Testing

This document is all about the *Testing plug-in* of the *Espruino Web IDE.* The chapters include a formal and exemplary description of the various testing elements. The Appendices includes a rich collection of working examples that come as default with the plug-in and are for learning by example and for complementing the formal, exemplary description with all the available options on loadable and running example. The formal, exemplary description would have become to abstract with including all the available options.

Table of content

- Table of content
- 1 Overview
- 2 Getting started - Your 1st Application
- 3 Testing Views
- 4 Form View
- 5 Data Point
  - 5.1 Basic Data Point Examples
  - 5.2 Data Point Data Types
  - 5.3 Active vs. Inactive Data Point
  - 5.4 Order of Data Points in definition List
- 6 Test Action
  - 6.1 Basic Test Action Examples
  - 6.2 Test Run with Test Actions executed
  - 6.3 Command Test Action with User Input
  - 6.4 Examples of Command Test Action with User Input
  - 6.5 Form Test Action
  - 6.6 Form Test Action Example
  - 6.7 Order of Test Actions in definition List and Forms
- 7 Image View
  - 7.1 Data Point Image Display Options
    - 7.1.1 N - Number - Display
    - 7.1.2 S - String - Display
  - 7.2 Test Action Image Display Options
    - 7.2.1 B - Button - Display
- 8 Properties
  - 8.1 Description
  - 8.2 Image .jpg
  - 8.3 Active Poll
  - 8.4 Inteval (secs)
  - 8.5 Poll Format
  - 8.6 Project .js
- 9 Elements of the Testing User Interface
  - 9.1 UI Elements of Top Bar of Form and Image Views
  - 9.2 UI Elements of Upper 'Half' Form View
  - 9.3 UI Elements of Lower 'Half' Form View
  - 9.4 UI Elements of Image View - The Canvas
  - 9.5 UI Elements of pop-up for data point image specs

9.6 UI Elements of pop-up for test action image specs
9.7 UI Elements of pop-up for loading a saved testing
9.8 UI Elements of pop-up for saving a testing
9.9 UI Elements of data entry pop-up in image view
10 Testing Operations
11 Transmission Examples
12 Sandbox directory structure
13 Log File: Test Recording
14 Testing Definition File
Appendices
Data Point Examples Explained
Action Point Examples Explained

1 Overview

Testing provides command and data entry and visualisation toolset to interact with Espruino boards through a GUI for testing, monitoring, and controlling. A testing has elements to display pulled values - called *Data Points* - and elements for setting coded or user solicited values and invoking functions with and without parameters - so called *Test Actions* . Data points are shown *Testing Views* as plain text, graphs, and gauges; test actions are shown as button with and without input fields, and whole forms. Testing views can have a *Background Image* defined in the *Properties* to look just like a real control panel.

A testing can be *started*, *paused*, *resumed*, and *stopped*. While running (started), the values get pulled repeatedly on an interval defined in the properties. The pulled values can be recorded - logged - in JSON format in *Log Files* with extension .json. The files are stored in sub-folder of the Espruino IDE configured project folder and can be reused for viewing and post processing either within Espruino IDE graphing feature or editor or any other (visualization) tool.

Testing definitions - the sets of user defined data points and test actions including the related properties - can be named and stored as .json file a sub-folder of the Espruino IDE configured project subfolder and reloaded in and instant at a later time for reuse. Testing can also pass control to another testings to create whole testing chains and user navigable testing suite dialogs.

Under the hood, testing uses the console command and log infrastructure that comes with the Espruino Web IDE. But rather having to type and copy and paste expressions repeatedly into the console pane and recall past commands and entries (which get kind of lost anyway on code upload), data points are updated automatically on intervals for viewing, and test actions are readily available on a click or tap for immediate execution.

When a test is started (in *Active Poll* mode), a function is composed from the expressions of the active data points and transmitted to the Espruino board. On defined *(poll) Intervall*, testing invokes the function, which sends the data point values as evaluated by the expressions for display. *Passive Poll* mode is when the code uploaded to the board includes the code to gather the data point values and send them back to testing for display on time terms defined too by this (user written) code. The code has to return the data point values only in *JSON Poll Format* that matches the test's data point definitions. Support for Passive poll mode is part of a next version. *Active Poll (mode)*, *(Pull) Intervall (time in seconds)*, and *Poll Format* are part of the test's properties.

2 Getting started - Your 1st Application

3 Testing Views

Testing has two views:

1. *Form* view - shows by *default*
2. *Image* or graphics view - shows *on-demand*

The form view serves also the purpose of defining the data points and actions points. The image view uses defined data points and test actions and has functionality to place them on the canvas. The can have a background image defined in the properties, which is is then overlaid with the data point and test actions representations.

User can switch anytime between form and image view, but only the visible view is updated with pulled test data. Switching to any view show the last shown update until next pull and update happens. Pull interval can be set in *Properties*.

Both views show *data points* - values pulled from Espruino board - and *test actions* - user controls with or without input, such as buttons and forms - that send commands to the Espruino board.

4 Form View

5 Data Point

Note: 'Data Point' and 'Datapoint' are used synonymously.

Data point is - any Espruino-valid - Javascript expressions that bottom line send their result value from Espruino board back to testing.

A data point can be very very simple - such as a variable reference - to quite a complex - such as a elaborate function. The data point value is shown in the form view as a line or annotation in a *Chart* and as text or graphic - such as bar or gauge - in the image view.

A data point's work can be 'done' with just the console by typing a expression into the console pane of the IDE and look at the printed result value. But data points do that way more efficient for many at a time and - most important - repeatable on a interval basics, and the result visualized not just as a text but in more comprehensive ways, such as charts, graphs, bars, gouges, etc - even custom.

The most simple data point is a global variable name, such as *myVar*. Another simple datapoint is a function invocation, such as *myFunction()*. Essential to a data point expression is that something is returned to testing for charting and logging other than *undefined* or *null* For that reasn, the expected return types has to be specified in the data point definition.

5.1 Basic Data Point Examples

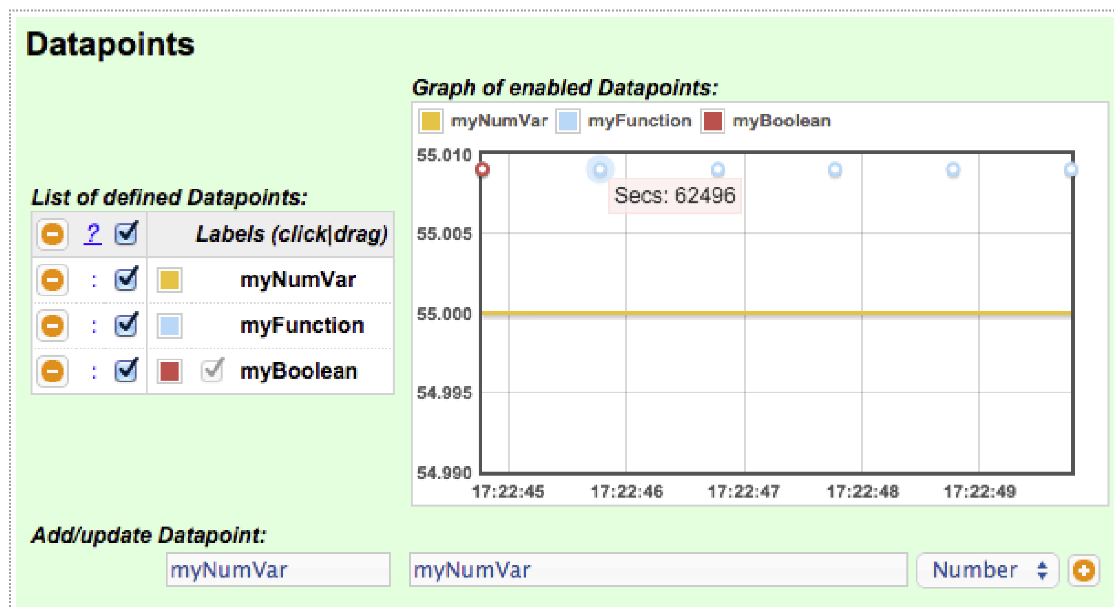
For the basic data point examples to work, assume below code snippet is uploaded to the board. Notice that *myFunction() {...}* flashes the (red) LED1 for 33 ms. By that we know when the function is invoked: every time data is pulled, (red) LED1 flashes. (You do not need to type the code, just copy it from below, paste it into the Web IDE Editor - the right page - and upload it.)

```
// Code to co op w/ Basic Data Points Examples

var myNumVar = 55;

function myFunction() {
  LED1.set(); setTimeout('LED1.reset()',33);
  return "Secs: " + Math.floor(getTime());
}

var myBoolean = true;
```

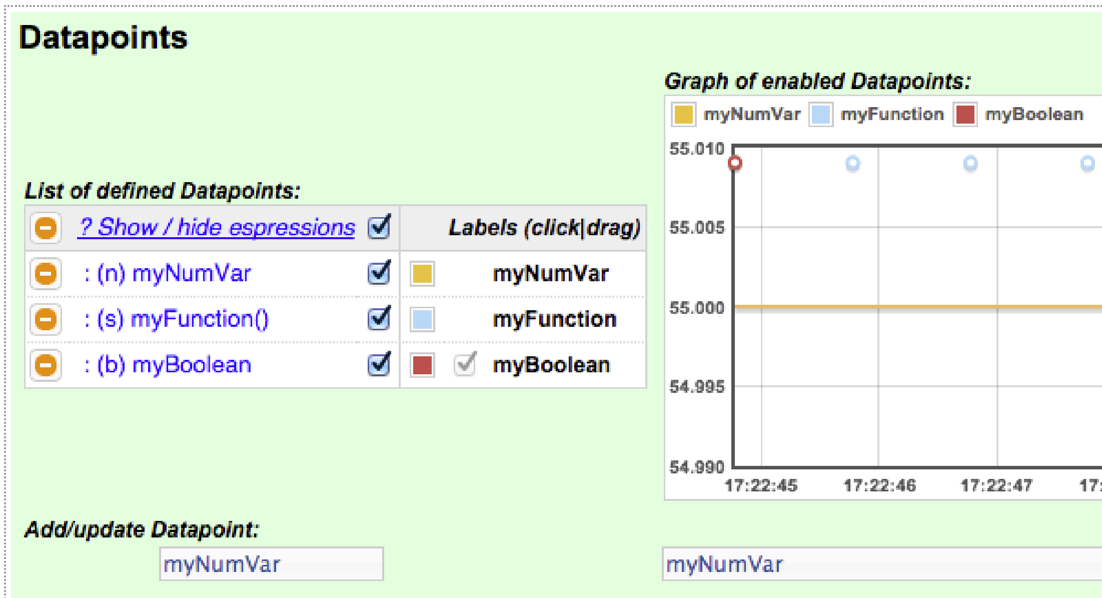
Basic data point examples:

Label	*Expression*	*Type*
myNumVar	**myNumVar**	Number
myFunction	**myFunction()**	String
myBoolean	**myBoolean**	Boolean

This is how the *List of defined Datapoints* looks in testing form view (after having run for a few seconds. Note that annotation - non-numeric values - show the value when hovering over the annotation circle):



This is how the list *with expanded expression column* looks:



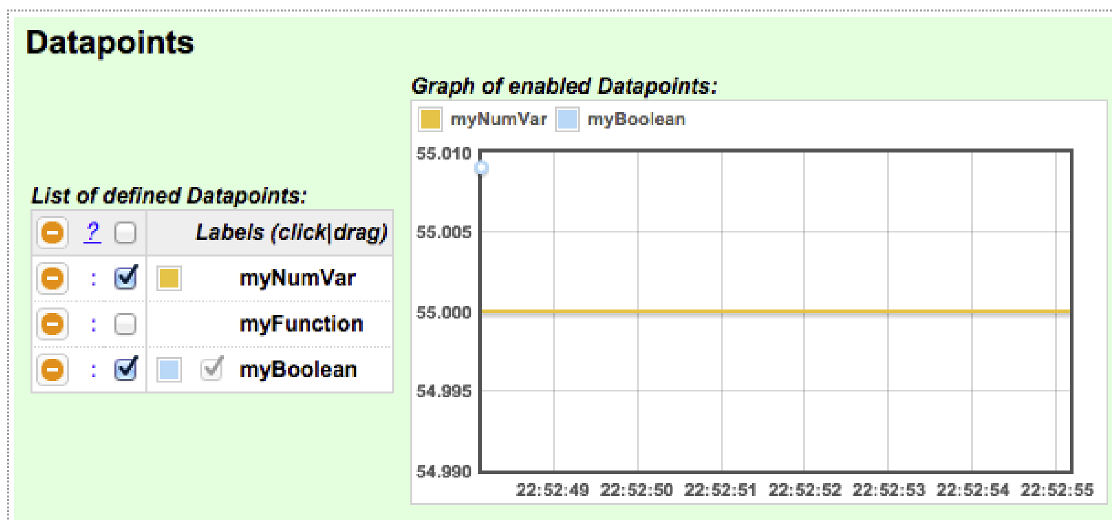
5.2 Data Point Data Types

Type	Description
Number	Any numeric value
String	A string value
Boolean	A boolean value

5.3 Active vs. Inactive Data Point

A data point can be (temporarily) deactivated for form or image view (rather than deleted from the list of data point definitions altogether). A data point that is not active in either of the views is not pulled nor logged. (Active for recording / logging only is not implemented yet).

This is how the list of defined data points and *Graph of enabled data points* looks like after deactivating (unchecking) *myFunction* data point and running the test for a few seconds. The annotation and legend entry for the *myFunction* are absent in the graph (and as well the data in the transmission and logging/recording).



5.4 Order of Data Points in definition List

Data points can be reordered in the list (using drag-drop). The order defines the execution sequence within Espruino board on pull, the order of the properties / values in the object or array return to testing, the sequence of lines in chart, the update sequence of the displays in the image view, and the properties / values in the recorded object or array in the loged files.

6 Test Action

Note: 'Test Action' and 'Testaction' and 'Action' are used synonymously.

Test Action is - any valid - JS expressions that makes Espruino do something on-demand - such as on clicking on a button - rather than on an interval as a data point does. A test actions may expect the user to enter one to n values before sending the expression with the values for execution to the Espruino board. Execution can be a variable assignment or function invocation with or without parameter values provide by user entry.

Test actions can be 'done' to by entering the expressions directly into the console, but it is just not that efficient and repeatable as through Testing GUI and also without any formal user input validation.

The most simple example for a test action is of type *command* and is the invocation of a function with no or only (hard-)coded parameters or the (declaration and) assignment of a (hard-)coded value to a variable.

A bit more elaborate test actions are the assignments of values to variables and parameters of function invocations with parameter, where the user is required to enter some the values first before the text action can be execution. User input values are formally validated. Therefore, a test action specification with user input includes data type specifications for every such user input specification. When the test action is a variable value assignment from a user input, then the value's data typ given by the test action type, such as *Number*, *String*, or *Boolean*.

Single test actions can be combined or grouped into a *Form Test Action* . A form test action is represented by a button. Clicking on that button in form views after entering values for all the form included individual test actions fires all those form included individual test actions. The buttons for the form included test actions are not show. For more details see Form Test Action .

In image views, *any* test action that requires user input, show on button click a pop-up to the user for entering the specified user values.

6.1 Basic Test Action Examples

For the basic test action examples to work, assume this code snippet is uploaded to the board (Notice the addions of *var myString = ...* and *mySetFunct() {...* compared to the code for the Basic Data Point Examples):

```
// Code to co op w/ Basic Test Action Examples

var myNumVar = 55;

function myFunction() {
```

```
  LED1.set(); setTimeout('LED1.reset()',33);
  return "Secs: " + Math.floor(getTime());
}

var myBoolean = true;

var myString = "THE string.";

function mySetFunct() {
  myNumVar = 44;
  myBoolean = false;
  myString = "THAT string.";
}
```

Basic test action examples:

Label	*Expression*	*Type*
Set_myNumVar	**myNumVar**	Number
Set_myString	**myString**	String
Set_myBoolean	**myBoolean**	Boolean
Invoke_myFunction	**myFunction()**	Command

This is how the *List of defined Actions* looks in testing form view:

Actions

List of Actions:

Label (click|drag) ?	*Value*	*Action (dry run)*
Set_myNumVar :	0	▶ Set_myNumVar
Set_myBoolean :	☐	▶ Set_myBoolean
Set_myString :		▶ Set_myString
Invoke_mySetFunct :		▶ Invoke_mySetFunct

Add/update Action:

Set_myNumVar | myNumVar | Number

This is how the list *with expanded expression column* looks:

Actions

List of Actions:

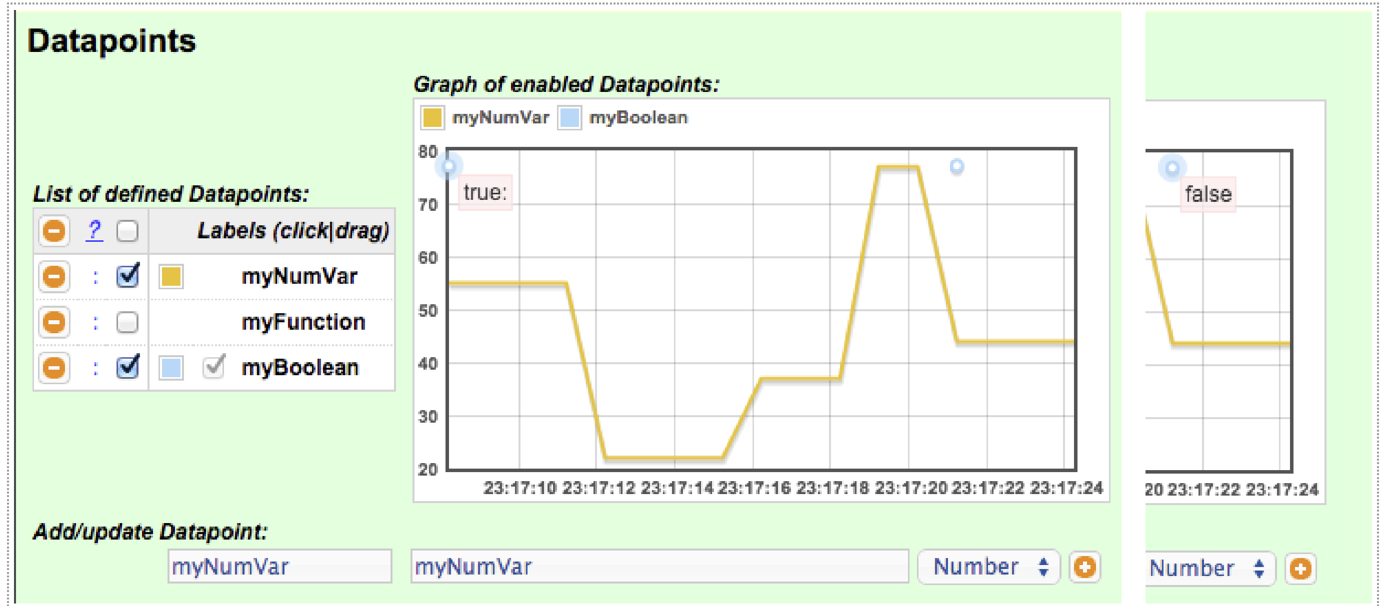
Label (click|drag)	*? Show / hide expressions*	*Value*	*Action (dry run)*
Set_myNumVar	: myNumVar	0	▶ Set_myNumVar
Set_myBoolean	: myBoolean	☐	▶ Set_myBoolean
Set_myString	: myString		▶ Set_myString
Invoke_mySetFunct	: mySetFunct()		▶ Invoke_mySetFunct

Add/update Action:

Set_myNumVar | myNumVar | Number

6.2 Test Run with Test Actions executed

This is how the chart with the enabled data points looks after running for 16 seconds and clicking on test action buttons - the buttons with the arrow triangles pointing to the right - and entering different values where needed. The test is defined as outlined in Basic Data Point Examples and Basic Test Action Examples with related code just uploaded to the board.



Test actions and related inputs were ('~:##' denotes second on time line with '~' = 23:17):

1. ~:08 - **Started test** with *myNumVar=55* and *myBoolean=true* (initial values): myNumVar graph line starts at level 55, and myBolean annotation shows true.
2. ~:12 - Clicked **Set_myNumVar** button with *22* entered in related input field: myNumVar graph line moves down to level 22 and no change in myBoolean annotation.
3. ~:16 - Clicked **Set_myNumVar** button with *37* entered in related input field: myNumVar graph line moves up to level 37 and no change in myBoolean annotation.
4. ~:19 - Clicked **Set_myNumVar** button with *77* entered in related input field: myNumVar graph line moves down to level 77 and no change in myBoolean annotation.
5. ~:21 - Clicked **Invoke_mySetFunct()** button: myNumVar graph line moves down to level 44 and myBoolean annotation shows change to false.
6. ~:24 - **Stopped test with** *myNumVar* at level *44* and *myBoolean* is *true*.

6.3 Command Test Action with User Input

The expression for a command - function invocation - *with one-to-many user input values* includes a comma separated list of parameter type and parameter name specifications of the form `{{type:name}}` , where *type* can have the values `n` | `s` | `b` for *number, string,* and *boolean*, and where *name* is the label and internal variable name of the user input field in the form view and in the pop-up in the image view.

6.4 Examples of Command Test Action with User Input

The previously know example code has to be extended with *myFlexSetF()* function and uploaded to the board in order co op with the first example and will look like:

```
// Code to co op also w/ Command Test Action with User Input example

var myNumVar = 55;

function myFunction() {
  return "Secs: " + Math.floor(getTime());
}

var myBoolean = true;

var myString = "THE string.";

function mySetFunct() {
  myNumVar = 44;
  myBoolean = false;
  myString = "THAT string.";
}

function myFlexSetF(nmbr,bln,strng) {
  myNumVar  = nmbr;
  myBoolean = bln;
  myString  = strng;
}
```

Examples of command test action with user input (For to work with specified code, add only the first example to the list of test actions):

Label	*Expression*	*Type*
Invoke_myFlexSetF	**myFlexSetF({{n:nmbr}},{{b:bln}},{{s:strng}})**	Command
stepper1_goTo_X_Y	**st1.goTo({{n:x}},{{n:y}})**	Command

With the (first) command test action with user input added, the list of (defined) test actions looks like this:

Actions

List of Actions:

Label (click|drag)	*? Show / hide expressions*	***Value***	***Action (dry run)***
Set_myNumVar	: myNumVar	0	▶ Set_myNumVar
Set_myBoolean	: myBoolean	☐	▶ Set_myBoolean
Set_myString	: myString		▶ Set_myString
Invoke_mySetFunct	: mySetFunct()		▶ Invoke_mySetFunct
Invoke_myFlexSetF	: myFlexSetF({{n:myNmbr}},{{b:myBln}},{{s:myStrng}})	0	**myNmbr >...**
		☐	**myBln >...**
			myStrng >...
			▶ Invoke_myFlexSetF

Add/update Action:

Invoke_myFlexSetF | myFlexSetF({{n:myNmbr}},{{b:myBln}},{{s:myStrng}}) | Command

Notice the three entry fields labeled - *myNbr, myBln,* and *myStrng* - generated by the *{{type:name}}* parameter specifications. The less-then (<) character and tripple-dots (...)

indicate that this entry field represents an input parameter of the function. function

Clicking on *Invoke_myFlexSetF* button will formally validate the related three inputs values and then invoke the (global) *myFlexSetF()* with them.

In order for the second example to work, *st1* - for *Stepper Motor 1* - must be a global variable referencing an object hat understands the method *.goto(x,y)*.

6.5 Form Test Action

Form test action is a command type test action that includes a group of 'individual' test actions - variable assignment and commands with and without user input.

The form test action is *always* of type *Form* and its label always *ends with an underscore* (_) - the only one in the label.

The individual test actions - variable assignments and function invocations with and without user input value - that belong to the form have the *form's label as prefix* of their label. Test actions belonging to a particular form show always together in the list of test actions and are followed immediately by the form test action. They do *not* show a button. The form's button executes the included test actions all at once one after the other (in the sequence as listed).

In image views, where the form test action is shown as a button, on click of the button a pop-up is shown to the user to enter the user values all at once and then submit them.

Note that forms can *never* include a test action of type *Form*.

6.6 Form Test Action Example

As you notice, the form (can and in this example does exclusively) reuse the existing (global) variable assignments and (global) function invocation(s) as earlier individually executable test actions used. (Therefore, no code addition is required to execute this form test action.)

Label	*Expression*	*Type*
form1_myNumVar	myNumVar	Number
form1_myBoolean	myBoolean	Boolean
form1_myString	myString	String
form1_myFunction	myFunction()	Command
form1_	form1 - optional comment	Command

With the form test action added to the test actions, the list looks like this:

Actions

List of Actions:

Label (click|drag) ?	Value	Action (dry run)
Set_myNumVar :	0	▶ Set_myNumVar
Set_myBoolean :	☐	▶ Set_myBoolean
Set_myString :		▶ Set_myString
Invoke_mySetFunct :		▶ Invoke_mySetFunct
Invoke_myFlexSetF :	0	myNmbr >...
	☐	myBln >...
		myStrng >...
		▶ Invoke_myFlexSetF
form1_myNumVar :	0	form1_myNumVar
form1_myBoolean :	☐	form1_myBoolean
form1_myString :		form1_myString
form1_myFunction :		form1_myFunction
form1_ :	form1_ using existing test actions	▶ SUBMIT form1_

Add/update Action:

form1_myFunction | myFunction() | Command

Clicking on *form1_* button will formally validate all related inputs values and then execute every individual test action in the form at once, one after the other in the sequence as listed.

6.7 Order of Test Actions in definition List and Forms

Action points can be reordered in the list (using drag-drop). But for execution only the order of individual test actions within each form matters. The sequence within the form defines the execution sequence on the Espruino board.

Drag-drop of individual test actions of a form is limited to the form scope. Moving the form test action includes the moving of the included individual test actions as a whole.

7 Image View

7.1 Data Point Image Display Options

7.1.1 N - Number - Display

```
format (sizs[1]):

- number, for example #, .#, #.# where # are digits and indicate number indicates
          digits before an after the decimal point, for example 10.2:
          - format number as integer, .decimalFraction, and integer.decimalFraction
```

```
          with proper rounding and display centered. Not fitting numbers are still
          displayed and will therefore not show adjusted as expected.

      - "My crazy number is {{-5.2}}, and this is fine with me!"
```

7.1.2 S - String - Display

```
format (sizs[1]):

- number, for example 5 - max length of string, longer will be cut to #-3...
          - String is displayed x/y-centered with maximum 'number' of characters.
            Longer strings are truncated to 'number - 3' and suffixed with '...'

- string: template - for example:

  - x/y-centered examples:

    - "=: {{*}}" --- '=' at first position:
          - '=' is replaced with label, for example: 'myString'
          - '*' is replaced with value, for example: 'THE String'
          - displays:
       myString: THE String

    - "{{5}} (=)" --- '(=)' at last position:
          - '=' is replaced with label, for example: 'myString'
          - '5' is replaced with max 5 characters of value, for example: 'THE S'(tring)
       THE S (myString)


  - x/y-left adjusted examples:

    - "^..." where '^' at the begin does left adjust, and remaining
             pattern string (...) is processed the same way as


min|other (sizs[3]):

- 0 normal font-weight and normal font-style
- 1 bold font-weight  (2^0 bit - bitwise AND - & 1)
- 2 italic font-style (2^1 bit - bitwise AND - & 2)
- 3 bold font-weight and italic font-style
```

7.2 Test Action Image Display Options

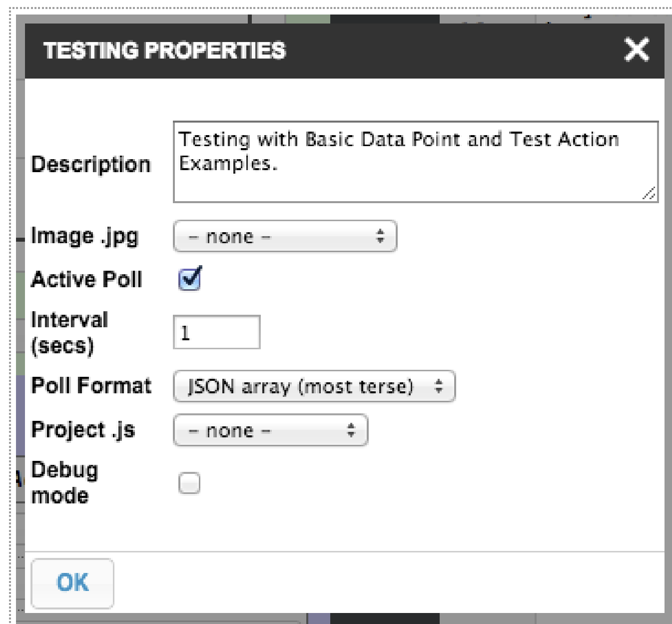
7.2.1 B - Button - Display

```
format (sizs[1]):

- number, for example #, .#, #.# where # are digits and indicate number indicates
          digits before an after the decimal point, for example 10.2:
          - format number as integer, .decimalFraction, and integer.decimalFraction
            with proper rounding and display centered. Not fitting numbers are still
            displayed and will therefore not show adjusted as expected.
- string: template - for example:
  - x/-y centered examples:
    - "=: {{*}}" ('=' at first position):
          - '=' is replaced with label, for example: 'myString'
          - '*' is replaced with value, for example: 'THE String'
          - displayed: 'myString: THE String'
    - "{{5}} (=)" ('(=)' at first position):
          - '=' is replaced with label, for example: 'myString'
          - '*' is replaced with value, for example: 'THE String'
          - displayed: 'THE String (myString)'
    - "My crazy number is {{-5.2}}, and this is fine with me!"
  - x/y-left adjusted examples:
    - "^..." where '^' at the begin does left adjust, and remaining
```

pattern string (...) is processed the same way as

8 Properties

Clicking on the **Properties** button in the testing form view shows the *Properties Dialog* in a pop-up.



8.1 Description

8.2 Image .jpg

8.3 Active Poll

If checked - active polling:

- Testing creates the data point value poll (send) function on test start based on the active data point specifications and uploads it to the Espruino board
- Invokes the poll function on specified intervall.
- Renders the incoming data in form or image view

If unchecked - passive 'polling':

- User defines data point value send ('pull') function including send interval handling and uploads it with or separate of the application code to Espruino board
- Testing listens for incoming data and renders it in form or image view

With passive 'polling', only JSON (full) poll format is supported and sent data type / format has to match data point definitions. Data for undefined / unknown or inactive datapoints is ignored. Data type / format mismatching data will will produce errors.

Note: passive polling is not supported with this version.

8.4 Inteval (secs)

Interval (time in seconds - or fractions there of) defines the invocation rate for the data point value poll (send) function / sampling rate for the rendering.

8.5 Poll Format

There are three poll formats (data formats) to choose from:

- JSON array (most terse) - [5, "String", true]
- JSON object optimized - { "_0":5, "_1":"String", "_2":true }
- JSON object - { "myNumber":5, "myString":"String", "myBoolean":true }

As obvious from a look at the transmitted data in the console, the *JSON array* poll format is the most efficient one. Pulled data shown in section Transmission Examples is created by Basic Data Point Examples .). When logging is on, poll format information is included at the begin of the log to enable proper parsing and consummation of the logged data.

Selected poll format has also an impact on the logging / recording of the pulled data as shown in section Log File: Test Recording .

Transmitted data with *Poll Format: JSON array (most terse)*, which is the default and has the best performance / least band with consumption:

8.6 Project .js

Loads javascript file from Sandbox projects folder into Web IDE editor and uploads it also to Espruino board.

9 Elements of the Testing User Interface

A great deal went into the user interface to support it use and provide help (at least as seen from a coder's point of view... 'creatives' will for sure be able to point out room for improvement though). Most element have a hover activated tooltip (with HTML title="..." attribute implemented). Move the (mouse) pointer from outside over an element and rest until the tooltip shows. The next paragraphs show the UI, name the elements and provide a brief description.

The *Testing UI* occupies the same real estate as the console and the frame around it of Espruino's Web IDE. The two form and image testing views and the console are like a stack of cards of which only one can be 'on the top' and visible at one time. Switching either either between the console and the default (form) or last shown testing view or between both testing form and image view.

When Web IDE starts up, the console is always the first view visible. Switching to the testing view and back to console happens by clicking the 'code' () and'eye'/'watch' icons in the bottom left of the frame. Switching between the form and image testing views happens by clicking on view labeled radio button in the bar at the top of either view.

9.1 UI Elements of Top Bar of Form and Image Views

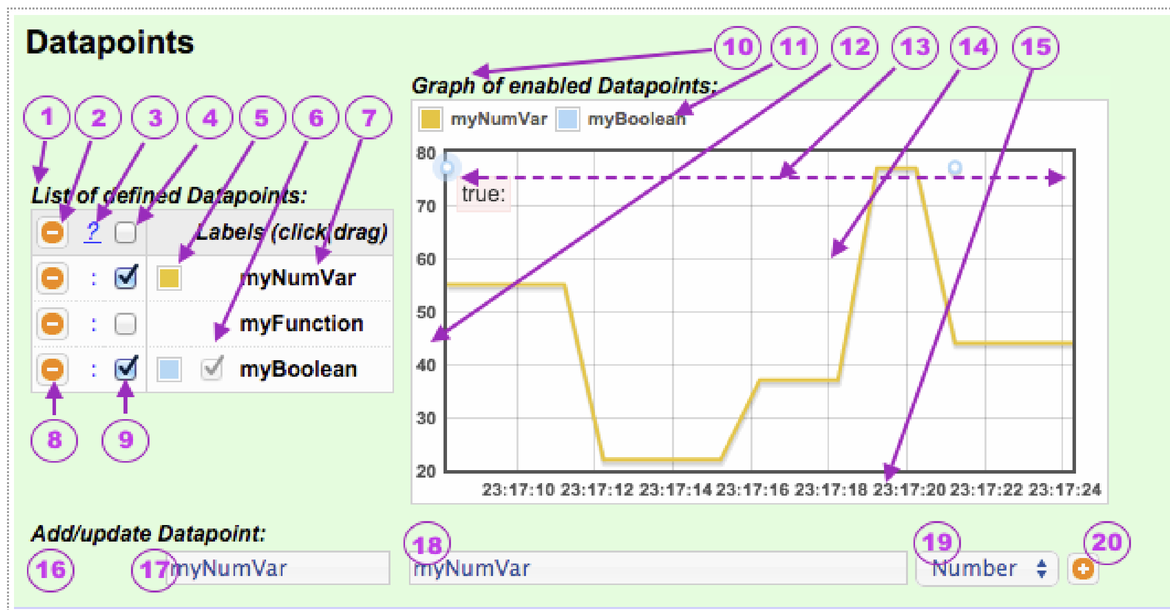
The top bar of both form and image views are practically identical. The bars host the high level navigation and function controls, such as load and store a testing, start and stop it, etc.

UI Elements of view top bar are:

1. Load saved testing specification (Shows File Dialog pop-up) - Button
2. Name of the test (file name) - Display
3. Save testing specification (Shows Save Dialog pop-up) - Button
4. Start test - Button
5. Stop test - Button
6. Reset data point values history / chart data - Button
7. Enable logging of data point values - Checkbox
8. Enable single shot: test pulls data only once and then stops - Checkbox
9. Switch between Form and Image view - Radio button
10. Show Properties pop-up - Button

9.2 UI Elements of Upper 'Half' Form View

The upper half of the form view is also called *datapoint section*. It hosts the list of (defined) data points, the chart of the enabled data points, and the edit and update controls for data points.



UI Elements of data points area are:

1. List of defined data points - Display
2. Delete all data points in list (no confirmation prompt) - Button
3. Hide / show data point expressions w/ data types - Sensitive area
4. Enable / disable *all* data points in graph (chart) - Checkbox
5. Show which color data pont ha in graph (chart) - Display
6. Show which data points are display on the image view - Display-Only Checkbox
7. Label (unique key/name) of data point (click to edit/update, drag-drop to change order - Sensitive area
8. Delete particular data point (no confirmation prompt) - Button
9. Enable / disable particular data point in graph (chart) - Checkbox
10. Graph (chart / *flotchart*) of enabled data points - Display

11. Mapping of line and notification color to enabled data point - Display
12. Auto scaling / grid-ing data point value axis - Display
13. Change notification area for non-numeric values - Display
14. Chart canvas for data line plotting in autoscaling grid - Display
15. Auto scaling / grid-ing time value axis - Display
16. Data entry area for create and edit/update data points - Display
17. Label (unique key/name) entry field for Data point - Entry field
18. Expression entry field for data point - Entry filed
19. Data type drop-down selection for data point - Drop-down
20. Add/Update data point (new label is add, existing is update) - Button

9.3 UI Elements of Lower 'Half' Form View

The lower half of the form view is also called *test action section*. It hosts the list of (defined) test actions and the edit and update controls for test actions.

UI Elements of test actions area are:

1. List of defined test actions - Display
2. Delete all test actions - Button
3. Delete particular test action - Button
4. Show/hide expressions - Sensitive area
5. Label of assignment test action of form1_ (click for edit/update, drag/drop to change order) - Display
6. Form test action label (click for edit/update) - Display
7. Command test action expression with parameter spec - Display
8. Form test action comment (optional) - Display

9. Input field of assignment test action of form 1_ - Input field
10. Form test action submit button - Button
11. Command test action with input execution button - Button
12. Command test action input parameter field - Input field
13. Command test action input parameter label - Display
14. Assignment test action input field - Input field
15. Assignment test action execution button - Button
16. Expression build / verification only when checked - Checkbox
17. Data entry area for create and edit/update test actions - Display
18. Label (unique key/name) entry field for test action - Entry field
19. Expression entry field for test action - Entry filed
20. Test action type drop-down selection for test action - Drop-down
21. Add/Update test action (new label is add, existing is update) - Button

9.4 UI Elements of Image View - The Canvas

The Image view canvas is - as the word says - a canvas and has therefore no elements by itself or just a background image when defined in the properties. The canvas is used for painting the text and graphical representation elements of the image enabled data points and test actions (text labels, bars, gauges, dials, dial hands, etc. and buttons).

9.5 UI Elements of pop-up for data point image specs

9.6 UI Elements of pop-up for test action image specs

9.7 UI Elements of pop-up for loading a saved testing

9.8 UI Elements of pop-up for saving a testing

9.9 UI Elements of data entry pop-up in image view

10 Testing Operations

11 Transmission Examples

The next sections show what is transmitted between Espruino Board and Web IDE / testing plug-in based on Basic Data Point Examples .

Transmitted data with *Poll Format: JSON array*:

```
>echo(0);
<<<<<[55,"Secs: 392",true]>>>>>
<<<<<[55,"Secs: 393",true]>>>>>
<<<<<[55,"Secs: 394",true]>>>>>
=undefined
>
```

Transmitted data with *Poll Format: JSON object optimized*:

```
>echo(0);
<<<<<{"_0":55,"_1":"Secs: 443","_2":true}>>>>>
<<<<<{"_0":55,"_1":"Secs: 444","_2":true}>>>>>
<<<<<{"_0":55,"_1":"Secs: 445","_2":true}>>>>>
=undefined
>
```

Transmitted data *Poll Format: JSON object*:

```
>echo(0);
<<<<<{"myNumVar":55,"myFunction":"Secs: 543","myBoolean":true}>>>>>
<<<<<{"myNumVar":55,"myFunction":"Secs: 544","myBoolean":true}>>>>>
<<<<<{"myNumVar":55,"myFunction":"Secs: 545","myBoolean":true}>>>>>
=undefined
>
```

The following loc excerpt cover a 16 seconds run (with 1 second pull-interval) of Test Run with Test Actions executed .

```
>echo(0);
<<<<<[55,true]>>>>>
<<<<<[55,true]>>>>>
<<<<<[55,true]>>>>>
<<<<<[55,true]>>>>>
=undefined
>Troas();
<<<<<[22,true]>>>>>
=undefined
>Troas();
<<<<<[22,true]>>>>>
=undefined
>Troas();
<<<<<[22,true]>>>>>
=undefined
>Troas();
<<<<<[22,true]>>>>>
=undefined
>echo(0);
=undefined
>Troas();
<<<<<[37,true]>>>>>
=undefined
>Troas();
<<<<<[37,true]>>>>>
=undefined
>Troas();
<<<<<[37,true]>>>>>
=undefined
>echo(0);
=undefined
>Troas();
<<<<<[77,true]>>>>>
=undefined
>Troas();
<<<<<[77,true]>>>>>
=undefined
>echo(0);
=undefined
>Troas();
<<<<<[44,false]>>>>>
=undefined
>Troas();
<<<<<[44,false]>>>>>
=undefined
>Troas();
<<<<<[44,false]>>>>>
```

```
=undefined
>Troas();
<<<<<[44,false]>>>>>
=undefined
>echo(1);
```

12 Sandbox directory structure

13 Log File: Test Recording

When *Log* checkbox is checked in form test view, pulled data points are logged / recorded in a .json file. The file has the same name as the testing definition suffixed with a *_YYYY_MM_DD__hh_mm_ss* timestamp (= the - local - time the test started). The file is stored in the *testinglog* folder of the user named Espruino project / sandbox folder. The timestamp entries in the log / recording are UTC, though.

Logged data / recording with *Poll Format: JSON array (most terse)* as file *BasicTestingExamples_2015_04_23__14_57_23*:

```
[{"UTC":1429826243022,"testing":"BasicTestingExamples","version":"1v74","pollFormat":"arrJSON",
  "receive":[["myNumVar","number"],["myFunction","string"],["myBoolean","boolean"]],
  "description":"Testing with Basic Data Point and Test Action Examples."}
,{"UTC":"1429826244055","data":[55,"Secs: 392",true]}
,{"UTC":"1429826245051","data":[55,"Secs: 393",true]}
,{"UTC":"1429826246059","data":[55,"Secs: 394",true]}
]
```

Logged data / recording with *Poll Format: JSON object optimized* as file *BasicTestingExamples_2015_04_23__14_58_13*:

```
[{"UTC":1429826293995,"testing":"BasicTestingExamples","version":"1v74","pollFormat":"optJSON",
  "receive":[["myNumVar","number"],["myFunction","string"],["myBoolean","boolean"]],
  "description":"Testing with Basic Data Point and Test Action Examples."}
,{"UTC":"1429826295016","data":{"_0":55,"_1":"Secs: 443","_2":true}}
,{"UTC":"1429826296018","data":{"_0":55,"_1":"Secs: 444","_2":true}}
,{"UTC":"1429826297020","data":{"_0":55,"_1":"Secs: 445","_2":true}}
]
```

Logged data / recording with *Poll Format: JSON object* as file *BasicTestingExamples_2015_04_23__14_59_53*:

```
[{"UTC":1429826393899,"testing":"BasicTestingExamples","version":"1v74","pollFormat":"objJSON",
  "receive":[["myNumVar","number"],["myFunction","string"],["myBoolean","boolean"]],
  "description":"Testing with Basic Data Point and Test Action Examples."}
,{"UTC":"1429826394921","data":{"myNumVar":55,"myFunction":"Secs: 543","myBoolean":true}}
,{"UTC":"1429826395923","data":{"myNumVar":55,"myFunction":"Secs: 544","myBoolean":true}}
,{"UTC":"1429826396924","data":{"myNumVar":55,"myFunction":"Secs: 545","myBoolean":true}}
]
```

14 Testing Definition File

```
{
  "version": "1v77",
  "testDescr": "",
  "imageUrl": "",
```

```
  "testMode": "Form",
  "pollInterval": 1,
  "activePoll": true,
  "pollFormat": "arrJSON",
  "testProject": "",
  "testDebug": true,
  "dataPoints": [
    {
      "label": "myNumVar",
      "expression": "myNumVar",
      "type": "number",
      "enabled": true,
      "x": 0,
      "y": 0,
      "display": "N",
      "onImg": false,
      "sizs": [
        55,
        2,
        12,
        0,
        90,
        -1,
        -1
      ]
    },
    {
      "label": "myFunction",
      "expression": "myFunction()",
      "type": "string",
      "enabled": true,
      "x": 0,
      "y": 0,
      "display": "S",
      "onImg": false,
      "sizs": [
        "off",
        "*",
        14,
        -1,
        -1,
        -1,
        -1
      ]
    },
    {
      "label": "myBoolean",
      "expression": "myBoolean",
      "type": "boolean",
      "enabled": true,
      "x": 90,
      "y": 45,
      "display": "I",
      "onImg": false,
      "sizs": [
        false,
        5,
        14,
        -1,
        -1,
        -1,
        -1
      ]
    }
  ],
  "testActions": [
    {
      "label": "Set_myNumVar",
      "expression": "myNumVar",
      "type": "number",
      "x": 0,
```

```
      "y": 0,
      "display": "B",
      "onImg": false,
      "sizs": [
        "",
        0,
        12,
        0,
        0,
        -1,
        -1
      ]
    },
    {
      "label": "Set_myBoolean",
      "expression": "myBoolean",
      "type": "boolean",
      "x": 0,
      "y": 0,
      "display": "B",
      "onImg": false,
      "sizs": [
        "",
        0,
        12,
        0,
        0,
        -1,
        -1
      ]
    },
    {
      "label": "Set_myString",
      "expression": "myString",
      "type": "string",
      "x": 0,
      "y": 0,
      "display": "B",
      "onImg": false,
      "sizs": [
        "",
        0,
        12,
        0,
        0,
        -1,
        -1
      ]
    },
    {
      "label": "Invoke_mySetFunct",
      "expression": "mySetFunct()",
      "type": "command",
      "x": 0,
      "y": 0,
      "display": "B",
      "onImg": false,
      "sizs": [
        "",
        0,
        12,
        0,
        0,
        -1,
        -1
      ]
    },
    {
      "label": "Invoke_myFlexSetF",
      "expression": "myFlexSetF({{n:nmbr}},{{b:bln}},{{s:strng}})",
      "type": "command",
```

```
        "x": 0,
        "y": 0,
        "display": "B",
        "onImg": false,
        "sizs": [
          "",
          0,
          12,
          0,
          0,
          -1,
          -1
        ]
      },
      {
        "label": "form1_myNumVar",
        "expression": "myNumVar",
        "type": "number",
        "x": 0,
        "y": 0,
        "display": "B",
        "onImg": false,
        "sizs": [
          "",
          0,
          12,
          0,
          0,
          -1,
          -1
        ]
      },
      {
        "label": "form1_myBoolean",
        "expression": "myBoolean",
        "type": "boolean",
        "x": 0,
        "y": 0,
        "display": "b",
        "onImg": false,
        "sizs": [
          "",
          0,
          12,
          0,
          0,
          -1,
          -1
        ]
      },
      {
        "label": "form1_myString",
        "expression": "myString",
        "type": "string",
        "x": 0,
        "y": 0,
        "display": "B",
        "onImg": false,
        "sizs": [
          "",
          0,
          12,
          0,
          0,
          -1,
          -1
        ]
      },
      {
        "label": "form1_myFunction",
        "expression": "myFunction()",
```

```
      "type": "command",
      "x": 0,
      "y": 0,
      "display": "B",
      "onImg": false,
      "sizs": [
        "",
        0,
        12,
        0,
        0,
        -1,
        -1
      ]
    },
    {
      "label": "form1_",
      "expression": "form1 - optional comment",
      "type": "command",
      "x": 0,
      "y": 0,
      "display": "S",
      "onImg": false,
      "sizs": [
        "",
        0,
        12,
        0,
        0,
        -1,
        -1
      ]
    }
  ]
}
```


Appendices

Data Point Examples Explained

Action Point Examples Explained

[End of Doc]