> This is a markdown preview of STL00318PB7 SA Technical Screen specification.md.
> It uses css from GitbookIO picked from Typora markdown editor, google-code-prettify, mermaid and Parsedown
> Click here to download original

STL00318PB7 Screen implementation

Introduction

This document describes the screen JSON, and what built-in tweaks that are available in SA.

Document revisions

STL00318PB3, 2019-05-08, Martin Dahlberg using https://www.typora.io/

Rev	Name	Comment	Date
PA1	Martin Dahlberg	Initial draft	2016-10-19
PA2	Martin Dahlberg	Formalized screen JSON	2016-12-15
PA3	Kana Omichi	Translated	2016-12-26
PA4	Martin Dahlberg	action print+script updated. screen.start deleted	2017-01-30
PA5	David Holmin	Updated fieldtype "grid" and "select"	2017-03-09
PA6	Martin Dahlberg	Updated source "table" , fieldtype "grid" and "select"	2017-03-10
PA7	Lars-Åke Berg	Updated "keyboard" in fieldtype input.	2017-09-07
PA8	Fredrik Johansson	Updated fieldtype "select" with new type ('input').	2017-09-12
PA9	Lars-Åke Berg	Old "keyboard" syntax removed for fieldtype input.	2017-09-13
PA10	Fredrik Johansson	Some corrections.	2017-09-13
PA11	Fredrik Johansson	Added 'qtyscreen' attribute details	2017-10-12
PA12	Fredrik Johansson	Added actions 'formatv2' and 'printv2'	2017-12-22
PA13	Martin Dahlberg	Added information about actions screen and video	2018-02-26
PA14	Martin Dahlberg	Updated documentation about lua_* events mostly	2018-03-02
PA15	Martin Dahlberg	Made most lua_ events post scripts, i.e. they can only modify the results/behaviors after the builtin functions are applied.	2018-03-12
PA16	Martin Dahlberg	Updated?	2018-03-21
PA17	Martin Dahlberg	Fixed version PA17	2018-05-10
A	Martin Dahlberg	Minor changes for lua_isa and lua_done	2018-12-04
PB1	Martin Dahlberg	Updates for 5.1.0	2018-12-07
PB2	Martin Dahlberg	bgcolorColumn in grid and select list, horizonalScroll in grid	2018-12-10
PB3	Martin Dahlberg	js_* events described	2019-05-08
PB4	David Holmin	keyboard attribute for grid, list and input	2019-09-16
PB5	Martin Dahlberg	AEP_js.js, AEP popup dialog, multiple columns in select list/dropdown.	2019-09-18
PB6	Olof Millberg	Added showKeyboard attribute to screen	2019-09-23
PB7	Martin Dahlberg	Corrected mistake in ws_msgs	2019-10-18

Related documents

STL00303 - Workspace JSON specification

Reserved attributes

There are some attributes reserved for future use by SATO.

_*, sdk_*, js_*, lua*

Attributes starting with a capital letter are reserved for application developers.

AEP GUI messages (objects)

The AEP applications can interact with the users with custom LCD-views. This works by communicating messages within the system firmware.

AEP sends messages to the browser which interprets them depending on their types. AEP supports these message object types: `message`, `select`, `html`, `input` and `screen`.

When the browser receives them, they are rendered from scratch by default.

As `html` is a native browser type, and the browser knows Javascript, this allows injecting Javascript by using the `html` object type. Javascript can also be injected by using `html` in a `screen` object.

By design contract, AEP (lua) can set `js_*` to names of Javascript methods added to the object `window.AEP_js={};`

When certain browser events occur, those can trigger callbacks, which makes a call to the injected Javascript method for the specified object.

Example in a `screen.lua_prepare`:

```
 screen.js_rendered="Rendered1"
if not screen.Script then
  screen:add_script([=[
AEP_js.Rendered1=function(view,$el) {
  alert("screen was rendered");
};
]=]
  )
end
```

The browser will run this code after the screen has been rendered. NB! rendered occurs before the actual elements are showing on the LCD.

```
 let rendered_fn_name=this.field.js_rendered;
let rendered_fn=window.AEP_js[rendered_fn_name];
if(typeof(rendered_fn)==="function")
{
    try { rendered_fn(this,this.$el); } catch(e) { console.log(e); }
}
```

The GUI message objects supports these events:

- `js_rendered(message_object,view)` or `js_rendered(view,message_object,$el)` Use-case: A notification that rendering is done.
- `js_afterRender(message_object,model,view)` Use-case: Nothing
- `js_onsend(value,message_object,view)` Use-case: To make it easier to customize what is sent back to AEP, by modifying value.

There is also a cleanup function called, when the current AEP application stops. This allows removing custom code/DOM added.

- `AEP_jsCleanup(dummy)`

The implementation in the browser uses Backbonejs to make models+views, and JQuery. Thus `model` refers to a Backbone model, `view` refers to a Backbone view and `$el` refers to a JQuery object representing the DOM.

Developer tip! Using e.g. Chrome Developer Tools is very useful to explore and get details on the Javascript parameters.

From now on, this document will only describe the features of `screen`.

Screen special asset screenimages/AEP_js.js

An AEP-application can load its Javascript from a custom asset with the specified name. The format of the Javascript-file is a RequireJS module. RequireJs version 2.1.15. Backbonejs version 1.1.2, Underscorejs version 1.7.0. Jquery version 2.1.1. Handlebarsjs 2.0.0.

```
define([
    'backbone',
    'jquery',
    'underscore',
    'websocket',
    'keyboard',
    'utils',
    'handlebars',
    'models/models',
    'AEP'
], function (
    Backbone,   /* Backbonejs module */
    $,          /* JQuery module */
    _,          /* Underscorejs module */
    Websocket,  /* SATO qui message channel module */
    Keyboard,   /* SATO on-screen keyboard module */
    Utils,      /* SATO Utils module */
    Handlebars, /* Handlebarsjs module */
    Models,     /* SATO Backbonejs Models module */
    AEP         /* SATO AEP module */
) {
    return {
        // user specified application specific function
        Rendered1:function(view,$el) {
            alert("something was rendered");
        },
        // Another user specified application specific function
        SaveFun:function(value,message_object,view) {
            alert("onsave:"+value);
        },
        my_popup: function() {
            AEP.popup({
                addText: ["Hello","World!"]
            });
            return false;
        }
        // User-specified cleanup function
        AEP_jsCleanup(dummy) {
            console.log("cleanup when AEP-application is stopped");
        }
    }
}
```

These populate window.AEP_js, and could be used on screens, or screen fields by referring to their names as

```
 screen.Item1.js_rendered="Rendered1"
screen.js_rendered="Rendered1"
screen.js_onsave="SaveFun"
screen.Button1.js_onclick="my_popup"
```

Screen JSON

The layout of the screen is saved as JSON, and it's generated in the AEPW3. The JSON-file is read into a Lua table. Some of the attributes have meanings only in the SA application, where others have meaning only in the web browser. The css object should only be made up of native browser CSS attributes, or how it's understood by JQuery version 2.1.1. The browser used is a QtWebkit-based browser.

The screen object has properties for itself, and for screen fields, that too are rendered on the LCD.

Screen object

The overall structure is a set of attributes for the screen, with an array of fields that make up the individual elements on the screen. If attributes are omitted, their defaults are used.

```
{
  "auto_disable":null|true, /* when true elements are set to disable at submit. Omit color from field css to see grayed out effect */
  "name":"Name", /* The *mandatory* name is used to reference a screens data */
  "title":null|"title string", /* unless null, used as screen title instead of name */
  "id":reserved, /* id is reserved for the designer */
  "error":null|"error message", /* reserved for error mesage */
  "invalid":null|true, /* reserved by SA to set invalid state */
  "_*":reserved, /* namespace reserved by SA,
                    _action is a flag to determine if the screen is "done"
                                */
  "[a-z]*":reserved, /* namespace reserved by aep+AEPW3/SDK */
  "sdk_*": reserved, /* namespace sdk_* attributes are reserved by AEP Works */
  "js_*": reserved, /* namespace js_* attributes are reserved */
  "lua*":reserved, /* attributes starting with lua are reserved by SA */
  "done":null|false|true, /* reserved by SA to indicate completed */
  "once":null|false|true, /* reserved by SA for run once screens */
  "selectable":null|true|false, /* mainscreen only: true if a station/checkpoint in the flow */
  "start":null,  /* The mainscreen that is the starting screen is specified in workspace.json*/
  "type":"screen"|"mainscreen"|"screen_progress", /* (label) screen or mainscreen, progress screen */
  "lua_prepare": null, /* lua event on before first screen send */
  "lua_update":null, /* lua event on fields update */
  "lua_done":null, /* lua event on screen done */
  "lua_actions":null, /* lua event on screen actions */
  "lua_finisher":null, /* lua event on mainscreen finishing */
  "lua_onsend":null, /* lua event on screen before each gui:send */
  "lua_onpopup":null, /* lua event on screen popup */
  "lua_source":null, /* lua event on screen after source */
  "lua__ondata":null, /* lua event when screen receives data */
  "lua__transition":null, /* lua event on mainscreen SA screen object for transition */
  "lua__from_memory":null, /* lua event on mainscreen SA screen object for transition memory control */
  "lua__loopmemorize":null, /* lua event on mainscreen SA screen object for transition memory control */
  "height":null|number, /* the height in pixels of the screen, defaults to 776 */
  "css":null|{key:value,...}, /* the css object is used as is by the browser */
  "backgroundImage":null|"image.jpg", /* background image */
  "fields":[{},...], /* specifies the fields on the screen */
  "qtyscreen": null|false|true, /* mainscreen only: true to mark screen as quantity screen
                                   which means it will be shown instead of the standard quantity
                                   screen, and it can also hold a label preview image*/,
  "splash":null|true,             /* screen is splash, no script support */
  "normalscreen":null|true,       /* SDK specific */
  "ws_msgs":null|{screen:heymsg()}, /* preloaded ws_msgs  */
  "[A-Z]*": reserved, /* attributes starting with a capital letter are reserved for the
                         application developers */
  "incremental_v": null | 1, /* turn on incremental update of screen */
  "showKeyboard": null|"auto"|"always"|"never", /* virtual keyboard behavior. null is the same as auto */

  "lua_before": obsolete, /* Obsoleted, use lua_prepare */
  "lua_onchange":obsolete, /* Obsoleted, use lua_update */
  "lua_validate":obsolete, /* Obsoleted, use lua_done */
  "lua_inscreenactions": obsolete, /* Obsoleted, use lua_actions */
  "lua_mainscreenactions": obsolete, /* Obsoleted, use lua_finisher */

  "js_rendered":null, /* inherited from GUI object*/
  "js_afterRender":null, /* inherited from GUI object */
  "js_onsend":null, /* inherited from GUI object */
}
```

The internal variables `_self`, `_gen`, `ci`, `rid` are used to identify messages and screen contents.

When `screen.incremental_v=1` is set, it changes how screen objects are rendered. If `_self` is the same as the previous instance, and the count of `screen.fields` is the same, individual fields are updated instead of redrawn from scratch, based on comparing the `_gen` value in the field of the previous instance. However, `grid` and `list` are examined item by item rather than by `_gen`.

A `field` or many, on the screen can be updated from lua by means of `screen:hey(msg)` as well, and if `for` is not specified, it will only be applied if the internal variables match. Read more further down.

Fields array

The fields array is made up of field objects, which share many common attributes, described here below

Common field attributes

```
{
  "name":"Name" /* The *mandatory* name is used to reference the object */
  "id":reserved, /* id is reserved by the design */
  "error":null|"error message", /* reserved for error mesage */
  "invalid":null|true, /* reserved by SA to set invalid state */
  "_gen":number, /* for incremental_v */
  "_*":reserved, /* reserved by SA */
  "sdk_*": reserved, /* sdk_* properties are resverved by the design */
  "js_*": reserved, /* namespace js_* attributes are reserved */
  "lua*":reserved, /* attributes starting with lua are reserved by SA */
  /* *mandatory* specifies the (major) type of screen element
    button,grid,input,select provide input
  */
  "fieldtype":"button"|"grid"|"image"|"input"|"label"|"line"|"select"|"html",
  /* Specifies what actions are taken after the field changes */
  "action":null|"feed"|"screen"| /* 1-item from value actions, "print" and "format" are deprecated */
    { /* or an action object can be used to specify more advanced actions */
      "action":"feed"|"formatv2"|"printv2"|"script"|"screen",
      key : value,... /* action specific items */
    }
  ,
  /* the value attribute holds the current value, it is null,boolean,number or string */
  "value":value,
  /* the source from where value is initialized */
  "source":"copy"|"fix"|"list"|"table",
  "backgroundImage":null|"image.png",
  "hPos":number, /* horisontal pixel position */
  "vPos":number, /* vertical pixel position */
  /* the css-object contains all browser native properties set */
  "css":{"width":"X px","height":"Y px", key:value,...}
  "[A-Z]*": reserved, /* attributes starting with a capital letter are reserved for the
                         application developers */
  "js_rendered": null,
}
```

The attribute source

The attributes depending on the attribute source vary, depending on the source.

Source "copy"

The attributes for copy depends on what is copied as described below. Values can be copied from screens, formats (labels) and tables.

```
{
  ...
  "source":"copy",
  "field":"nameOfFieldToCopyFrom"| /* name of field when copying from current screen */
    { /* object when copying from different screen, format or table */
      "field":"nameOfItem", /* the item to copy */
      "screenName":null|"nameOfScreen", /* null unless copy from a screen */
      "formatName":null|"nameOfFormat", /* null unless copy from a format */
      "tableName":null|"nameOfTable", /* null unless copy from a table */
    }
}
/* copy from current screen, field input1 */
{
  "source":"copy", "field":"input1"
}
/* copy from Mainscreen2, field input1 */
{
  "source":"copy", "field":{"field":"input1","screenName":"Mainscreen2"}
}
/* copy from format Label1, field price */
{
  "source":"copy", "field":{"field":"price","formatName":"Label1"}
}
/* copy from table Table1, column product */
{
  "source":"copy", "field":{"field":"product","tableName":"Table1"}
}
```

Source "list"

The list source is not implemented all the way yet, but is listed as it is an alternative to source "table" if the options are known and short. It should not be used in current designs.

```
{
  ...
  "source":"list",
  "list":"v1,v2,v3", /* comma-separated list of values */
}
```

Source "table"

The table source is useful for fieldtype "grid" and "select", but also for "image", "input" or "label". When used on "grid" and "select" a table row is selected, all columns in the row are accessible.

For other fieldtypes it is initialized from the table column when the row selection changes.

```
{
  ...
  "source":"table",
  "column":"nameOfTableColumnToCopyFrom", /* from which column value is picked for fieldtype
                                           image,input,label */
  "tableName":null|"nameOfTable", /* null for last selected table or the name of the table */
  "row_id": null|number /* the last selected row */
  "displayColumn": null, /* reserved for grid | select */
  "searchColumn": null, /* reserved for grid | select */
  "imageColumn": null, /* reserved for grid | select */
  "valueColumn": null, /* reserved for grid | select */
  "sortColumn": null, /* reserved for grid | select */
  "lua_extends": null|object /* used in grid | select */
}
```

The attribute action

The action attribute specifies how the field value is used regarding flow. The action attribute/object can trigger two lua events: in_screen_actions or main_screen_actions.

- **"feed"** is classified as an "in screen action", meaning that if a button has action "feed", the feed will take place while the LCD stays in the same screen.
- ~~"format" is used to select a label format. If used on a button, the flow moves to processing the label format.~~ **Deprecated**
- **"formatv2"** is used to select a label format and specify a print quantity.
- ~~"print" is used to specify a print quantity. If used on a button, the flow moves to processing the label with the specified print quantity. If used in a mainscreen, it can select a format.~~ **Deprecated**
- **"printv2"** is used to specify a print quantity from a label screen.
- **"script"** can be used as an "in screen action", or "mainscreen action", and there are two different attributes in the action object.
- **"screen"** is used to control navigation among mainscreen.
- **"video"** is used to play a video

When the action object is present, it contains the action and the attributes for the action.

```
{ /* action object for more advanced actions */  "action":"feed"|"formatv2"|"printv2"|"screen"|"script"|"video",
 key : value,... /* action specific items */
}
```

The sourcedProperty

formatv2, **printv2**, **screen**, **video** supports the sourced property object. The sourced properties can pick up values from other screen fields or table sources.

Depending on the nature of the property, it usually instantiates as a number or string.

```
{
    ...
    "property":null|number|string|{   // use null to exclude from setting
        "value":null|number|string,   // value will be updated at runtime
        "source":null|"copy",         // if source is null, field is null
        "field":null|string|{
            "field":null|string,      // use null to copy from self
            "screenName":string|null, // if not defined, current screen will be used
            "tableName":string|null   // copy value from table (column as specified by "field")
        }
}
```

Action feed object

In case the feed action should feed a distance instead of a label, the object notation must be used.

```
{ /* feed action */
  "action":"feed",
  "value":null|number, /* the feed distance in millimeters (floating point) or null for label */
}
```

Action formatv2 object

The "formatv2" action is used to load a label format and can also specify a print quantity and return screen. It can only be used in a main screen.

```
{
  "action": {
    "action": "formatv2",
    "formatName": sourcedProperty(of string),
    "quantity": sourcedProperty(of number),
    "returnScreen": sourcedProperty
  }
}
```

Action printv2 object

The action "printv2" object can be used to set the print quantity and/or specify the return screen. It can only be used in label screens.

```
{ /* print action */
  "action": {
    "action": "printv2",
    "quantity": sourcedProperty(of number),
    "returnScreen":sourcedProperty
  }
}
```

Action screen object

The screen action comes in two styles.

```
{ /* screen action */
  "action":"screen",
  "value":"nameOfScreen"|"-1", /* the screen name to jump to or other flow reference */
}
```

or

```
{
    "action": {
        "action": "screen",
        "screen": sourcedProperty(of string),
        "quantity": sourcedProperty(of number),
        "returnScreen": sourcedProperty(of string|number)
    }
}
```

Action video object

The video action will start playing a video.

```
{
    "action": "video",
    "video": sourcedProperty(of string),
    "options": {
        "autoPlay":true|false,
        "closeOnDone":true|false,
        "landscape":true|false,
        "loop":true,false
    }
}
```

Action script object

The action script is used to run user provided scripts when a field specifies script as action. In a label screen, only `lua_isa` is executed every time the data in that field receives data.

In a main screen a field specifying `lua_isa` and `lua_msa`, both will be executed. The `lua_msa` is called from the `builtin_finisher` (see further ahead).

The prototype for `lua_isa` is `isa,fn=lua_isa(screen,sf,af,builtin_actions)`. The return values must have the types `boolean,function*`. The `lua_isa` is called from the `builtin_actions`, and the returned function `fn` is added to the table of functions to execute, when the boolean `isa` is true. It can be seen in the `lua_actions` (see further ahead).

If the `lua_isa` script returns `false`, the value and type of `fn` is ignored (5.1.0-~). When the `lua_isa` is set on a button field, and it returns `false`, the screen session is completed.

The `lua_isa` script action that returns `true,fn` does not cause the screen session to be completed, unless `af.submit` is set to `true`.

List of lua_isa builtin_actions

All the `lua_isa builtin_actions` can be used in a `lua_isa` script like this `return builtin_actions(screen,sf,new_af)`

action	Description of new_af
feed	feed action object
printv2	printv2 action object
tableeditor	"tableeditor" or {action="tableeditor",password="xxx"}
launch	{action="launch", launch_name="screenname", launch_options=options (see screen._launch), lua_prelaunch=function, lua_postlaunch=function, }
unlaunch	return "unlaunch","just end" to exit "screen" mode. Used with "launch"

The prototype for `lua_msa` is `mode,destination=lua_msa(screen,sf,af,builtin_actions)`. The purpose of this function is to be able to compute the new `mode` and `destination` for where the user should end up when the current main screen is finished.

mode	destination	Description
"screen"	-	The name or path for the main screen
"format"	-	The name of the label / format to jump into. The format must also be loaded into memory. This pattern can be used for that: `builtin_actions.format(screen,sf,{value=Name})`
"table"	-	Enter table mode (not used)
"unlauch"	-	Return from screen._launch
nil	nil	The user remains in the current main screen one more time

```
{ /* script action */
  "action":"script",
  "lua_isa":null|"luaFunction", /* the lua function to execute as in_screen_action */
  "lua_msa":null|"luaFunction" /* the lua function to execute as mainscreen action */
}
```

List of lua_msa builtin_actions

All the `lua_msa builtin_actions` can be used in a `lua_msa` script like this `return builtin_actions(screen,sf,new_af)` The lua table `new_af` is created by the `lua_msa`, and corresponds to that type of action object.

action	Description of new_af
screen	screen action object
format	{value=Name} , loads the format specified by Name
formatv2	formatv2 action object
video	video action object

The fieldtypes

Fieldtype "button"

The button fieldtype is rendered as an div element. The attribute "disabled" can be used to modify how it is rendered with a lua event script. For screens with multiple editable fields, buttons are used to complete the screen. `lua_done` must be used to prevent fieldtype `"button"` to complete the screen.

```
{
  ...
  "fieldtype":"button",
  "disabled":null|true|false, /* HTML attribute */,
  "icon":url|object /* absolute or relative to screenimages, data-url too,
    and an object can be used to use system icon-font, e.g.
    {content="M",css={"font-size":"24px"}}
  */
  "js_rendered":null, /* method(view,$el) */
  "js_onclick":null, /* approve=method(field) */
}
```

The use-case for `js_rendered`, is symmetry and it allows customizing the element.

The use-case for `js_onclick`, is to control what happens which the button is clicked. It could be used to open an AEP-popup, or by returning `false`, to ignore the click event.

Fieldtype "grid"

The grid fieldtype is rendered as a grid of push elements. They can be text-only, image-only. A grid can be searched. Practically it can only be used with a table.

```
{
  ...
  "fieldtype":"grid",
  "buttonHeight":number, /* common height in pixels for all grid buttons */
  "buttonWidth":number, /* common width in pixels for all grid buttons */
  /* these refer to the column names of the table source */
  "displayColumn":null|"nameOfTableColumn", /* Which column contains titles.
                                       If null, titles are not displayed. */
  "imageColumn":null|"nameOfTableColumn", /* Which column contains image names.
                                      If null, images are not displayed. */
  "keyboard":null| /* Select default keyboard. Only applicable when searchColumn is specified.*/
    false | /* Hide keyboard. */
    {"layout":"alphanumeric"|"numeric"|"hex"|"ipv6",   /* Select keyboard layout.  */
    "keyset":null|"alpha"|"alphaShift"|"numeric"|"special"},/* Select keyset in layout. Null selects the first keyset of the layout.*/
  "searchColumn":null|"nameOfTableColumn"|["searchCol1","searchCol2"], /* Which column(s) to search from.
                                       If null, search input is hidden. */
  "sortColumn":null|"nameOfColumn", /* which column to sort by */
  "valueColumn":"nameOfTableColumn", /* which column value to write in attribute value */,
  "bgcolorColumn":null|"nameOfColumn", /* column for background-color */
  "colorColumn":null|"nameOfColumn", /* column for text color (css:color) */
  "horizontalScroll":null|true, /* horizontal scrolling */

  "disabled":null|true|false, /* HTML attribute */
  "autoWrap":null|true|false, /* wrap long lines if true (+maxLines+hardLineBreak) */
  "maxLines":null|number, /* maximum number of lines used for long lines */
  "hardLineBreak":null|"string"|["string"...], /* which characters are hard line breaks */
  "_preloaded":object, /* initial contents, populated before send */
  "preload_block":null|true, /* block _preloaded if truthy */
  "required":null|true, /* require a value when true */
  "required_css":null |{css-to-decorate-a-required-grid}, /* the default required_css is a 2px solid red border */
  "js_rendered":null, /* method(view,$el) */
  "js_elems_added":null, /* method(view, $el) */
  "js_elems_adjust":null, /* if(method(view,$el,item,field)){add_elem($el);} */
  "js_mark":null, /* method(view,$el) */
  "js_unmark":null, /* method(view, $el) */
  "js_onselected":null, /* if(method(field,payload,item)) { send } */
}
```

The grid type supports a few js_events:

- js_rendered(view,$el) Use-case: symmetry
- js_elems_added(view,$el) Use-case: e.g. customize scrollbar after adding data to the grid
- js_elems_adjust(view,$el,item,field) Use-case: to conditionally remove items or customize css per cell

```
AEP_js.tweak=function(view,$el,item,field) {
  /*
  item[0]:id, item[1]:displayColumn(s),item[2]:imageColumn,
  item[3]:bgColorColumn,item[4]:colorColumn
  */
  if(item[1].length>20) {
      $el.css({"font-size":"10px"});
  }
  else if(item[1].length==0)
  {
     return false; // do not show any empty cells
  }
  return true; // keep this cell
}
```

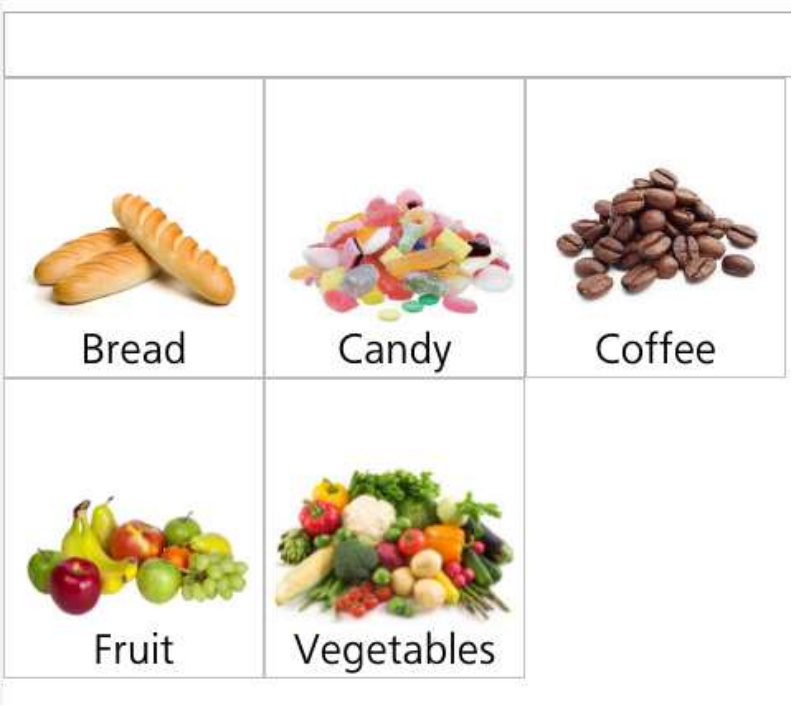
- js_mark(view, $el) Use-case: customize what the selected item looks like

```
AEP_js.mark=function(view,elem$) {
    elem$.each(function() {
        this.style.setProperty("background-color","red","important");
        this.style.setProperty("color","white", "important");
    });
};
```

- js_unmark(view, $el) Use-case: undo custom js_mark effects
- js_onselected(field,payload,item) Use-case: e.g. to customize the payload, or to block selecting certain elements (e.g. empty items)

```
AEP_js.blockEmpty=function(field,payload,item){
return item[1]!==""; // returns true for non-empty displayCol...
};
```

A rendered `grid with search,images and display`



Fieldtype "html"

The fieldtype "html" is rendered as HTML, and it is currently only supported through scripting, preferably as a one-shot addition in a `lua_prepare` script. It is mostly for symmetry.

```
{
  ...
  "fieldtype":"html",
  "css":{"top":y,"left":x}, /* mandatory to avoid issues */
  "name":mandatory, /* per screen unique name is mandatory for proper integration */
  "value": HTMLDomObjectAsText, /* necessary even if not displaying */
  "js_rendered":null, /* method(view,$el) */
}
```

Fieldtype "image"

The image fieldtype is rendered as an HTML `<img src=value>` element. The value for the HTML-src attribute is the value from the field. The image formats that are supported are jpg,png,bmp,svg. In the quantity/preview screen, `value` can be set to ":preview:" in order to make it a placeholder for the label preview image. Image also supports js_rendered, for symmetry.

```
{
  ...
  "fieldtype":"image",
  "value":"screenimages/image1.jpg"
  "js_rendered":null, /* method(view,$el) */
}
```

Fieldtype "input"

The input fieldtype is rendered as the equivalent of an HTML `<input type="text">`. The disabled, readonly and required attributes are mostly intended for lua event scripting. When the pattern attribute is described in Javascript regexp style, more powerful control is given compared to SA's implementation.

```
{
  ...
  "fieldtype":"input",
  /* AEP SA styled patterns: "%maxlength[.decimals]s|u|d|n|f"
    s: string, u: unsigned integer, d: signed integer, n:unsigned float, f: signed float
    E.g.
    %12s => up to 12 letters
    %3d  => up to 3 digits
    %2.2n => 0 .. 99.99
    %2.2f => -99.99 .. 99.99; decimal point can be omitted.
    If the pattern doesn't start with %, it's assumed to be a Javascript regular expression
    var regexp=new Regexp(pattern);
    */
  "pattern":null|"<SA-style>"|"Javascript regular expression"|{"re":"regexp","m":"modifier"},
  "keyboard":null |                                    /* Select default keyboard. */
    false |                                            /* Hide keyboard. */
    {"layout":"alphanumeric"|"numeric"|"hex"|"ipv6",   /* Select keyboard layout.  */
    "keyset":null|"alpha"|"alphaShift"|"numeric"|"special"},/* Select keyset in layout. Null selects the first keyset of the layout.*/
  "disabled":null|true|false, /* HTML attribute */
  "readonly":null|true|false, /* HTML attribute */
  "required":null|true|false, /* HTML attribute */
  "js_rendered":null, /* method(view, $el) */
  "js_onfocus":null, /* method(view, $el) */
  "js_confirminput":null, /* method(view,$el,confirmevent) */
}
```

If custom Javascript is used triggering a `focusout` element on the input element will make it sent to the backend. The input element also supports these js-events:

The grid type supports a few js_events:

- js_rendered(view,$el) Use-case: symmetry
- js_onfocus(view,$el) Use-case: e.g. to show/hide something to the user when input is focused.
- js_confirminput(view,$el,confirmevent) Use-case: to conditionally e.g. show/hide something when focus is lost ("lostfocus") or it is confirmed ("confirmed").

Fieldtype "label"

The label fieldtype is rendered as plain HTML text, and cannot be edited, and has no special attributes, except `js_rendered(view,$el)`, which exists for symmetry.

Fieldtype "line"

The line fieldtype is rendered as a static HTML element oriented horizontally or vertically, intended as a separating element, but recommended not to use. The line color can be specified using css background-color. For symmetry reasons, js_rendered is supported.

```
{
  ...
  "fieldtype":"line",
  "deltaH":number, /* number >0 unless vertical line */
  "deltaV":number, /* number >0 unless horizontal line */
  "thickness": number, /* thickness in pixels of line */
  "js_rendered":null, /* method(view,$el) */
}
```

Fieldtype "select"

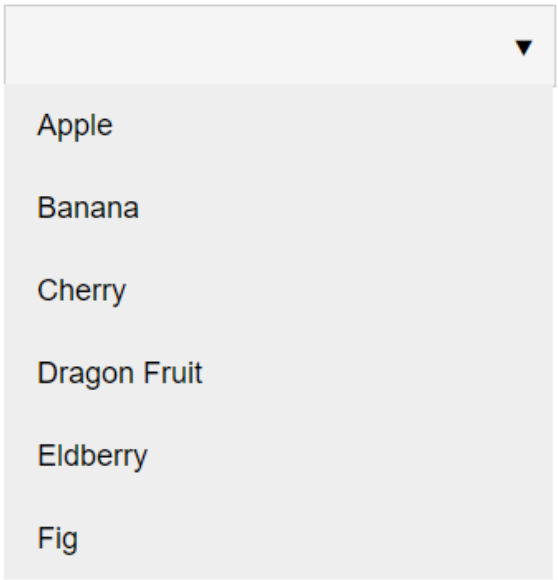
The select fieldtype is rendered with a combination of HTML `<input type="text">` if the `searchColumn` is given, and list, or visually comparable to an HTML `<select><option>...</option></select>` when type is `"dropdown"`. Practically it can only be used with a table as source. The default maximum number of lines for `"dropdown"` is currently 50 items. When type is set to "input", the element will visually look like a standard input element, but upon confirmation the value will be validated against the searchColumn in the source table. If an exact match is not found, the field will become invalid. If searchColumn is specified, it behaves like a combobox, i.e. shows an input field, and a small arrow that can be clicked to show the dropdown list. If something is entered in the input field, the dropdown list will be filtered against the search value.

```
{
  ...
  "fieldtype":"select",
  "type":"dropdown"|"list"|"input",
  /* these columns refer to the column names from source table */
  "displayColumn":"nameOfTableColumn"|["Col1","Col2",...], /* specifies which column(s) are shown */
  /* Keyboard is only applicable when searchColumn is specified for "dropdown" and "list" type. Always applicable for type "input". */
  "keyboard":null |                                        /* Select default keyboard. */
    false |                                                /* Hide keyboard. */
    {"layout":"alphanumeric"|"numeric"|"hex"|"ipv6",       /* Select keyboard layout.  */
    "keyset":null|"alpha"|"alphaShift"|"numeric"|"special"},/* Select keyset in layout. Null selects the first keyset of the layout.*/
  "searchColumn":null|"nameOfTableColumn"|["Col1","Col2",..], /* Which column(s) to search from.
                                      If null, search input is hidden. */
  "sortColumn":null|"nameOfColumn", /* which column to sort by */
  "valueColumn":"nameOfTableColumn", /* which column value to write in attribute value */
  "bgcolorColumn":null|"nameOfColumn", /* background-color column (list) */
  "colorColumn": null |"nameOfColumn", /* css color attribute (text color) */
  "disabled":null|true|false, /* HTML attribute */
  "required":null|true|false, /* HTML attribute */
  "required_css":null |{css-to-decorate-a-required-grid}, /* the default required_css is a 2px solid red border */
  "hardLineBreak":null|"string"|["string"...], /* which characters are hard line breaks */
  "_preloaded":object, /* initial contents, populated before send */
  "preload_block":null|true, /* block _preloaded if truthy */
  "js_rendered":null, /* method(view,$el) */
  "js_elems_added":null, /* method(view, $el) (type list only)*/
  "js_elems_adjust":null, /* if(method(view,$el,item,field)){add_elem($el);} (type list only)*/
  "js_mark":null, /* method(view,$el) */
  "js_unmark":null, /* method(view, $el) */
  "js_onselected":null, /* if(method(field,payload,item)) { send } */
}
```

The grid type supports a few js_events:

- js_rendered(view,$el) Use-case: For symmetry.
- js_elems_added(view,$el) [type: list ~~dropdown,combobox~~] Use-case: e.g. customize scrollbar after adding data to the list.
- js_elems_adjust(view,$el,item,field) [type: list ~~dropdown,combobox~~] Use-case: to conditionally remove items or customize css per cell. See grid example.
- js_mark(view, $el) Use-case: customize what the selected item looks like. See grid example.
- js_unmark(view, $el) Use-case: undo custom js_mark effects
- js_onselected(field,payload,item) Use-case: e.g. to customize the payload, or to block selecting certain elements (e.g. empty items). See grid example.

A rendered `select "dropdown"` in expanded view.



A rendered `select "list" with search`

This ends the JSON-definitions.

Showing popup dialogs from AEP

Starting from 5.2.0, AEP-applications, can utilize popup dialogs. They can be invoked by Javascript, or by screen:hey() messages.

```
screen:hey(
{
  .api={{"popup",options}}
}
)
```

It can be chosen if the popup should save its data to the lua-side, or be local to the browser. It is controlled with the options:

```
{
  -- system parameters
  text=nil|table|string, -- translated text, centered, one line per array
  addText=nil|table|string, -- as-is text, centered, one line per array item,
  contents=nil|html -- hmtl as-is, centered by default
  acceptCode=0, -- 0 is default
  acceptEnabled=true,
  acceptText='k', -- rendered as a checkmark with the icon font
  cancelCode=0, -- 0 is default
  cancelEnabled=true,
  cancelText='c', -- rendered as a x with the icon font
  dialogType='text', -- 'text'|'list'|'multiList','confirmList'
  listValues=nil | {}, -- {{keyCode,displayValue},...} if type is "list"|"multilist"|"confirmList"

  -- AEP specific
  cb=nil|js-function -- can be used with save=false
  save=true|false, -- default true
}
```

In order to create a popup with only a OK-button, use `cancelCode=-1`, to create a popup with a X-button, use `acceptCode=-1`. This is system specification.

When `options.save==nil` or `options.save==true`, when the popup dialog is closed for some reason, the frontend sends a message with the payload

```
{
  type="popup",
  state=xxx,
  ret=return_value, -- true|false
  value=value,
  list=list,
  options=options
}
```

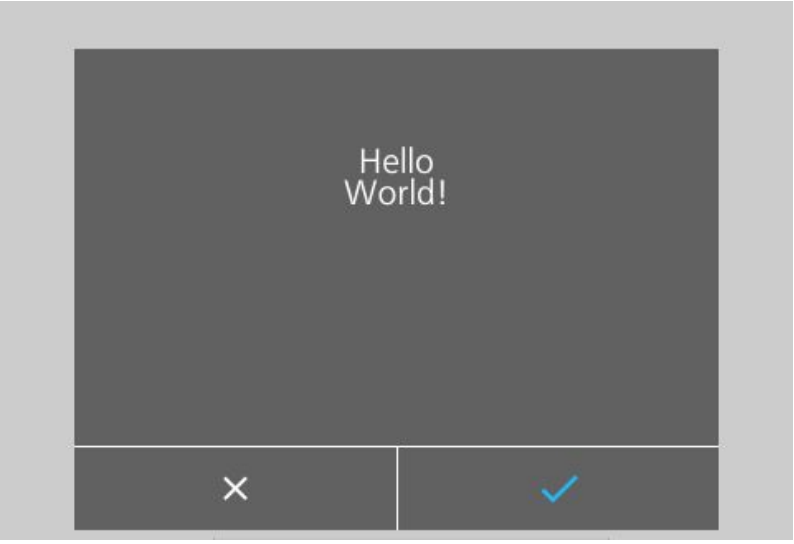
`.state` is `"completed"` when closed by the user. `.state` is `"aborted"` when closed by the system.

NB! The AEP popups will be `"aborted"` when they are shown, and the user opens the cover, or if the application sends a new popup-message to be shown, or if the printer system creates some kind of popup message.

The AEP popups are only shown in

- AEP-screen
- ONLINE-screen
- OFFLINE-screen
- HOME-screen

A simple sample of a popup:



bla bla

Accessing items from Lua

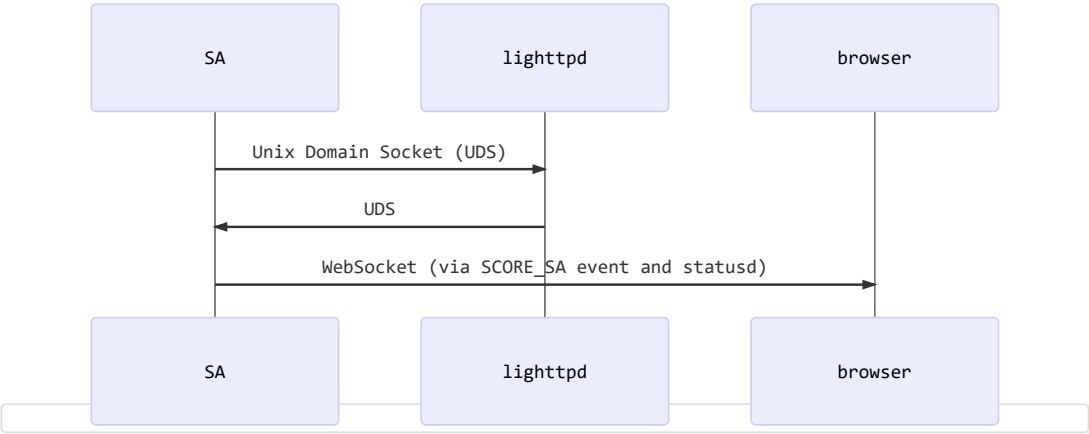
The screens support is an extension of the `SA` application developed for TH2, and much of the data access is handled the same way. As before the Lua variable `Format` references the current label format. The Lua variable `Row` referenced the last selected row from a table. The Lua variable `Screen` is added for referring to the current screen. As before `Format.Field.value` access a field value in the `Format`. Similarly `Screen.Field.value` access a field value in the `Screen`, and `Row.Column` access the value in `Row`.

In addition to this, `Formats.FormatName.Field.value` , `Screens.ScreenName.Field.value` and `Rows.TableName.Column` are available after their data has been resolved.

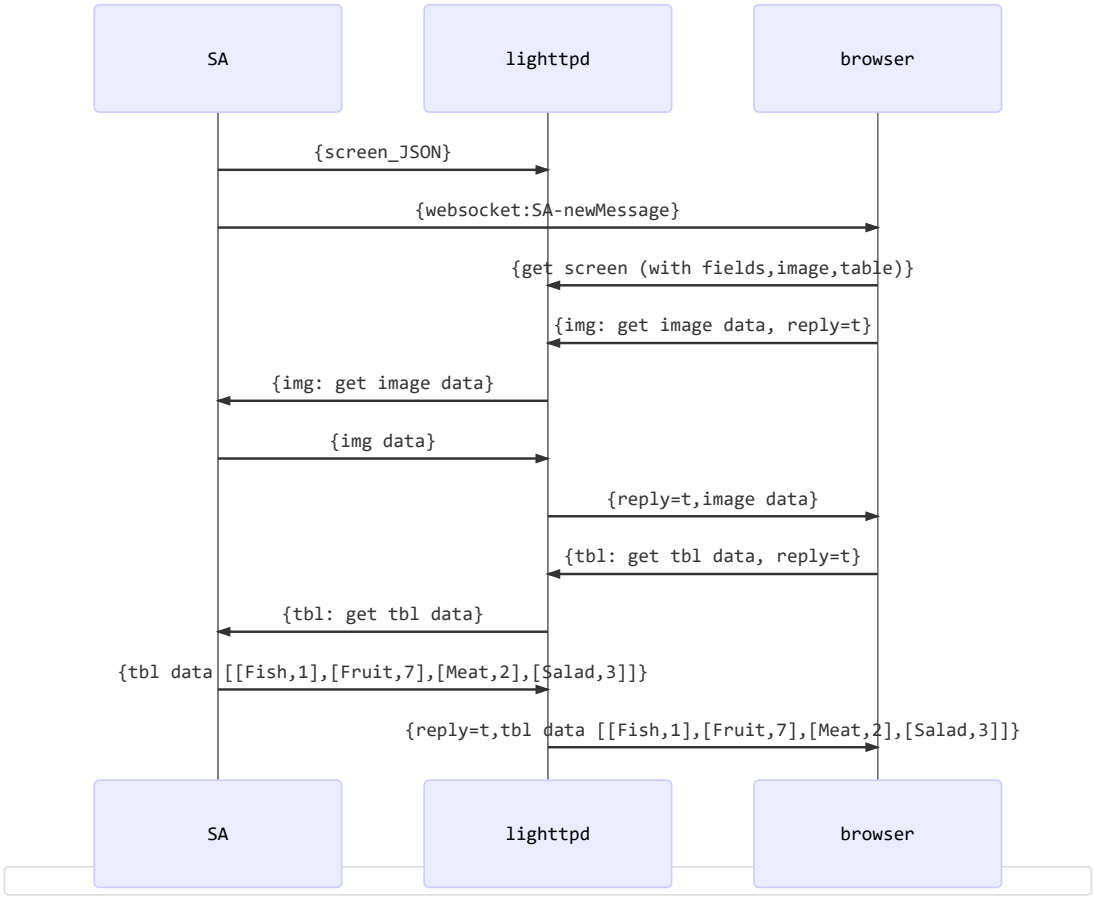
Screen Implementation data flow

The communication between `SA` and the browser is done with a mixture of Unix Domain Socket and WebSocket. lighttpd acts as the server of the UDS and `SA` as the client. `SA` is the Standard Application implemented with AEP to run the applications developed in AEPWorks.

The Lua API for this is implemented in `require("autoload.gui")`.



When SA wants to display the screen contents, it computes the value property for each field based on the fields source attribute. Then the Lua table is encoded into JSON, and then submitted via the `gui` communication model.

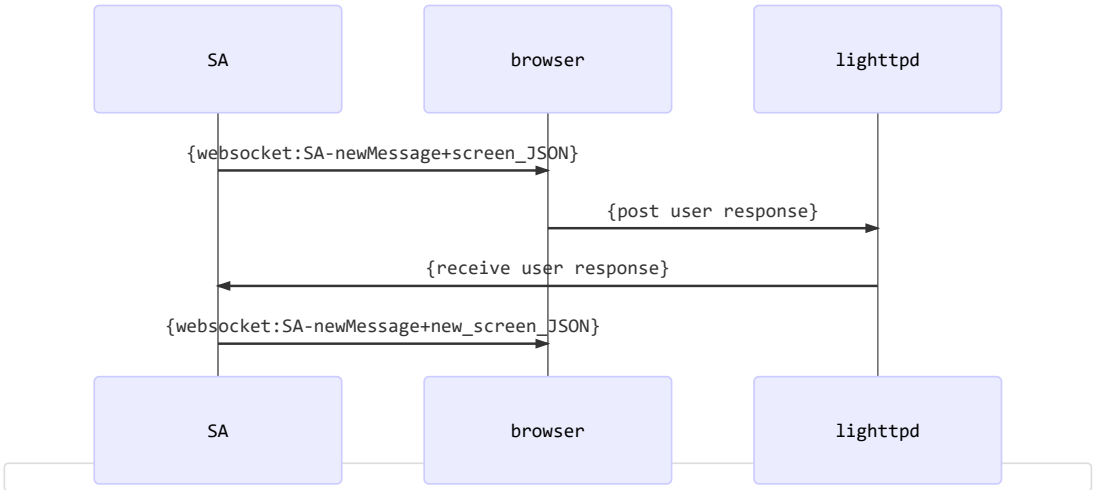


As the fields are parsed by the browser, new `XHR POST` requests are issued from the browser to fetch the items retrieved by `SA`. The `querystring` contains `reply=t` to indicate that `lighttpd` must wait for `SA` to complete the response. The `XHR POST` messages are used to fetch table data for `select` and `grid` elements.

`{img data}` is fetched as depicted through `SA` only for preview images. Other images are fetched from lighttpd.

If multiple browsers are connected, the `{screen_JSON}` is cached in `lighttpd`, but each browser, will issue its own `reply=t` requests.

Starting with 5.2.0, the communication is better described like this:



The websocket message contains the screen payload, and grid+select (list) fields are preloaded with their first contents, so the number of data turns back and forth are reduced.

A screen simplified screen data structure

```
{
  "name":"Screen1",
  "selectable":true|false
  "fields": [
    {
    "fieldtype":"list",
    "source":"table",
    "tableName":"Table1",
    "name":"List1",
     "backgroundImage":"screenimages/img1.png",
    "value":"Fruit"
    },
    ]
}
```

The Javascript code renders the screen data so that it matches the Screen editor. When the user modifies a screen field on the LCD, the new value is sent to `SA`:

```
{...,"fields":[{"List1":{"valid":true,"row_id":7,"value":"Fruit"}}]}
```

This will flow down to `SA` which will run various lua events when receiving the data. See the example further down on how to traverse the received data.

More Screen implementation details

The `SA` code handles label screens (`type:"screen"`) and main screens (`type:"mainscreen"`) slightly different. The flow between label screens is not controlled by the screen itself, but by how the label references the label screens.

The main screens adds navigation, which allows specifying where to go next. The navigations can be going to another main screen, label (possibly with label screens) or a table (not used).

Start showing a screen

This is a high level description of how `SA` implements displaying a screen. The `lua_*` names are describing the implementation, but variable names and other function names are just explanatory.

```
 function init_from_sources(screen,init)
    builtin_source(screen,init)
    lua_source(screen,init) end
end

function whileRunningScreen(screen)
    init={".*"}
    init_from_sources(screen,init)  -- optional lua_source event
    done=false
    isa_actions=nil
    while not done do
        init=nil
        lua_onsend(screen)
        data,err=gui:show(screen,isa_actions)
        lua__ondata(data,err)
        init=builtin_update(screen,data)
        newinit=lua_update(screen,data,init)
        if newinit ~= nil then init=newinit end
        init_from_sources(screen,init)
        isa_actions=builtin_actions(screen,data)
        newisa_actions=lua_actions(screen,data,isa_actions)
        if type(newisa_actions)=="table" then isa_actions=newisa_actions end
        done=builtin_validate(screen,data,isa_actions)
        newdone=lua_done(screen,data,done,isa_actions)
        if newdone~=nil then done=newdone end
    end
    run_any_remaining_isa_acions(isa_actions)
end

function startShowingScreen(screen)
    builtin_prepare(screen)
    lua_prepare(screen)
    whileShowingScreen(screen)
end
```

Main screens navigating

The main screens function looks roughly like this

```
function MainscreensBrowser(history)
    _spath={}
    mode="screen"
    path=""
    name=""
    last_so,so=nil,nil

    if history._save then              mode,name,path=lua__from_memory(dflt_from_memory,history,mode,name,path)
    end

    repeat
        so=Screens[name]
        lua__loop_memorize(dflt_looopmemorize,history,last_so,so,path)
        startShowingScreen(so.screen)
        nextmode,nextdst=builtin_finisher(so.screen,so.screen._submit)
        nextmode,nextdst=lua_finisher(so.screen,so.screen._submit,nextmode,nextdst)
        if nextmode and nextdst then
            mode,name,path=lua__transition(dflt_transition,history,so,path,nextmode,nextname)
        end
    until mode~="screen"
    reset_non_checkpoint_history(history)
end
```

The `MainscreensBrowser` function continues displaying the main screens until the `lua_finisher()` comes back with a new operation mode, e.g. `"format"`.

Modifying screen behavior with lua functions

The built-in behaviors per screen can be modified by specifying a function that is called when the "event" occurs. This is not necessary very often, but the opportunity exists.

The data from the browser

When the user edits field1, followed by a focus event to edit another field and so on, the data from such a editing sequence will be received in lua as below. The user edits Name and then FavoriteFruit.

```
{...,
    fields={
        {Name={valid=true,value="Tarzan"}},
        {FavoriteFruit={valid=true,row_id=7,value="Banana"}}
     }
}
```

Traversing the data received in lua

In the `fields` an array of tuples exists. As shown above, there can be multiple `fields` received in one event. This is how to traverse the fields in lua. Let `update_data` be the object received.

```
function onChange(screen,update_data,init)
  -- ex data {key="NOT_EN",fields={ {Field={value=X,..} },... } }
    for field,data in screen:pairs(update_data) do
        print(field.." updated:")
        print("data is valid",data.valid)
        print( (data.row_id and "does" or "does not") .. " reference a table row")
        print("the received value is:",data.value)
    end
end
```

The declared events

- `screen.lua_prepare` prototype `fn(screen)` ①

  Called after `builtin_prepare`, before the initial send of the screen JSON. Can be used to add/remove fields in the array `screen.fields`, reset data before showing screen again.

- `screen.lua_update` prototype `init=fn(screen,data,init)` ①

  Called after `builtin_update`. When `SA` receives information that a field is modified by the user, the screen fields value property is updated, and a table with fields that should be updated again is returned. The `init` parameter contains the `init` pattern produced by `builtin_update`. At `init[1]` a lua regexp that matches the fields to be updated from source can be set. The other pattern is `init["FieldName"]=1`, meaning that `FieldName` should be updated from it's source again.

- `screen.lua_done` prototype `done=fn(screen,data,done,isa_actions)` ①

  Called after `builtin_validate` which checks that all the fields on the screen have valid values, before advancing to the next screen. When `true` is returned, the screen is done. When `false` is returned, the screen is not done. The argument `done` is the result from the `builtin_validate`. The argument `isa_actions` is the return value from the `actions`. NB! The `builtin_validate` will return `false` as long as `isa_actions` is not an empty table.

- `screen.lua_actions` prototype `isas=fn(screen,data,isa_actions)` ①

  This is called when a screen field with an action receives data, and it is handled while the screen is showing. The return value is a lua table with functions to run. They are called after the new screen is sent, but before a new response/request is handled by `SA`. A function that is slow to complete, is not a good function to run this way. The `isa_actions` is the default functions to run. If a `lua_isa` returns a function to run, it is inside `isa_actions`. Thus `table.remove/insert` of functions in `isa_actions` is possible. See note in `lua_done`.

- `screen.lua_finisher` prototype `nextmode,nextdst=fn(screen,submitfield,nextmode,nextdst)` ① The function is called when a main screen stops showing, but before the next destination is finalized. It is only available to `mainscreens`. This allows writing a script to control the flow beyond what is done the designer.

- `screen.lua_onpopup` prototype `emulate_update=fn(popup_message)` ① This function allows using popup-messages to receive data interaction from a system popup dialog. A `popup_message` can be converted to a normal update message, by returning `true`. This callback has higher priority than `emulate_update=cbOnPopup(popup_message)`, which is a catch all, to catch any AEP-popup response.

- `screen.lua_onsend` prototype `fn(screen)` ①

  The function is called before sending the screen data to the browser.

- `screen.lua_source` prototype `fn(screen,init)` ①

  The function is called after the screen has been updated from it's screen sources.

- `screen.lua__ondata` prototype `data,err=fn(data,err)` ②

  This internal function is called with the decoded data.

- `screen.lua__transition` prototype `mode,name,path=fn(dflt_transition,_spath,so,path,nextmode,nextdst)` ③

  This internal function is called, only from `mainscreens`, after the `lua_finisher()` returns. The `dflt_transition` parameter is the `builtin_transition` function, and it should be called with all the remaining arguments. If the call is omitted, `lua__transistion` is responsible for the full implementation.

- `screen.lua__from_memory` prototype `mode,name,path=fn(dflt_from_memory,history,mode,name,path)` ③

  This internal function is called, only from `mainscreens`, at the entry to restore the screen object. The `dflt_from_memory` parameter is the `builtin_from_memory` function, and it should be called with all the remaining arguments. If the call is omitted, `lua__from_memory` is responsible for the full implementation.

- `screen.lua__loopmemorize` prototype `path=fn(dflt_loopmemorize,history,last_so,so,path)` ③

  This internal function is called, only from `mainscreens`, to record the screen transistions. The `dflt_loopmemorize` parameter is the `builtin_loopmemorize` function, and it should be called with all the remaining arguments. If the call is omitted, `lua__loopmemorize` is responsible for the full implementation.

It is possible to add the event handlers to the screen object by implementing the callback function `cbScreenLoad(screen)` as below:

```
function cbScreenLoad(screen)
  screen.lua_update=onChange
end
```

By using the Generic AEP popup callback, the messages generated from AEP popups can be managed. If the function returns `true`, the contents of `msg` will be processed as a standard message.

```
function cbOnPopup(msg)
    if msg.state=="aborted" then
        return -- the dialog was aborted by the system
    end

    -- i.e. msg.state=="completed"
    if msg.ret==true then
        -- the user closed it with checkmark, let's emulate a click on Button1
        msg.fields = {{Button1={valid=true,value="1"}}
        return true
    end
    return
end
```

① The screen.fields members are cross-referenced by Name in screen.Name. The return value(s) can be omitted. ② Avoid using this function. The screen.fields are not cross-referenced. The return values are mandatory. ③ Avoid using this function.

Screen completion

Label screens and main screens are considered completed slightly differently. They have in common that the fields must have valid data.

Rules for completing a screen

The following table describes how `builtin_validate` works

Update Data	Completes	Description
invalid	×	*If any field has invalid data, it cannot complete*
required	×	*If any field is required and the value is "", it cannot complete*
editable	×	*fielditem of type select (Combo Box)|grid|input*
button	○	**Any button normally completes the screen**, except for when the button belongs to an `lua_isa` where `af.submit` is not `true`
1 editable item	○	If only one editable field is on the screen it completes
>1 editable items	○	When an update data field has an action specified. When the action is script, it must set `.submit=true`
last editable item	○	**For label screen only** A label screen doesn't need an action to complete.

× - no, does not complete ○ - yes, does complete

Main screen specifics

A main screen starts and is finishes according to the table above, even when there is no new destination to go to. Using scripts this is observable by noticing the `lua_prepare` event being fired.

Scripts using lua_extends

The `lua_extends` object transported as is from the application, inside the screen object to the browser, and from the browser to the application in the `tbl` operations. As of now, all fieldtype `grid` and `select`, will query the application for the data every time. This is used in the SDK to create table filters, but it can be also be used to modify certain aspects of the `tbl` responses.

The `tbl` response contains a table `t` used for input/output arguments. These are the described fields in the payload:

```
{
    tableName=nameOfTable,     -- mandatory
    firstword=nil or true,     -- affects indexing
    index=nil or indexColumn, -- usually from searchColumn
    sortBy=nil or sortColumn, -- usually from sortColumn
    search=nil or string,      -- search key
    where=nil or table,        -- where-clause for sdb.tqwuery
    columns={column names},    -- columns to return, affects where-clause
    offset=nil or number,      -- sdb.twqery offset parameter
    offsetById=nil or id,      -- sdb.twquery offset parameter
    rows=number,               -- maximum number of rows to return
    colmap={columnName=i,...} -- the Javascript array index of where a column is found
    result=queryResult or {}, -- the output result
    error=nil or string,       -- in case of sdb.twquery errors this is set
    lua_extends=nil or table, -- the lua_extends argument
    _req=true,                 -- internal flag
    _img=nil or imgColumn,     -- if a column refers to image names
}
```

There are two hooks available for `tbl` responses

- `pre` - run before the `sdb` query operations begins.
- `post` - run after the `sdb` query operations finishes.

There are five built-in functions:

- `append(t,args)` - append the table data in `args` into `t`.
- `delete(t,args)` - delete the table data keys in `args` from `t`.
- `set(t,args)` - set the table data in `args` in `t`.
- `filter(t,args)` - used by the SDK to create filtering. Modifies `t.where`.
- `distinct(t,args)` - used by the SDK to create distinct filtering. Modifies a member in `t`.

A high level description of the query function

```
function queryFunction(t)
   connectToSdbAndMakeIndex(t.tableName,t.index or t.sortBy or "id")
   t.where=t.where or {}
   t.where.columns=t.columns
   apply(t,"pre")
   local rows=sdb.twquery(
                       index,
                       t.search or '',
                       t.rows,
                       t.offsetById and {id=t.offsetById} or t.offset or 0,
                       t.where
                       )
   sort(rows,t.index,t.sortBy)
   t.result=rows or {}
   t.colmap=mapcolumns(t) -- map lua[1]==Javascript[0] => Javascript
   apply(t,"post")
   return rows and #rows or 0
end
```

The easiest way is to show by an example. The browser fetches rows in batches of 30 rows. This example demonstrates using the built-in `set` function.

```
local function getMoreRows(f)
  local le=f.lua_extends or {}
  le.pre=le.pre or {}
  table.insert(le.pre,{fn="set",args={rows=100}})
  f.lua_extends=le
end
getMoreRows(screen.Select1)
```

Workspace functions can also be used, e.g. specify `fn="MyFunction"` for your own `MyFunction`.

Utility functions

To make it easier to script screens, the following functions are available from 5.1.0 in the `"lua_event"` functions and on `Screen`.

- `screen:add(sf)` Adds the screen field sf to screen.fields once, and cross-references it. `sf.name` is necessary. `sf.css=sf.css or {left=0,top=0}`. `sf.fieldtype` must be set for proper operation.
- `screen:add_html(sf)` Adds the screen field sf to screen.fields once, and cross-references it. `sf.name` is necessary. `sf.fieldtype="html"` is set if omitted. If sf.value is a lua file, it will be read, and closed.
- `screen:add_script(sf[,pos])`

  Adds the screen field sf to screen.fields once, and cross-references it. `sf.name` is necessary. `sf.fieldtype="html"` is set if omitted. If sf.value is a lua file, it will be read, and closed. The function adds `<div/><script>/*<![CDATA[*/`, the value (or reads if from file pointer), and `/*]]*/</script>` for better interoperability. If `pos` is given (>=1), the field is inserted at that position in screen.fields.
- `screen:apply(data,fn)` Applies the call `fn(field,fielddata)` to all received fields in data.
- `screen:get_lua_fn(name)` Returns the lua function instance specified by name, e.g. `lua_prepare`.
- `screen:hey(msg[,uds])` This function can be used to update screen data from the backend asynchronously for single item types (input,label,textarea,button). The msg specifies what field and part to update: `value`, `text`, `html`, `trigger`. E.g. `{set={{name="Input1",value="15"},{name="Input1",trigger="focus"}}}`. The member `['for']` is automatically added, to direct it only to the same screen instance. This can be a bit tricky, so, by adding `['for']={}`, it will target any screen instance.
- `screen:heymsg(msg)` This function returns the generated message from msg, for tweaking. The generated message is contains the field `['for']` which targets a specific instance. It can be applied broader if less items are set in `['for']`. The default value is

  ```
  {
    _self=self._self, -- _self==tostring(screen)
    _ci=self.ci,
    _rid=self.rid,
    _gen=self._gen
  }
  ```

  It can also be used in `lua_prepare` to set `screen.ws_msgs=screen:heymsg(msg)`, to set focus on a specific field.
- `screen:luae_factory(sf,options)` This function can be used to create a lua table driven grid/select data source in a favor of a sdb driven data source.
- `screen:pairs(data)` Returns an iterator for data.fields. Synposis: `for field,data in screen:pairs(data) do print(field,json.encode(data)) end`
- `screen:session_refresh()` API to complete the current screen session, and [possibly] restart it.
-

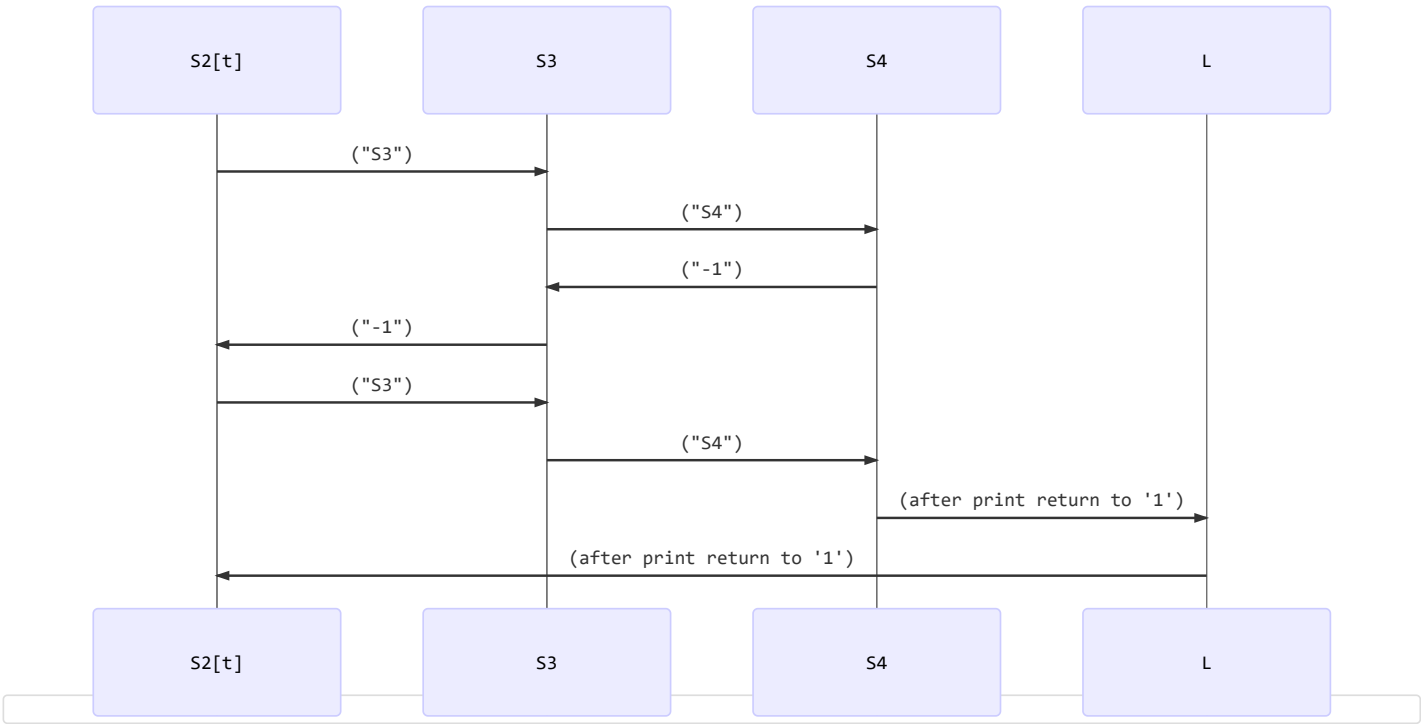
Some other functions are also available as plain calls.

- `screen._runbyname(name)` Runs the screen with the name `name`. If action `screen` is used, it can navigate to another screen, but when action `back` is used, the function returns. There's no memory kept for remembering the navigation history.
- `screen._launch(name,options)` Runs the screen with name `name`. In `options`, `memory`, `_first`, `_until` and `_return` can optionally be made as `table`, `function`s to tweak the behavior. This works a bit like `screen._runbyname()`, but it is a little bit more sticky. To finish, the start screen must also be navigated out of as long as the mode is `"screen"`.

Main screen navigation

Mainscreens are grouped by the property `selectable=true|false`. When `selectable` it can be returned to. This is similar to the `selectable` property of `label/format|table`. In the Screen editor, the selectable property is referred to as `checkpoint`.

Mainscreens with `selectable=false` can be navigated back to as long as the the user navigates within the group. In the diagram below S* are mainscreens and L is label. `S2[t]` is a `selectable` screen. The other screens or the label are not `selectable`.



When moving forward the name to move to is used, but the following patterns can also be used to specify the destination:

```
 value='-1' // Go back to previous screen in the same group or up to parent group
value='1' // Go to the first screen in the same group
value='/1' // Go to the Start screen
value='../1' // Go back to the first screen in the group above (relative)
value='../S5/S2' // Go up one group and then go to S5, but if visited already, go to S2
```

If jumping to `/S3`, it will not create a new group, but the group will be based on the parent group, e.g. `/`.