480.

xMax

Type: `number`

Set this attribute to the upper limit for x-axis range. A value of `null` will cause the component to automatically set the limit.

Default value is `null`.

xMin

Type: `number`

Set this attribute to the lower limit for the x-axis range. A value of `null` will cause the component to automatically set the limit.

Default value is `null`.

yMax

Type: `number`

Set this attribute to the upper limit for y-axis range. A value of `null` will cause the component to automatically set the limit.

Default value is `null`.

yMin

Type: `number`

Set this attribute to the lower limit for the y-axis range. A value of `null` will cause the component to automatically set the limit.

Default value is `null`.

xStep

Type: `number`

Set this attribute to the size of each background column. A value of `null` will cause the component to automatically set the size. A value of 0 or less will disable background columns.

Default value is `null`.

yStep

Type: `number`

Set this attribute to the size of each background row. A value of `null` will cause the component to automatically set the size. A value of 0 or less will disable background rows.

Default value is `null`.

xLabels

Type: `array`

This attribute holds custom labels positioned on the x-axis. The model is an array containing any number of labels. Each label is represented by another array, containing the label text and the x value.

Example:

```
myLineChart.xLabels = [  
    ["A", 1],  
    ["B", 4.5]  
];
```

yLabels

Type: `array`

This attribute holds custom labels positioned on the y-axis. The model is an array containing any number of labels. Each label is represented by another array, containing the label text and the y value.

Example:

```
myLineChart.yLabels = [  
    ["A", 1],  
    ["B", 4.5]  
];
```

xAutoLabelStep

Type: `number`

Set this attribute to the distance between each automatic label on the x-axis. Preferably, this value should be a multiple of the `xStep` attribute.

A value of `null` will cause this attribute to be automatic, i.e. same as the `xStep` attribute.

A value of 0 or less will disable automatic labels on the x axis.

Note that mixing automatic labels and custom labels (using the `xLabels` attribute) might cause layout collisions. In this case, custom labels take precedence.

Default value is `null`.

yAutoLabelStep

Type: `number`

Set this attribute to the distance between each automatic label on the y-axis. Preferably, this value should be a multiple of the `yStep` attribute.

A value of `null` will cause this attribute to be automatic, i.e. same as the `yStep` attribute.

A value of 0 or less will disable automatic labels on the y axis.

Note that mixing automatic labels and custom labels (using the `yLabels` attribute) might cause layout collisions. In this case, custom labels take precedence.

Default value is `null`.

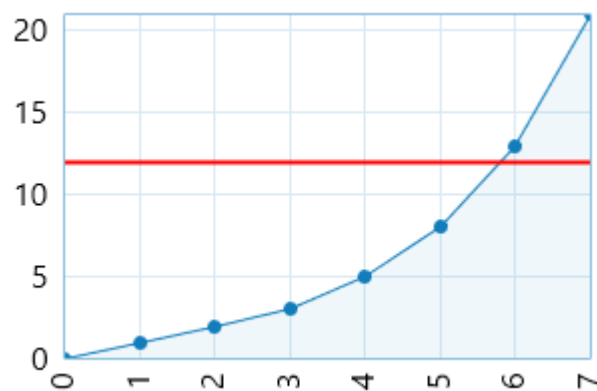
`model`

Type: `array`

This attribute represents the various items that will be displayed in the line chart, such as data sets and functions.

The model is an array which contains any number of objects. Each object contains various attributes which decide what should be displayed.

The following example contains two objects, a data set and a marker on the y axis:



```
myLineChart.model = [
  {
    // First object
    points: [
      [0,0],
      [1,1],
      [2,2],
      [3,3],
      [4,5],
      [5,8],
      [6,13],
      [7,21]
    ],
  },
  {
    // Second object
    yMarker: 12,
    thickness: 2,
    red: 255,
    green: 0,
  }
]
```

```

        blue: 0
    }
];

```

The available attributes for the objects are:

Attribute	Description
points	<p>A data set with any number of points. Represented by an array containing single points, also arrays, where the first element is the x value and the second the y value.</p> <p>Example:</p> <pre>points: [[102,673], // Point 1 [132,822], // Point 2 [155,223] // Point 3]</pre> <p>The points will be automatically sorted by x value.</p>
fill	<p>A boolean indicating whether a data set graph should be filled (true) or simply a line (false).</p> <p>Only valid together with the <code>points</code> attribute.</p> <p>Default value is <code>true</code>.</p>
dots	<p>A boolean indicating whether the individual points in a data set graph should be indicated by a small dots (true) or not (false).</p> <p>Only valid together with the <code>points</code> attribute.</p> <p>Default value is <code>true</code>.</p>
thickness	<p>A number representing the thickness of the data set graph line, marker or function graph line. The unit is pixels.</p> <p>Default value is 1.</p>
red, green, blue	<p>If all three color attributes are present, use this RGB color value when drawing the object. Each attribute is a number from 0 to 255.</p>
hidden	<p>A boolean indicating whether this object should be displayed (false) or not (true).</p>
xMarker	<p>One or more x values that should be marked with a straight line. Can be either a number or an array with several numbers.</p> <p>Examples:</p> <pre>xMarker : 17</pre> <p>or</p>

	<code>xMarker : [17, 28, 53]</code>
<code>yMarker</code>	One or more y values that should be marked with a straight line. Can be either a number or an array with several numbers.
<code>xFunc</code>	A function, $y = f(x)$. This function will be called repeatedly to draw a graph from <code>xMin</code> to <code>xMax</code> . The function should be of the signature <code>function (x)</code> and should return the y value for any given x value. Example: <pre>xFunc : function (x) { return Math.sin(x); }</pre>
<code>yFunc</code>	A function, $x = f(y)$. This function will be called repeatedly to draw a graph from <code>yMin</code> to <code>yMax</code> . The function should be of the signature <code>function (y)</code> and should return the x value for any given y value. Example: <pre>yFunc : function (y) { return Math.sin(y); }</pre>
<code>xFuncStep</code>	The resolution (x value step size) for resolving a <code>xFunc</code> function. Should be a positive number.
<code>yFuncStep</code>	The resolution (y value step size) for resolving a <code>yFunc</code> function. Should be a positive number.

Setting the model attribute will automatically refresh the line chart:

```
myLineChart.model = [ ... some model ... ]
```

... will cause a refresh.

However, simply modifying the current model:

```
myLineChart.model[2].hidden = true;
```

... will *not* cause a refresh, since the component will not notice that the model has changed. To force a refresh, set the model to itself after modifying it:

```
myLineChart.model[2].hidden = true;
```

```
myLineChart.model = myLineChart.model;
```

or call the `repaintLater()` method described below.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

repaintLater

Signature: `repaintLater ()`

Will repaint this component asynchronously at some point in the future.

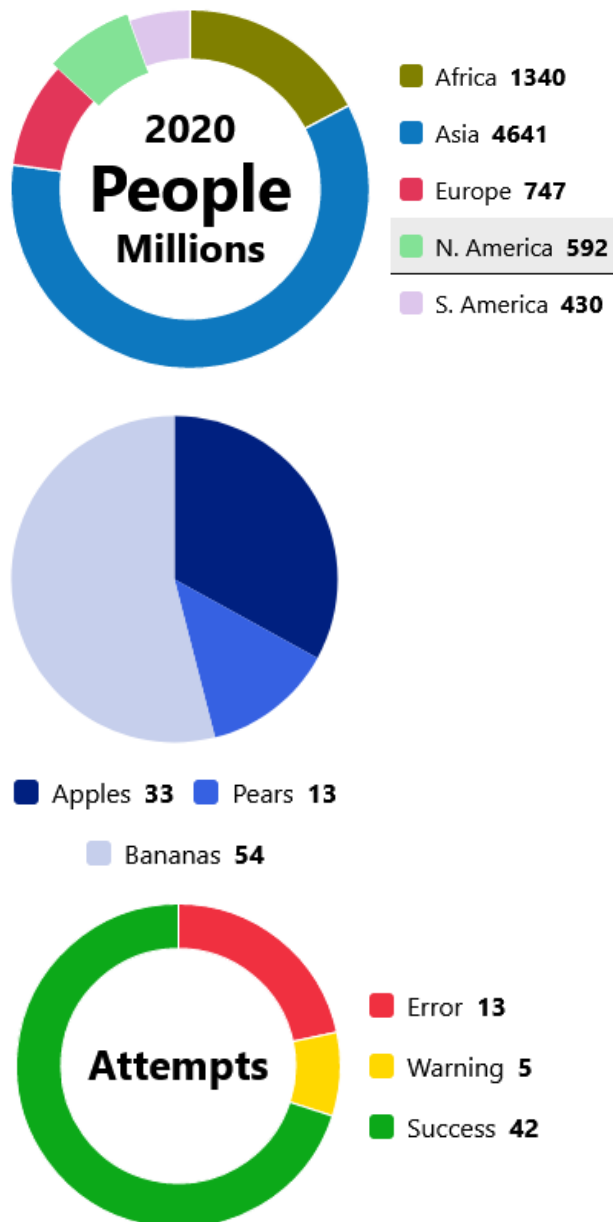
Most attribute changes will automatically cause this method to be called, but it can be used after changing the internals of complex attributes like `model` or `xLabels` without explicitly setting the attributes on the components handle.

Calling this method several times may result in a single update.

5.16 Pie Chart

Description

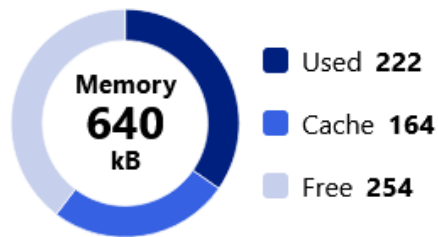
The Pie chart component implements a configurable pie- or donut chart with support for automatic color distribution and selectable sectors for easy reading of values.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-piechart-a.js"></script>
```

Example



```
var myDonut = new FPComponents.Piechart_A();

myDonut.model = [
    [222, 'Used' ],
    [164, 'Cache' ],
    [254, 'Free' ],
];

myDonut.topText = "Memory";
myDonut.centerText = "640";
myDonut.bottomText = "kB";
myDonut.size = 150;

myDonut.attachToId("myDiv");
```

Handle constructor

```
var myPieChart = new FPComponents.Piechart_A();
```

Creates a handle to a new pie/donut chart.

Handle attributes

size

Type: number

Set this attribute to the desired width/height of the graphical part of the component, in pixels.

Note that this value is not including space required for labels. The space required by the labels are dependent on number of labels and their names and values.

Default value is 300.

donut

Type: `boolean`

Set this attribute to `true` for the graphical part to use the Donut chart visual style. Set it to `false` to use the Pie chart style instead.



Default value is `true`.

showLabels

Type: `boolean`

Set this attribute to `true` for the labels to be visible. Set it to `false` to hide them.

Default value is `true`.

labelsBelow

Type: `boolean`

Set this attribute to `true` for the labels to be positioned underneath the graphical chart. Set it to `false` for the labels to be positioned on the side.

Default value is `false`.

model

Type: `Array`

This attribute represents the data to be displayed in the chart.

The model is an array of arrays where each inner array represents a single value.

The format of each inner array is either

[`<number value>`, `<string description>`]

or

[`<number value>`, `<string description>`, `<string color>`]

If the color is omitted, it will instead be automatically decided by the component. See the `hue` and `multiHue` attributes for more related information.

Example:

```
myDonut.model = [
    [222, 'Used' ],
    [164, 'Cache' ],
    [254, 'Free' ],
];
```

or

```
myDonut.model = [
    [222, 'Used',  '#FF2222' ],
    [164, 'Cache', '#22FF22' ],
    [254, 'Free',  '#2222FF' ],
];
```

Setting the `model` attribute will automatically refresh the chart:

```
myDonut.model = [ ... some model ... ]
```

... will cause a refresh.

However, simply modifying the current model:

```
myDonut.model[2][0] = 150;
```

... will *not* cause a refresh, since the component will not notice that the model has changed. To force a refresh, set the model to itself after modifying it:

```
myDonut.model[2][0] = 150;
```

```
myDonut.model = myDonut.model;
```

Default value is an empty array.

hue

Type: `number`

Unless the sector colors are specified directly, this attribute can be set to a hue value within the HSV/HSL color space, i.e., a value between 0 and 360 – the angle on the color wheel.

The hue will be used for automatically picking sector colors when no color is specified in `model`.

Some example hues:

0	Red	
30	Orange	
60	Yellow	

90	Chartreuse	
120	Green	
150	Emerald	
180	Cyan	
210	Azure	
240	Blue	
270	Violet	
300	Magenta	
330	Crimson	
360	Red	

Default value is

225	Cobalt blue	
-----	-------------	--

multiHue

Type: `boolean`

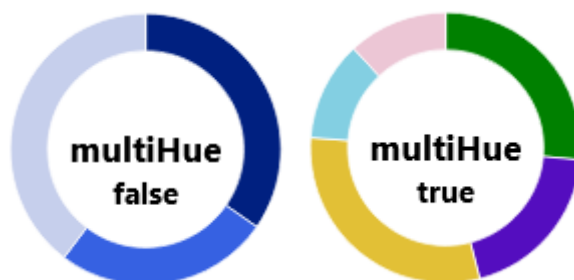
When no color is specified for sectors in `model`, the component will automatically select colors.

If this attribute is set to `false`, all auto-selected sector colors will have the same hue, specified by the `hue` attribute. They will differ only in saturation and lightness.

The single hue mode is works best when there are few sectors.

Setting the `multiHue` attribute to `true` enables the multi hue mode. In this mode, the sector hues will have different hues. The base hue (set using the `hue` attribute) will be used for the first sector, while other sectors will have other hues, as far away from each other as possible. Changing the `hue` attribute will affect all sectors.

The multi hue mode should typically be used when there are a larger number of sectors.



The default value is `false`.

centerText

Type: `string`

Set this attribute to a string that should be shown in the middle of the donut chart. This text is not visible in pie chart mode.

The text size is automatically adjusted to fit inside the donut.

Default value is an empty string.

topText

Type: `string`

Set this attribute to a string that should be above the center text of the donut chart. This text is not visible in pie chart mode.

The text size is automatically adjusted to fit inside the donut.

Default value is an empty string.

bottomText

Type: `string`

Set this attribute to a string that should be below the center text of the donut chart. This text is not visible in pie chart mode.

The text size is automatically adjusted to fit inside the donut.

Default value is an empty string.

selected

Type: `number`

This attribute represents the currently selected sector. The value is either `null` (for no selection) or the numeric index of a sector in the `model` array.

The initial value is `null`, but it will be changed when the user selects a sector.

onselection

Type: `function (number index, number value)`

Set this attribute to a function that should be called when the sector selection changes.

The function should accept two arguments, the index of the selected sector in the `model` array, and the numeric value of that sector.

Default value is `null`.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

`appendData`

Signature: `appendData (number value, string label, string color=null)`

Shorthand method for adding another sector to the chart. The sector will be added to the `model` array.

The arguments corresponds to the fields in each sector entry within the `model` array.

`clearData`

Signature: `clearData ()`

This method is equivalent to setting the `model` attribute to an empty array, i.e., the chart is cleared from sectors.

`attachToId`

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty `div` element. Do not attach the component to an element with existing children.

`attachToElement`

Signature: `attachToElement (Element element)`

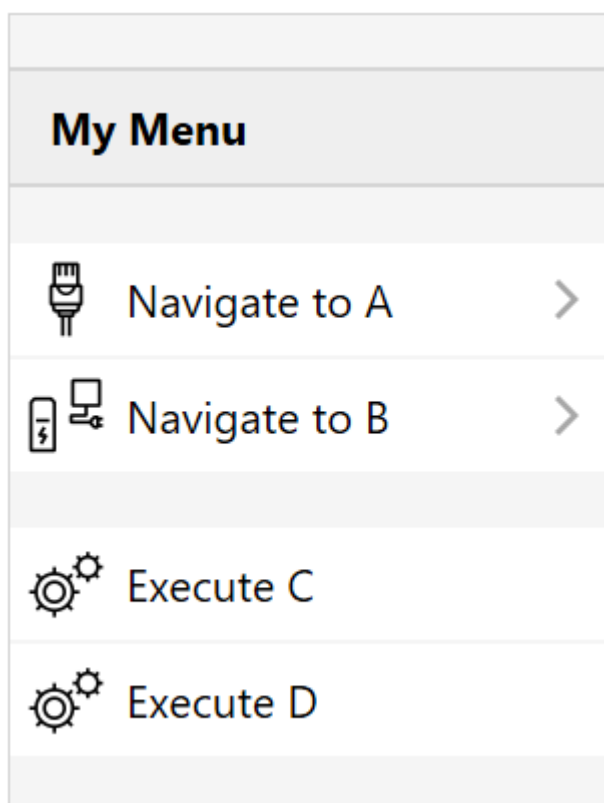
Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.17 Menu

Description

The `Menu` component implements a configurable menu, typically used for app navigation and main actions. It can be embedded in a `Fold-in` component (see below), some other custom component or simply put directly on the main view.

It has built-in support for buttons with optional icons, dividers and labels.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-menu-a.js"></script>
```

Example

```
var myMenu = new FPComponents.Menu_A();
myMenu.model = {
  content: [
    {
      type: "gap"
    },
    {
      type: "label",
      label: "My Menu"
    },
    {
```

```
        type: "gap"
    },
    {
        type: "button",
        label: "Navigate to A",
        icon: "icons/A.png",
        arrow: true,
        onclick: function () {
            // Code to execute
        }
    },
    {
        type: "button",
        label: "Navigate to B",
        icon: "icons/B.png",
        arrow: true,
        onclick: function () {
            // Code to execute
        }
    },
    {
        type: "gap"
    },
    {
        type: "button",
        label: "Execute C",
        icon: "icons/C.png",
        arrow: false,
        onclick: function () {
            // Code to execute
        }
    },
    {
        type: "button",
        label: "Execute D",
        icon: "icons/D.png",
        arrow: false,
```

```

        onclick: function () {
            // Code to execute
        }
    },
    {
        type: "gap"
    }
]
};
myMenu.attachToId("myDiv");

```

Handle constructor

```
var myMenu = new FPComponents.Menu_A();
```

Creates a handle to a new menu.

Handle attributes

model

Type: object

This attribute represents the elements in the menu.

The object should contain an attribute named `content`, which should contain a list of elements, where each element is represented by an object.

```

myMenu.model =
{
    content: [
        { ... element 1 ... },
        { ... element 2 ... },
        { ... element 3 ... },
        { ... element 4 ... }
    ]
}

```

The following attributes can be used on each element object:

Attribute	Description
<code>type</code>	Mandatory attribute. Can be either "label", "gap", or "button".
<code>onclick</code>	Valid for type "button" only. A callback function that should be called when the button is clicked.
<code>label</code>	Valid for type "label" and "button" only. The text that should be displayed on the element.
<code>icon</code>	Valid for type "button" only. The path (relative to the web root, i.e. where the WebApp manifest is located) to an image to be displayed on the button. The optimal size for the icon image is 40x40 pixels, but it will be automatically resized.
<code>arrow</code>	Valid for type "button" only. If set to <code>true</code> , an arrow will be displayed on the right side of the button. Use this to indicate that the clicking the button will cause UI navigation, rather than potentially affecting the system directly. If set to <code>false</code> (default), no arrow will be displayed.
<code>flash</code>	Valid for type "button" only. If set to <code>true</code> , the button will flash. Use this to draw attention when the user needs to be aware of something, e.g. an alarm or a finished task. If set to <code>false</code> (default), the button will not flash.
<code>flashColor</code>	Valid for type "button" only. Set to any valid CSS color string to change which background color the button will flash to if <code>flash</code> is set to <code>true</code> as well. See section 5.3 for standard colors to use. Default color is set to <code>--fp-color-YELLOW-WARNING</code>

<code>enabled</code>	<p>Valid for type "button" only.</p> <p>If set to <code>true</code> (default), this attribute has no effect.</p> <p>If set to <code>false</code>, the button will be disabled, i.e. it will be "grayed-out" and cannot be clicked.</p>
----------------------	--

Setting the model attribute will automatically refresh the menu:

```
myMenu.model = { ... some model ... }
```

... will cause a refresh.

However, simply modifying the current model:

```
myMenu.model.content[0].label = "New button label";
```

... will *not* cause a refresh, since the component will not notice that the model has changed. To force a refresh, set the model to itself after modifying it:

```
myMenu.model.content[0].label = "New button label";
```

```
myMenu.model = myMenu.model;
```

See example above.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

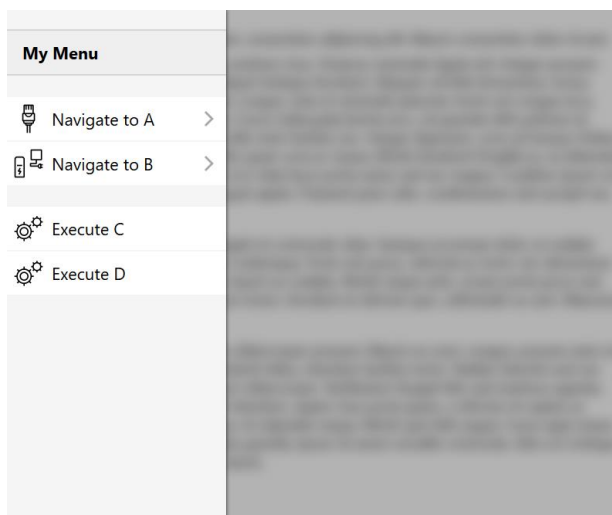
Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.18 Fold-in

Description

The Fold-in component implements an area which can be temporarily shown on the left side of the screen, on top of the rest of the UI.

It can be populated with any content. In the image below the Menu component example from section 5.16 is used.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-foldin-a.js"></script>
```

Example

```
var myFoldin = new FPComponents.Foldin_A();
myFoldin.contentId = "foldinId";
myFoldin.width = 350;
myFoldin.attachToBody();

...

document.getElementById("foldinId").textContent = "My content";
myFoldin.show();
```

In this example, the content DOM element created by the component, is fetched by calling the standard JavaScript `document.getElementById` function, using the string identifier provided as `contentId`.

To fill the Fold-in component with more complex content, many approaches exist, such as

- Creating content in JavaScript and appending that content using the standard JavaScript `append` method, available on the element.
- Moving a part of the existing DOM tree into the content element using the same `append` method. This sub-tree could be from plain HTML.
- Using the `attachToId` method available on components from the Components Library. This is especially useful for creating a Menu component inside a Fold-in.

Handle constructor

```
var myFoldin = new FPComponents.Foldin_A();
```

Creates a handle to a new Fold-in.

Handle attributes

width

Type: `string` or `number`

This attribute represents the width of the Fold-in. Examples:

```
myFoldin.width = "350px";    // 350 pixels
myFoldin.width = 350         // 350 pixels
myFoldin.width = "40vw";     // 40 % of screen width
```

contentId

Type: `string`

This attribute represents the element ID that will be used for the content div element in this Fold-in. After attaching the Fold-in to the document body, this element ID can be used for adding content to the Fold-in. See example and general explanation above.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToBody

Signature: `attachToBody ()`

Creates the visual component as DOM tree elements directly under the document body.

Further updates to the component handle will affect this visual component.

The Fold-In applies a blur filter to all sibling elements under the document body when shown. If performance is an issue, consider putting the body content in a wrapping `<div>`.

The blur filter will not affect text nodes directly under the body. To avoid this issue, put any such nodes in `<div>`, `<p>` or `` blocks.

show

Signature: `show ()`

Makes the Fold-in visible.

hide

Signature: `hide ()`

Makes the Fold-in hidden.

5.19 Popup Dialog

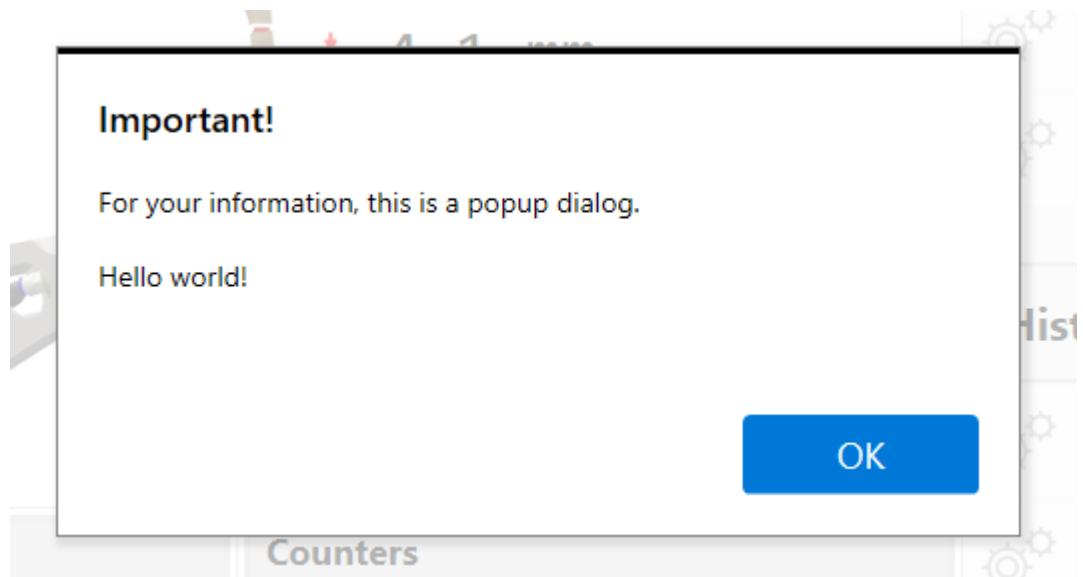
Description

A popup dialog is a modal component that can be used to in a simple manner provide the user with information, typically error/info messages or other occasional messages.

Another possibility is to have a simple "OK/Cancel" confirmation dialog.

It's easy to change the style of the dialog by setting the style to either warning or danger, which will add icons and change the color of the top border.

There's also support for additional customization options.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-button-a.js"></script>
<script src="fp-components/fp-components-popup-a.js"></script>
```

Example 1

```
FPComponents.Popup_A.message(
    "Important!",
    [
        "For your information, this is a popup dialog.",
        "",
        "Hello world!"
    ]
);
```

Example 2

```
FPComponents.Popup_A.message(
    "Warning!", "Something went wrong", function (action) {
        // callback for clicking OK button
    }, FPComponents.Popup_A.STYLE.WARNING);
```

Example 3

```
FPComponents.Popup_A.confirm(
    "Heads up!", "Should we continue?",
```

```
function (action) {  
    if (action == FPComponents.Popup_A.OK) {  
        console.log("Let's go!");  
    } else if (action == FPComponents.Popup_A.CANCEL) {  
        console.log("Abort mission!");  
    }  
};
```

Example 4

```
FPComponents.Popup_A.confirm(  
    "Heads up!", "Should we continue?",  
    function (action) {  
        switch (action) {  
            case FPComponents.Popup_A.OK:  
                console.log("Let's go!");  
                break;  
            case FPComponents.Popup_A.CANCEL:  
                console.log("Abort mission!");  
                break;  
        }  
    }  
);
```

Example 5

```
var model = {  
    header: [  
        { type: "text", text: "This is a header"}  
    ],  
    content: [  
        { type: "text", text: "This is a line in a message" },  
        { type: "text", text: "This will be on another line" },  
    ],  
    footer: [  
        {  
            type: "button",  
            text: "Custom button",  
            highlight: true,  
            closeDialog: false,  
            action: "myAction"  
        }  
    ]  
}
```

```
FPComponents.Popup_A.custom(model, function(action){  
    // implement callback for button clicks.  
});
```

Usage

Three functions are available directly on the Popup_A object, there is no need to instantiate a new object.

First, there is the message function. This variant produces a simple popup with just an 'OK' button.

```
FPComponents.Popup_A.message(string topic, string/string[]  
message, function callback=null, string style)
```

To produce a confirmation popup, with both 'OK' and 'Cancel' buttons, use the confirm function.

```
FPComponents.Popup_A.confirm(string topic, string/string[]  
message, function callback=null, string style)
```

Argument	Description
topic	A string, describing the topic of the message.
message	<p>The actual message. Can be either a simple string or, if several lines are required, an array where each element is a string with a message line.</p> <p>Example 1: "A message"</p> <p>Example 2: ["Message line 1", "Message line 2"]</p>
callback	<p>A function that will be called when the user dismisses by pressing the 'OK' or 'Cancel' button.</p> <p>This argument is optional, and the default value is null.</p> <p>The function signature should be</p> <pre>function (action)</pre> <p>where <i>action</i> is a string describing the action. The strings are predefined and can hence be compared using the == or === operator.</p> <p>The possible values for <i>action</i> are</p> <pre>FPComponents.Popup_A.OK</pre> <p>and</p> <pre>FPComponents.Popup_A.CANCEL</pre>

style	<p>A string determining the predefined style of the popup.</p> <p>This argument is optional, and the default value is null.</p> <p>The possible values for <i>style</i> are</p> <p><code>FPComponents.Popup_A.STYLE.INFORMATION</code></p> <p><code>FPComponents.Popup_A.STYLE.WARNING</code></p> <p><code>FPComponents.Popup_A.STYLE.DANGER</code></p> <p>Information style will add a blue information icon to popup.</p> <p>Warning style will add a warning icon to popup.</p> <p>Danger style will add a danger icon to popup</p>
-------	--

To produce a custom popup, without any default buttons, use the custom function. The custom function takes a model as parameter, which is detailed below.

```
FPComponents.Popup_A.custom(object model, function callback=null)
```

The model used by custom popup should always have three properties called header, content and footer, which will add elements to corresponding sections of the popup.

```
var model = {
  header: [...elements],
  content: [...elements],
  footer: [...elements],
}
```

See full example in examples section above.

Attribute	Description
type	Mandatory attribute. Can be either "text" or "button".
text	<p>Optional attribute.</p> <p>The text that should be displayed on the element.</p> <p>For multiline text, use several text element objects.</p>
action	<p>Valid for type "button" only.</p> <p>The action that should be passed to the callback function when the button is clicked. This action can be any value.</p>
highlight	Valid for type "button" only.

	Set to <code>true</code> to use highlight styling for button, same as button component. Default is <code>false</code>
<code>closeDialog</code>	Valid for type "button" only. Set this to <code>false</code> to prevent dialog from closing when the button is clicked. Default is <code>true</code> .

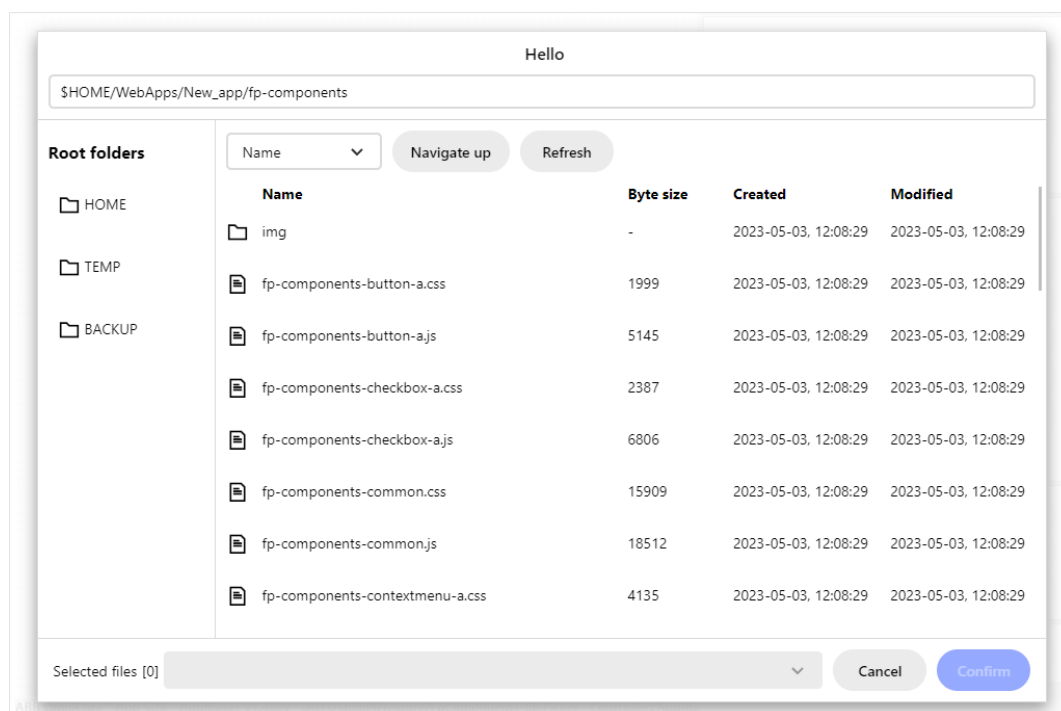
Each property has a list of elements in it, where each element is represented by an object. Each element object has the following possible attributes:

5.20 File System Dialog

Description

A file system dialog is a modal component that is used to provide an interactive method for selecting files or folders from the controller's file system.

The dialog can be customized with different options, such as to use a custom topic, use a file name filter, select single or multiple file system objects, and select which type of file system object to be target by the dialog (files or folders).



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-button-a.js"></script>
<script src="fp-components/fp-components-dropdown-a.js"></script>
<script src="fp-components/fp-components-filesystemdialog-
a.js"></script>
```

Please note that this component uses the Robot Web Services Client Library.

Example 1

```
let filePath = await FPComponents.Filesystemdialog_A.select();
```

Example 2

```
let filePath = await FPComponents.Filesystemdialog_A.select(
    FPComponents.Filesystemdialog_A.SelectionMode.Single,
    FPComponents.Filesystemdialog_A.FileSystemObjectType.File,
    "A custom dialog topic",
    [".txt", ".png", ".js", ".css"],
    FPComponents.Filesystemdialog_A.ROOT_FOLDER_HOME);
```

Example 3

```
let folderPathArr = await FPComponents.Filesystemdialog_A.select(  
    "multi",  
    "folder",  
    null,  
    "$BACKUP/myFolder");
```

Usage

Just like the Popup component, there is no need to instantiate a new object. The `select` function can be called directly from the component class object, without the need of a constructed instance.

```
FPComponents.Filesystemdialog_A.select(selectionMode="single",  
fileSystemObjectType="file", string textHeading=null, string[]  
validFileEndings=null, string defaultFolderPath="$HOME")
```

Argument	Description
<code>selectionMode</code>	Denotes if single or multiple file system objects can be selected from the dialog. This argument is optional, and the default value is "single". Valid inputs are "single" or "multi".
<code>fileSystemObjectType</code>	Denotes the file system object type, that is to be targeted by the selection in the dialog (file or folder). This argument is optional, and the default value is "file". Valid inputs are "file" or "folder".
<code>textHeading</code>	The optional heading/topic text of the dialog. This argument is optional, and the default value is null.
<code>validFileEndings</code>	An array of strings, denoting valid file name endings. This argument is optional, and the default value is null. Only file names on the VC/RC that ends with at least one of the strings in this array will be displayed by the dialog. This filter is ignored when set to null (dialog displays all file names).
<code>defaultFolderPath</code>	The default folder path of the dialog.

	<p>This argument is optional, and the default value is null.</p> <p>If path is invalid, this path will automatically be set to \$HOME.</p> <p>The dialog considers the following folder paths as valid: \$HOME, \$TEMP, \$BACKUP, or any of their sub folders.</p>
--	--

The `select` method returns a Promise. The Promise result depends on the following scenarios:

Use cases	Promise result
User cancels the dialog	null
User confirms selection, with one or more file system objects selected.	An array of strings, where each string represents a chosen file or folder.

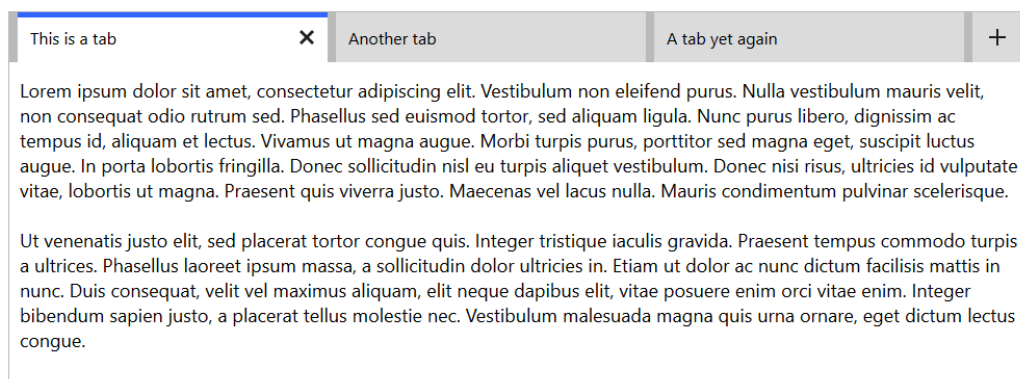
5.21 Tab Container

Description

The Tab container component implements a view-switching area with a visual tab bar at the top.

It can be populated with any content and can be configured for most use cases. Any number of tabs are supported.

Optionally, the component can be used as a generic view-switcher without any visible tab bar.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
<script src="fp-components/fp-components-tabcontainer-a.js"></script>
```

Example 1

The following example produces a simple full-window tab container with two tabs.

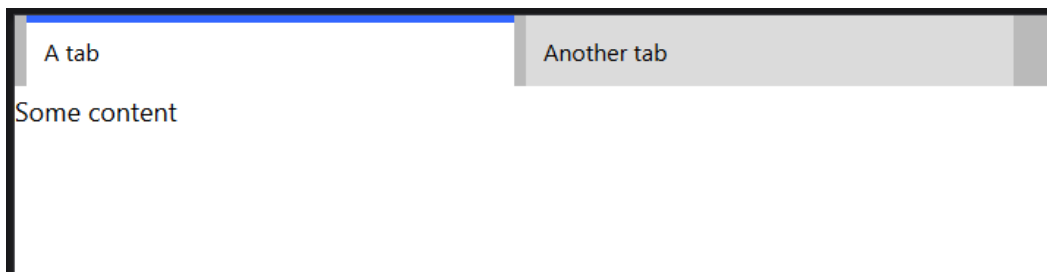
Note that it does not matter where the content div elements are placed, they will be moved into the tab container when the `addTab` method is used.

```
var myTabContainer = new FPComponents.Tabcontainer_A();
myTabContainer.addTab("A tab", "contentDiv1");
myTabContainer.addTab("Another tab", "contentDiv2");
myTabContainer.attachToId("myDiv");

...

<body>
  <div id="myDiv" style="height: 100vh"></div>

  <div id="contentDiv1">Some content</div>
  <div id="contentDiv2">Some more content</div>
</body>
```



Handle constructor

```
var myFoldin = new FPComponents.Tabcontainer_A();
```

Creates a handle to a new Tab container

Tab identifier objects

Many of the attributes and methods available on the Tab container component handle uses *tab identifier objects* as a way of referring to the tabs themselves.

When creating a tab using the `addTab` method, a tab identifier object is returned.

IMPORTANT: When comparing tab identifier objects, make sure to use the JavaScript *identity operator*, i.e. `===`, **not** the *equality operator*, i.e. `==`.

Handle attributes

`activeTab`

Type: `Object`

This attribute represents the currently active tab in the form of a tab identifier object.

Set this attribute to another tab identifier object to switch tab/view programmatically.

`onchange`

Type: `function (Object oldTabId, Object newTabId)`

Set this attribute to a callback function that should be called when a new tab becomes active, i.e. the container has switched view to another tab.

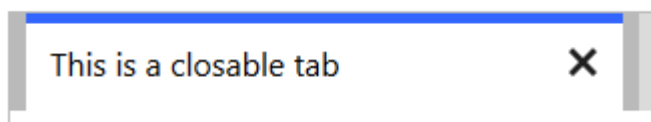
The `oldTabId` and `newTabId` parameters are both tab identifier objects which respectively identify the previously active tab and the currently active tab. It is possible for one of the parameters to be null, e.g. when a first tab is created, or when the last tab is removed.

`userTabClosing`

Type: `boolean`

Set this attribute to `true` to allow the user to close tabs using a button on each active tab.

Default value is `false`.



`onuserclose`

Type: `boolean function (Object tabId)`

Set this attribute to a callback function that should be called when the user clicks on a tab's "close" button. The parameter `tabId` is the tab identifier object for the tab being closed.

If this attribute is not set, the tab will be closed immediately.

If it is set to a function, that function will be called, and if the function returns `true`, the tab will be removed directly.

If the function returns `false`, the tab will not be removed, and it is up to the developer to later close the tab using the `removeTab` method. This is useful for implementing user interaction when closing the tab, e.g. "Do you want to save?" or "Do you really want to close?".

onplus

Type: `function ()`

Set this attribute to a callback function that should be called when the user clicks on the plus ("add-tab") button. Clicking the button has no effect other than this callback function being called.

Typically, this callback function will dynamically build a new view and then add it as a new tab using the `addTab` method.

Setting this attribute to a function will cause the plus button to appear in the tab bar.

**plusEnabled**

Type: `boolean`

Set this attribute to `false` to disable and gray out the plus ("add-tab") button, whenever visible.



Default value is `true`.

hiddenTabs

Type: `boolean`

Set this attribute to `true` to completely hide the tab bar. This means that only the tab content will be visible.

This is useful for using the Tab container component as a generic view-switcher.

Default value is `false`.

tabIdList

Type: `array`

This attribute represents a list of objects identifying the current tabs in the Tab container.

The order of the list is the same as the order of the tabs. The list is generated each time it is read from the handle object, which means that a read list will not change unless read again.

Example, closing the third tab:

```
myTabContainer.removeTab(myTabContainer.tabIdList[2]);
```

This attribute is read-only.

parent

Type: Element

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

addTab

Signature: `addTab (string title, string/Element content, boolean makeActive=false)`

Creates a new tab and adds it to the Tab container.

The `title` string parameter specifies the tab label.

The `content` parameter specifies the content element of the tab.

The parameter can be either the id of an element existing in the DOM tree, or an Element object directly.

If the element is a child of a parent element, it will be removed from the parent before being placed within the Tab container. This means that if tab content is declared as plain HTML (with an *id* attribute), it can be placed anywhere within the document before being added to the Tab container, since it will be moved automatically.

The optional `makeActive` parameter can be set to `true` to automatically make the tab active after adding it.

The `addTab` method returns a tab identifier object for the new tab. This can be used for referring to the tab at a later stage.

removeTab

Signature: `removeTab (Object tabId)`

Removes the specified tab from the Tab container.

The `tabId` parameter should be a tab identifier object referring to the tab.

getTabTitle

Signature: `getTabTitle (Object tabId)`

Returns the title of the specified tab.

The `tabId` parameter should be a tab identifier object referring to the tab.

setTabTitle

Signature: `setTabTitle (Object tabId, string title)`

Updates the title of the specified existing tab.

The `tabId` parameter should be a tab identifier object referring to the tab.

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

5.22 Hamburger Menu

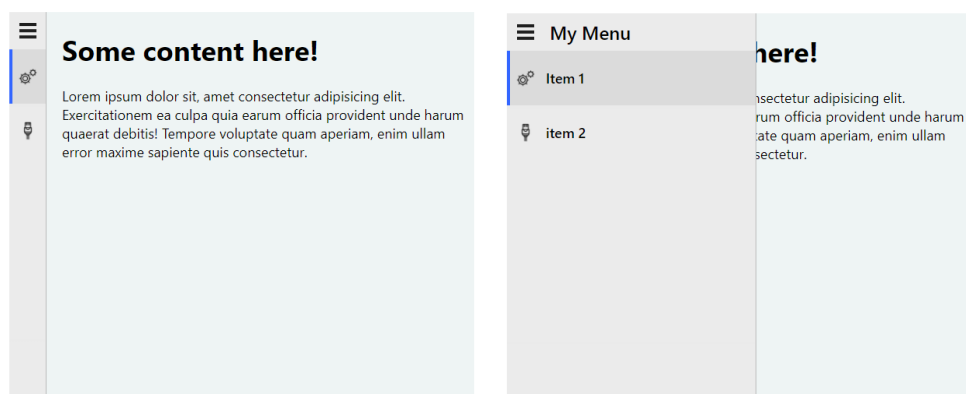
Description

The Hamburger Menu component implements a view-switching area with a hamburger menu. The hamburger menu can be configured to either always be visible or to only display a hamburger icon on the top left of the screen.

When configured to always be visible, the visible part will be only wide enough to display the icons of the menu and will push aside the content.

When configured to not be visible, the hamburger icon will be placed on top of the content, and on the top left. In this case it is up to the app developer to keep that area free of other elements.

The hamburger menu can be populated with any content and can be configured for most use cases. Any number of views are supported.



Requires

```
<script src="fp-components/fp-components-common.js"></script>
```

```
<script src="fp-components/fp-components-hamburgermenu-a.js"></script>
```

Example

This example creates a full screen hamburger menu and has the second item initially active. Note that the location of the content elements does not matter, they will be pulled into the hamburger menus content area when the model is set.

```
var myHamburgermenu = new FPComponents.Hamburgermenu_A();
myHamburgermenu.addView("Item 1", "contentDiv1", "option1.png", false);
myHamburgermenu.addView("Item 2", "contentDiv2", "option2.png", true);
myHamburgermenu.attachToId("myMenuContainer");
...

<body>
  <div id="myMenuContainer" style="height: 100vh; width: 100vw"></div>

  <div id="contentDiv1">Some content</div>
  <div id="contentDiv2">Some more content</div>
</body>
```

Handle constructor

```
var myHamburgermenu = new FPComponents.Hamburgermenu_A();
```

Creates a handle to a new hamburger menu component

Handle attributes

alwaysVisible

Type: boolean

Set this attribute to `true` (default) to have the hamburger menu always be visible. In this state the hamburger menu will be wide enough for the icons of each menu item to be visible when closed. When the menu is opened, it will be full width.

Set this attribute to `false` to hide the menu when closed, and only display the hamburger menu icon.

title

Type: string

Set this attribute to any string to display a title next to the hamburger menu icon when the menu is open.

Set this attribute to `false` (default) to hide the menu when closed, and only display the hamburger menu icon.

onchange

Type: `function (Object oldViewId, Object newViewId)`

Set this attribute to a callback function that should be called when a new view becomes active, i.e. the container has switched view to another content element.

The *oldViewId* and *newViewId* parameters are both view identifier objects which respectively identify the previously active view and the currently active view. It is possible for one of the parameters to be null, e.g. when the first view becomes active.

activeView

Type: `Object`

This attribute represents the currently active view in the form of a view identifier object.

Set this attribute to another view identifier object to switch view programmatically.

viewIdList

Type: `array`

This attribute represents a list of objects identifying the current views in the Hamburger menu.

The order of the list is the same as the order of the menu items. The list is generated each time it is read from the handle object, which means that a read list will not change unless read again.

Example, setting the third menu item and it's view as active:

```
myHamburgerMenu.activeView = myHamburgerMenu.viewIdList[2];
```

This attribute is read-only.

parent

Type: `Element`

After attaching the component to an element, this attribute will contain the parent element that the component has been attached to.

This attribute is read-only.

Methods

attachToId

Signature: `attachToId (string elementId)`

Creates the visual component as DOM tree elements under a parent element, specifying the parent element using an element ID (*elementId*).

Further updates to the component handle will affect this visual component.

IMPORTANT: Always use an empty div element. Do not attach the component to an element with existing children.

attachToElement

Signature: `attachToElement (Element element)`

Same as `attachToId`, but takes an `Element` object directly as argument instead of an element ID.

getViewButtonLabel

Signature: `getViewButtonLabel (Object id)`

Returns the label of the specified views menu item.

The `id` parameter should be a view identifier object referring to the view.

setViewButtonLabel

Signature: `getViewButtonLabel (Object id, string label)`

Sets the label of the specified views menu item to the provided label.

The `id` parameter should be a view identifier object referring to the view

addView

Signature: `addView (string label, string/Element contentElement, string icon, boolean active=false)`

Creates a new view with corresponding menu item and adds it to the Hamburger menu.

The `label` string parameter specifies the menu item label.

The `contentElement` parameter specifies the content element of the view to be displayed.

The parameter can be either the id of an element existing in the DOM tree, or an `Element` object directly.

If the element is a child of a parent element, it will be removed from the parent before being placed within the Hamburger menu. This means that if tab content is declared as plain HTML (with an *id* attribute), it can be placed anywhere within the document before being added to the Hamburger menu, since it will be moved automatically.

The `icon` parameter specifies a path to an icon to be used in the menu item. The optimal size for the icon is 24x24 px but it will be automatically resized.

The optional `active` parameter can be set to `true` to automatically make the view active after adding it.

The `addView` method returns a view identifier object for the new view. This can be used for referring to the view at a later stage

removeView

Signature: `removeView (Object id)`

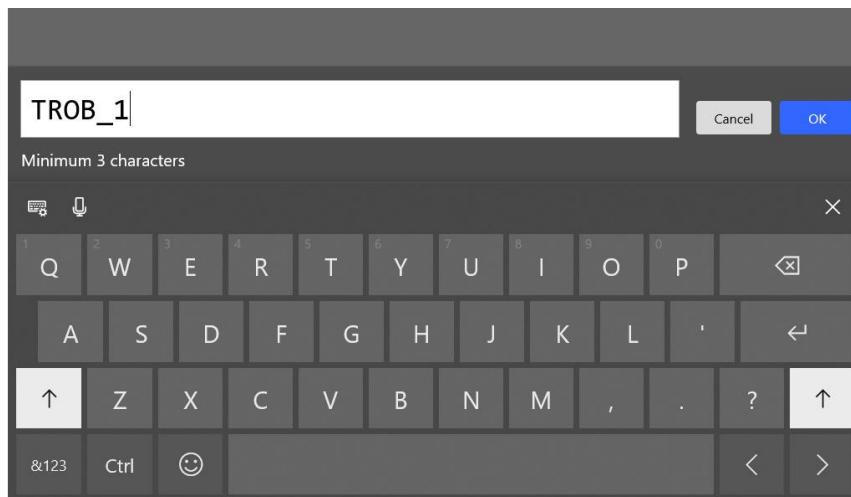
Removes the referenced view from hamburger menu component, meaning the menu item is removed and if the view was active, the content is removed as well.

The `id` parameter should be a view identifier object referring to the view.

5.23 On-screen Keyboard

Description

The on-screen keyboard can be used when text input from the user is required, but the Input field component (which has integrated support for this keyboard) cannot be used.

**Requires**

```
<script src="fp-components/fp-components-common.js"></script>
```

Example

```
fpComponentsKeyboardShow( function(text) {  
    // This function will execute when  
    // the user has finished editing.  
}, "Change me" );
```

Usage

```
fpComponentsKeyboardShow ( function callback, string initial=null,  
string label=null, object variant=null, RegExp regex=null,  
function validator=null )
```

Argument	Description
<code>callback</code>	<p>A callback function that will execute when the user has finished the input.</p> <p>The function signature should be:</p> <pre>function (text)</pre> <p>If the user finishes input and clicks on the “OK” button, <code>text</code> will be a string containing the new value.</p> <p>If the user cancels input (by clicking on the close button or by clicking on the shaded background), <code>text</code> will be <code>null</code>.</p>
<code>initial</code>	<p>Initial string that should be edited by the user. Can be omitted, default value is <code>null</code>.</p>
<code>label</code>	<p>Descriptive label string that will be visible below the editor field on the keyboard. Should describe the value that the user is editing, preferably including any input limitations.</p> <p>Can be omitted, default value is <code>null</code>.</p>
<code>variant</code>	<p>Initial keyboard variant to be shown. Possible values are <code>FP_COMPONENTS_KEYBOARD_ALPHA</code> or <code>FP_COMPONENTS_KEYBOARD_NUM</code>.</p> <p>Can be omitted, default value is <code>null</code>.</p> <p>NOTE! <i>This argument is currently ignored and will not affect the appearance of the keyboard. This is due to limitations in built-in touch keyboard. It is possible that it will work in a future release.</i></p>
<code>regex</code>	<p>Standard JavaScript regular expression object used for validating and allowing the input.</p> <p>Example:</p> <pre>/^-?[0-9]+(\.[0-9]+)?\$/</pre> <p>will only allow input of floating-point numbers or integers.</p> <p>Can be omitted, default value is <code>null</code>.</p> <p>Can be used in combination with the <code>validator</code> argument.</p>

<code>validator</code>	<p>A callback function that will execute whenever the user is altering one or more characters in the editable field.</p> <p>The function signature should be:</p> <pre>function (val)</pre> <p>The <code>val</code> argument will be the current value (string) as it is being edited by the user. The callback function should return <code>true</code> when the value is acceptable and <code>false</code> when not acceptable.</p> <p>Can be omitted, default value is <code>null</code>.</p> <p>Can be used in combination with the <code>regex</code> argument.</p>
------------------------	--

5.24 Console Log Overlay

Description

When developing for the FlexPendant, it can be useful to access the standard JavaScript console log. Normally, this is not possible, since – unlike a common web browser – the FlexPendant has no integrated native console.

The Console log overlay component can be used to view the log (`console.log`, `console.info`, `console.error`). Since the overlay jacks into the default logger in JavaScript, it is likely that any third-party code will have its output printed here.

The overlay should only be used during development.

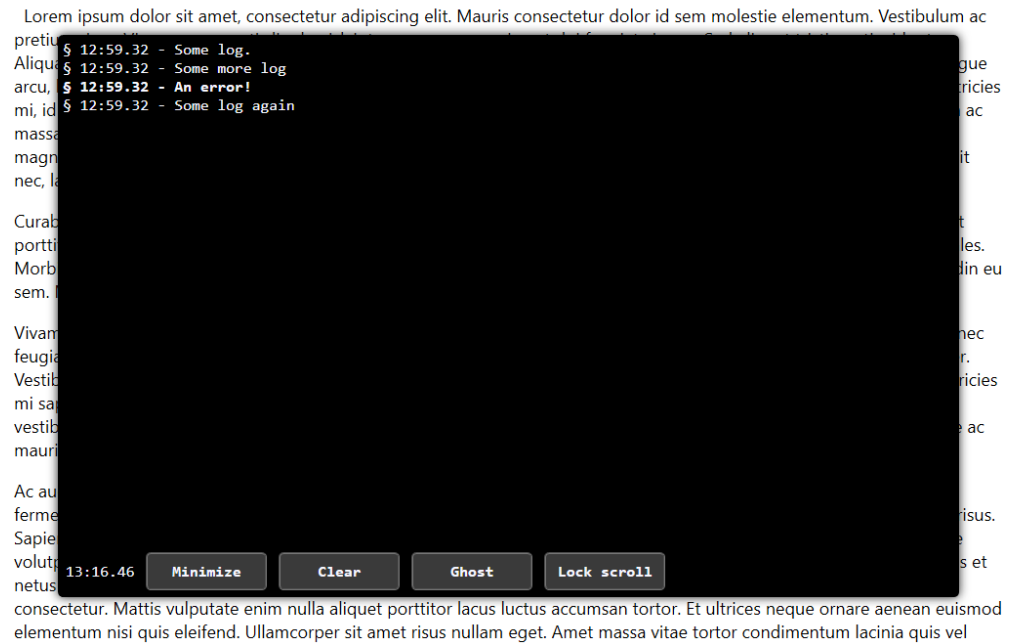
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris consectetur dolor id sem molestie elementum. Vestibulum ac pretium risus. Vivamus venenatis ligula nisl. Integer posuere sapien et dui feugiat viverra. Sed aliquet tristique tincidunt. Aliquam vel felis fermentum, luctus augue nec, ullamcorper justo. Donec congue, ante et venenatis placerat, lorem est congue arcu, lobortis congue neque lacus id odio. Fusce malesuada lacinia arcu, vel gravida nibh pulvinar id. Vestibulum rutrum ultricies mi, id mollis enim facilisis nec. Integer dignissim, urna vel tempor finibus, metus enim ultricies odio, sed sagittis quam urna ac massa. Morbi hendrerit fringilla ex, eu bibendum neque facilisis ac. Suspendisse eget orci vitae lacus porta varius sed nec magna. Curabitur ipsum orci, lobortis ac erat vitae, finibus consequat sapien. Praesent justo odio, condimentum sed suscipit nec, lacinia non nisi.

Curabitur faucibus eros odio, ac feugiat mi commodo vitae. Quisque accumsan dolor ut sodales tempus. Praesent tincidunt porttitor scelerisque. Proin nisi purus, vehicula ac tortor vel, elementum egestas purus. Fusce scelerisque in mauris eu sodales. Morbi neque ante, ornare porta purus sed, vehicula iaculis lectus. Vivamus ipsum lorem, tincidunt at ultricies quis, sollicitudin eu sem. Maecenas molestie erat a fringilla congue.

Vivamus fermentum metus eu dolor ullamcorper posuere. Mauris eu nunc congue, posuere ante non, fermentum eros. Donec feugiat hendrerit tellus, interdum facilisis tortor. Nullam lobortis erat nec urna pellentesque, at hendrerit ipsum ullamcorper. Vestibulum feugiat felis sed maximus egestas. Nullam maximus, quam nec blandit interdum, sapien risus porta quam, a ultricies mi sapien ac neque. Quisque ut malesuada augue, id vulputate neque. Morbi quis felis augue. Fusce eget massa vel nibh vestibulum ultrices. Phasellus gravida, ipsum sit amet convallis commodo, felis orci tristique justo, non imperdiet justo ante ac mauris.

Ac auctor augue mauris augue. Luctus venenatis lectus magna fringilla urna porttitor rhoncus dolor. Felis imperdiet proin fermentum leo vel orci porta non pulvinar. Euismod elementum nisi quis eleifend. Nisl purus in mollis nunc sed id semper risus. Sapien eget mi proin sed libero enim sed faucibus turpis. Eleifend donec pretium vulputate sapien nec sagittis. Enim neque volutpat ac tincidunt vitae semper quis lectus. Nunc consequat interdum varius sit amet mattis vulputate. Tristique senectus et netus et malesuada. Turpis egestas maecenas pharetra convallis posuere morbi. Neque laoreet suspendisse interdum consectetur. Mattis vulputate enim nulla aliquet porttitor lacus luctus accumsan tortor. Et ultrices neque ornare aenean euismod elementum nisi quis eleifend. Ullamcorper sit amet risus nullam eget. Amet massa vitae tortor condimentum lacinia quis vel

In the example above, note the gray square in the lower left corner. It will be visible when the overlay is enabled. Clicking on it will bring up the overlay:



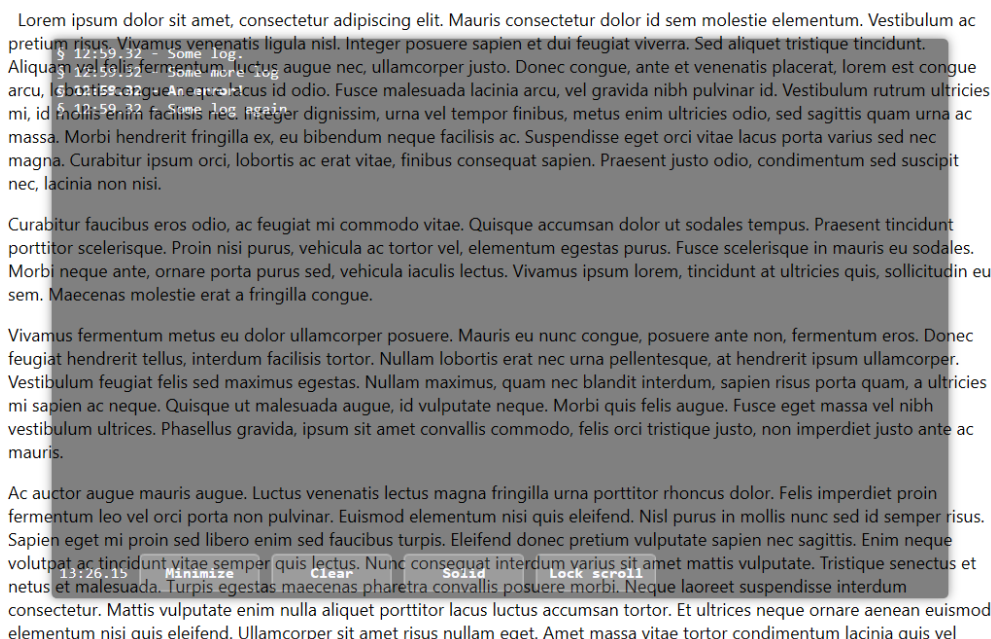
The overlay features a scrollable log, a reference clock, and a few buttons:

The **Minimize** button will hide the overlay and once again display the launcher square in the lower left corner.

The **Clear** button will clear the log.

The **Lock scroll** button will stop automatic scrolling until clicked again.

The **Ghost** button will cause the overlay to be semi-transparent. When this mode is active, it is possible to “click through” the overlay. This is useful for interacting with the UI and monitor the log at the same time. See image below.



```
<script src="fp-components/fp-components-common.js"></script>
```

```
// Run this early in the application when debugging.
fpComponentsEnableLog();
```

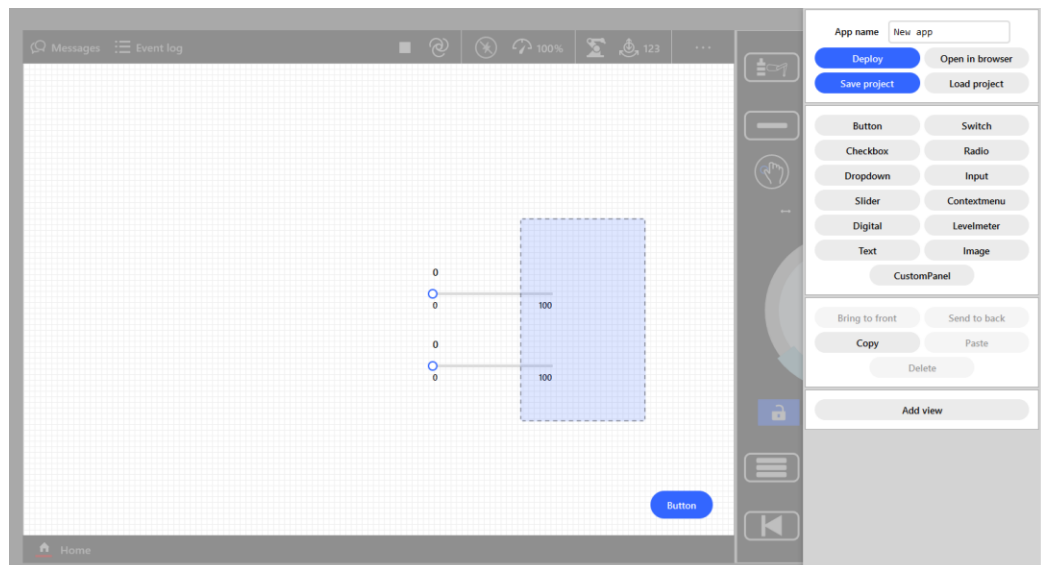
fpComponentsEnableLog ()

6 AppMaker

6.1 Introduction

What is AppMaker?

AppMaker is an optional visual editing tool for designing and generating OmniCore App SDK-based user interfaces.



AppMaker can be used for...

- Creating simple user interfaces, complete with bindings to various resources on the controller, such as RAPID or I/O
- Creating simple user interfaces, extended with manually written code
- Quickly setting up an application skeleton, which can be used as a starting point for a manually coded OmniCore App SDK application

Features

AppMaker provides a useful set of features to speed up development of simple UI applications for the OmniCore FlexPendant:

- An easy-to-use, drag-and-drop, snap-to-grid layout system without the need to learn complicated CSS layouts
- Direct support for most components from the Components Library
- Flexible multi-view support - any number of tabs can be added to the application
- Various automatic bi-directional value bindings, i.e., RAPID and I/O – configuration is simply a matter of selecting from lists with available resources
- Binding execution of RAPID procedures to buttons

- Support for automatically disabling ("greying out") components when selected conditions are met, such as the robot controller being in a specific state, a RAPID boolean is *true*, or an I/O signal is set
- Quick deployment to the controller directly from the visual editor
- Generates human-readable self-contained apps with no dependencies to AppMaker itself - which means that AppMaker is not needed to run the applications
- No external toolchain is required
- Good support for extending the generated app with manually written code, clearly separated from generated code – useful for adding behavior that is not directly supported by AppMaker itself

6.2 Installation

Installation

AppMaker is installed as a RobotWare Add-In on a robot controller which will be used for development.

The robot controller is used as a web server for AppMaker, which itself is a web-based application. The controller is also used as the target for deploying the generated app, and as a reference for picking various resources such as I/O signals and RAPID variables and procedures.

A virtual robot controller (VC) is perfectly fine for development purposes.

The Add-In can be found in the OmniCore App SDK bundle, downloaded from RobotStudio Developer Center (<https://developercenter.robotstudio.com>), and is in the *rspak* format. It can be added in the Add-In tab in RobotStudio by using the *Install* button.

After adding the *rspak* file to RobotStudio, when installing the robot system, simply add the product *OmniCore App SDK AppMaker* after adding a RobotWare distribution. No option needs to be selected for AppMaker, adding the product itself is sufficient.

Please note that the AppMaker add-in is **not** required to be installed on a system to run an application that has been created using AppMaker. Each generated app is self-contained.

AppMaker comes with a bundled version of the corresponding version of the App SDK. The library files will automatically be included in each generated app. There is no need to install/copy the App SDK libraries manually.

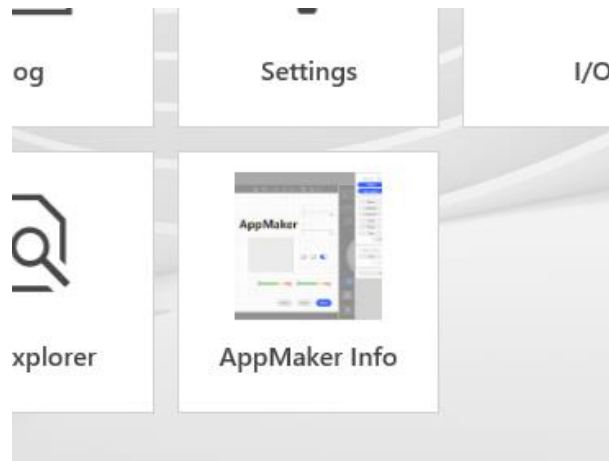
6.3 Accessing AppMaker

Accessing AppMaker

AppMaker is accessed using a desktop browser, e.g., Chrome, Firefox, or Edge. The controller must be running while AppMaker is used.

the URL for AppMaker depends on several factors, including version of AppMaker, whether the controller is virtual or not, and – when virtual – number of controllers running on the system.

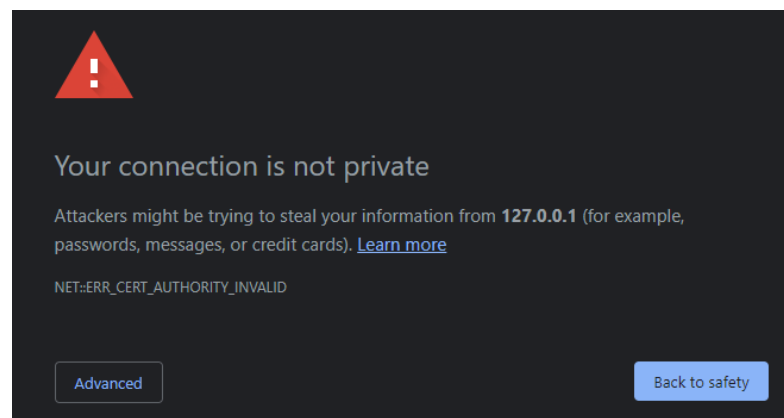
After starting up the system that has AppMaker installed, the URL can be found in a FlexPendant app called “*AppMaker Info*”, available on the FlexPendant’s home menu. When running on a virtual system, the URL can be marked, copied, and pasted into the browser.



Unless a valid SSL certificate has been installed on the system (unlikely if virtual), a warning about the certificate will likely appear in the browser. The meaning of the warning is that since there is no valid certificate, the browser cannot verify the identity of the server, in this case the controller.

Information on how to install a valid certificate on a physical controller is beyond the scope of this manual. Valid certificates are recommended for non-virtual controllers.

Example from the Chrome browser:



Since the certificate is self-signed, an exception is needed to proceed.

In Google Chrome, click on “Advanced”, then “Proceed to 127.0.0.1 (unsafe)”.

In Microsoft Edge and Mozilla Firefox, the warnings are very similar.

6.4 Workflow

Creating a new app

1. Enter a name for the app in the **App name** input field.

2. Click on the **Deploy** button. This will cause an empty app structure to be created under *\$HOME/WebApps* on the controller, including files for user code.
3. Start/restart the FlexPendant to see the new app in the home menu.

Editing the app

1. Use the visual editor to create and configure UI components on the workspace. See more details below.
2. Generate the app and deploy it by clicking on the **Deploy** button. Generated files in *\$HOME/WebApps/<appname>* will be overwritten. These generated files are ***index_generated.html*** and ***code_generated.js***. Do not edit them manually, since they will be overwritten during each deployment.

In addition, all App SDK files will be synchronized to the corresponding bundled version in AppMaker.

Files meant for user code - such as *code.js* and *styles.css* - will be created if they do not exist, but they will not be overwritten if already existing.

Other files will be ignored.

3. Open the app on the FlexPendant and view the results. If the app is already open, first close it by right-clicking (on virtual) or long-pressing on the apps button in the activity bar on the bottom on the FlexPendant screen and select "Close". There is no need to restart the FlexPendant to simply refresh the app.
4. To view the deployed app directly in the browser, click in the **Open in browser** button in AppMaker.

Saving the AppMaker project

To be able to keep working on the app at a later stage, make sure to save the AppMaker project by clicking on the **Save project** button. An *appm* file will be downloaded to the default download folder of the browser.

Make sure to save this file in a safe place, preferably a version control system.

IMPORTANT: The project file only contains layout information and properties for the graphical controls, and other information strictly tied to AppMaker itself. In other words, it is the source file for the generated files *index_generated.html* and *code_generated.js*.

Any other resources added by the user, such as custom manually written code, icons, and images need to be backed up from *\$HOME/WebApps/<appname>* by the developer and stored in a safe place.

AppMaker is a tool for generating code, not storing the results.

Project file autosave

A backup copy of the project file will be copied directly under *\$HOME* whenever the project file is downloaded, or the app is deployed. Consider this a backup in case of accidental loss of the downloaded project file, or if the browser window/tab is closed before downloading the project file. The autosaved project file is not meant to be a replacement for downloading the project file.

The autosaved project file will be named `<appname>.autosave.appm`, and will be overwritten each time an autosave occurs.

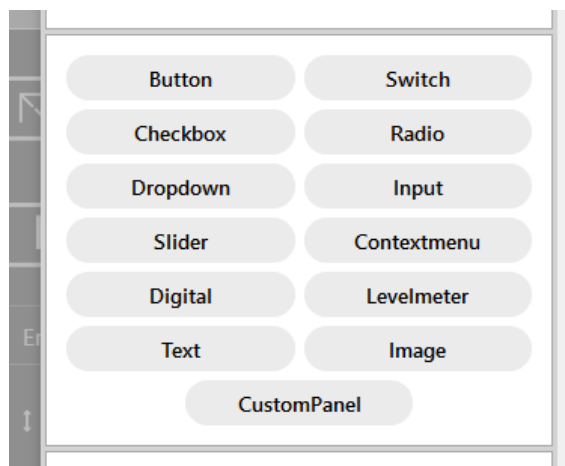
Loading an AppMaker Project

To keep working on the project after closing down AppMaker, start AppMaker again and click on the **Load project** button. Select the previously saved project file.

6.5 Using The Visual Editor

Creating UI components

Create and add new UI components to the workspace by clicking on the corresponding button in the components palette:

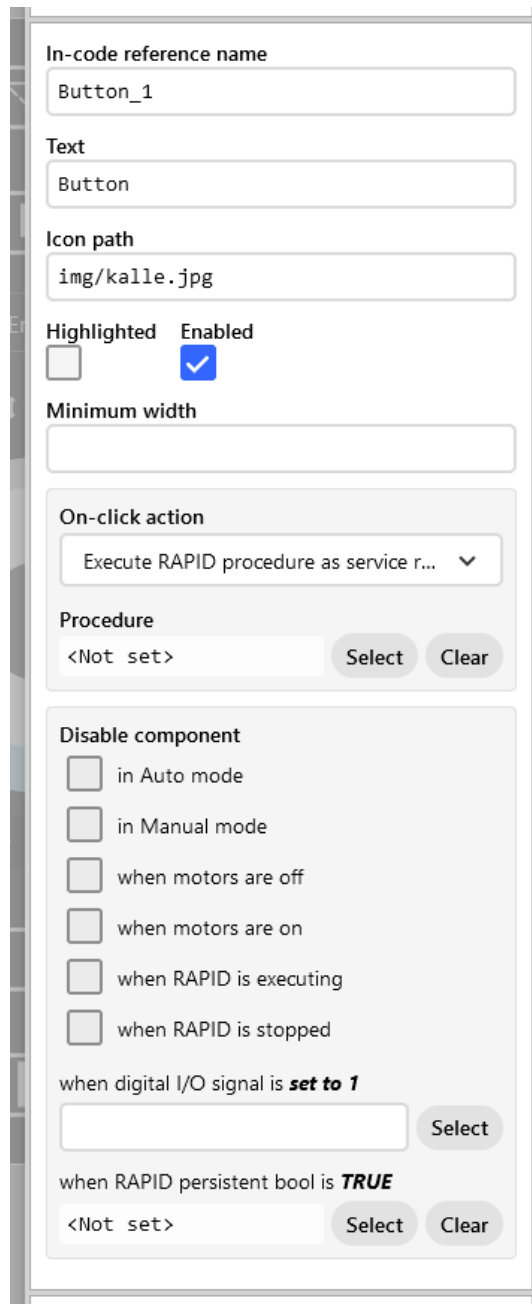


Components appear in the workspace, and can be selected and dragged around.

When moved, the components "snap" to a grid of 8x8 pixels. This makes it easier to place the components in good alignment with each other.

Configuring UI components

While a single component is selected, its configuration settings is shown. E.g., when a Button is selected, the following settings can be seen:



The screenshot shows a configuration panel for a component. It includes the following fields and options:

- In-code reference name:** A text field containing "Button_1".
- Text:** A text field containing "Button".
- Icon path:** A text field containing "img/kalle.jpg".
- Highlighted:** A checkbox that is unchecked.
- Enabled:** A checkbox that is checked, indicated by a blue checkmark.
- Minimum width:** An empty text field.
- On-click action:** A dropdown menu showing "Execute RAPID procedure as service r...".
- Procedure:** A text field containing "<Not set>" with "Select" and "Clear" buttons.
- Disable component:** A section with six checkboxes:
 - ☐ in Auto mode
 - ☐ in Manual mode
 - ☐ when motors are off
 - ☐ when motors are on
 - ☐ when RAPID is executing
 - ☐ when RAPID is stopped
- when digital I/O signal is *set to 1*:** A text field with a "Select" button.
- when RAPID persistent bool is *TRUE*:** A text field containing "<Not set>" with "Select" and "Clear" buttons.

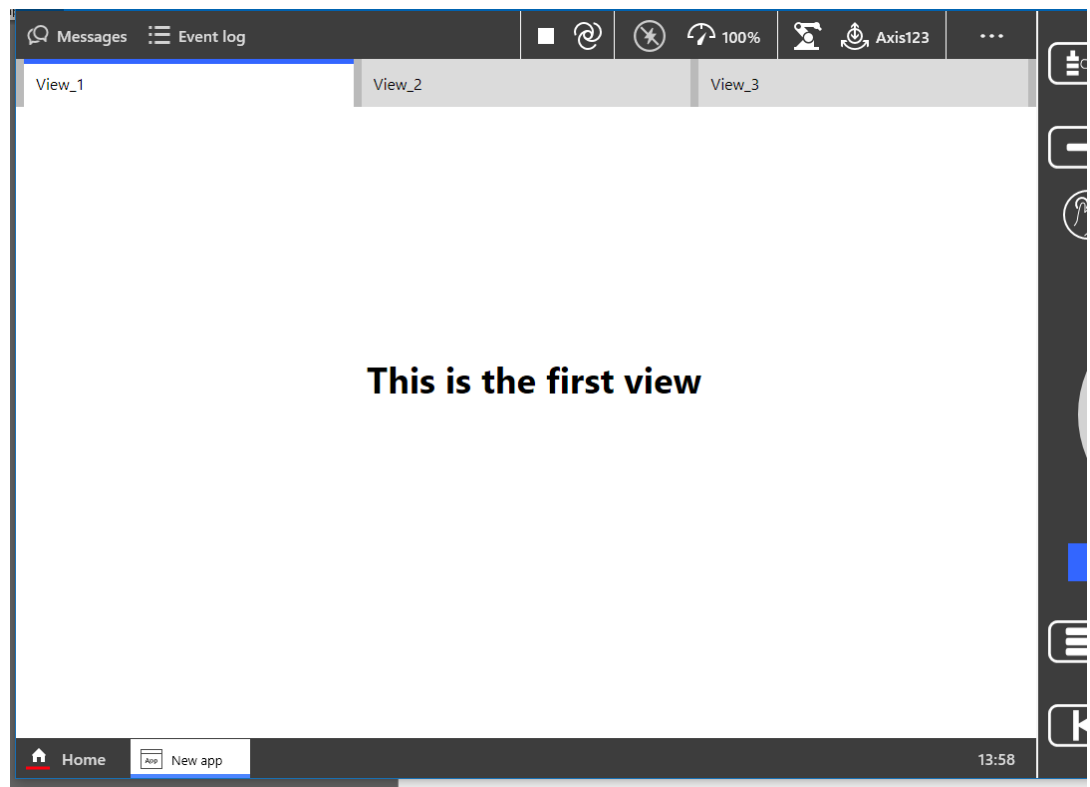
Each type of component has different settings, but they come in three categories:

- The **In-code reference name**, an identifier which will be used to refer to the component when writing custom code. The default value can be left as-is, but it can be a good idea to give the component a more descriptive name. This attribute value must be unique within the app.
- Settings regarding the behavior of the component itself. This is analogous to the corresponding attributes of the graphical components in the App SDK's component library. In the Button example above, these settings are **Text**, **Icon path**, **Highlighted**, **Enabled** and **Minimum width**

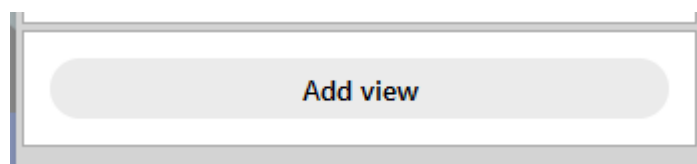
- Settings regarding automatic binding of values, actions or state to controller resources or custom code.
In the Button example above, these settings are **On-click action** (and the related **Procedure**) and **Disable component**.

Multiple views

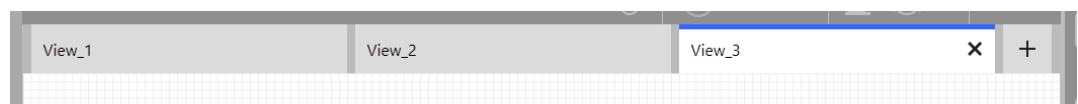
AppMaker supports apps with multiple views, in the form of *tabs*.



To add a view to the project, click on the **Add view** button.



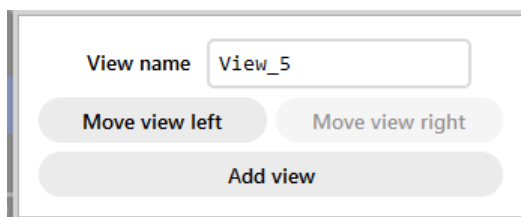
When the project has more than one view, the tabs will be visible at the top of the workspace.



Each view (except the last one remaining) can be removed by clicking on the **X** on the right side of the tab.

New tabs can be created by either clicking on the **+** to the far right, or by clicking the **Add view** button again.

Tabs be reordered by using the **Move view left** and **Move view right** buttons. The name of the currently selected view can be changed in the **View name** field.



UI components cannot be directly moved between views. However, it is possible to use the **Copy** and **Paste** buttons to copy UI components from one view to another.

6.6 Advanced - Adding Code

Motivation

Although AppMaker offers functionality for making basic applications, more advanced applications might require custom logic not directly available from AppMaker. This could be communication with the controller, custom actions, filters, custom components or anything else.

AppMaker comes with great flexibility for making such additions, with direct access to the App SDK APIs.

Setting up a development environment

While AppMaker is great for quickly making basic apps, it is not a complete development environment for writing JavaScript code.

Any editor of choice can be used for this purpose. We recommend Microsoft's **Visual Studio Code**, an open source editor/IDE for a large number of languages. It can be downloaded for free from <https://code.visualstudio.com>. Please observe that this is not an ABB site or product.

To start editing the app, start up Visual Studio Code, and from the **File** menu, select **Open folder** and select the folder of the generated app under the virtual controller's `$HOME` folder, i.e. `$HOME/WebApps/<appname>`.

To find out where the virtual system's `$HOME` directory is located on the computer, right-click on **HOME** in RobotStudio and select **Open folder**. If the folder cannot be found, make sure that the app has been deployed from AppMaker at least once.

User files

The following files meant for custom code will be created when deploying the app the first time, but will not be overwritten by AppMaker in consecutive deployments:

- `code.js`, for JavaScript code
- `styles.css` for custom CSS styles

For custom HTML, see "*The Custom Panel component*" below.

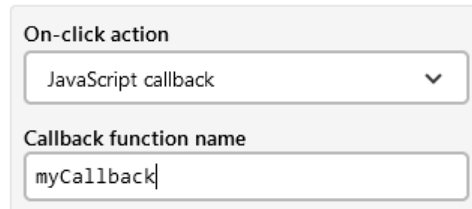
The `code.js` file contains several stubbed empty functions that can be extended with code. The file also contains plenty of comments to help.

All App SDK libraries are included and accessible.

Several components (e.g. Button) has the possibility to configure a JavaScript callback. Such callbacks should be created as functions in global scope, named accordingly to the configured function name in the component settings.

Example – a button starting RAPID execution:

A Button component has a callback with the name **myCallback** configured:



On-click action

JavaScript callback

Callback function name

myCallback

The callback is defined in the *code.js* file, starting RAPID execution using the *RWS.Rapid* API from App SDK when the button is clicked:

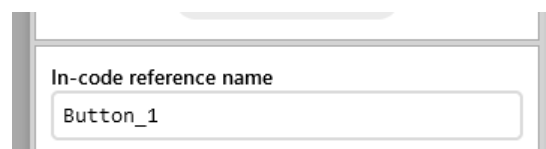
```
async function myCallback() {
  try {
    await RWS.Rapid.startExecution();
  } catch (e) {
    // Error handling here
  }
}
```

Accessing the UI components from custom code

The generated components can be accessed from the *code.js* file. References are stored under a global object named *Controls*. The **In-code reference name** of each UI component defines the name of a property of the *Controls* object with the reference as a value.

The main container, a *TabContainer* component, can be accessed through the global reference *MainContainer*.

Example, setting the **Highlight** status of a Button component (making it blue) from a callback function:



In-code reference name

Button_1

```
async function myCallback() {
  Controls.Button_1.highlight = true;
}
```

The type of the component reference depends on the component. Looking into the generated *code_generated.js* file shows

```
/**
 * @type {{
 *     Button_1: !FPComponents.Button_A,
 * }}
 */
let Controls = {}
```

This is a JSDOC comment, informing that the *Button_1* reference in the *Controls* object is of type *FPComponents.Button_A*.

This information is automatically read by Visual Studio Code, and used for auto-completion.

The Custom Panel component

The **Custom Panel** UI component can be used to inject HTML files into the app, or as a way to create an empty *<div>* element that can be accessed from *code.js*.