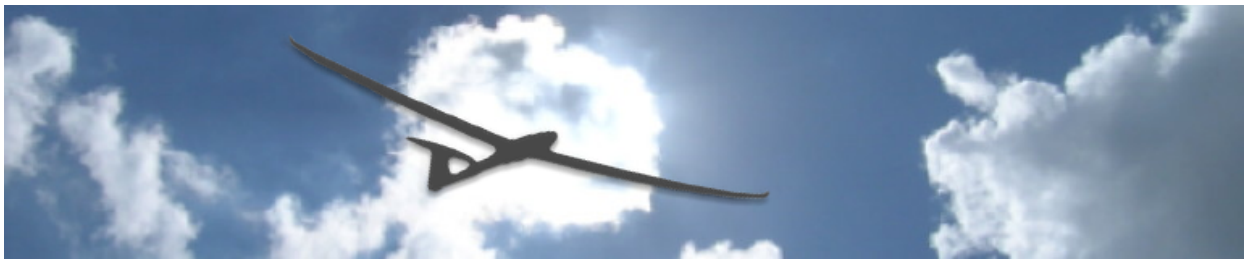


iMSB Lua Widget Scripting

Reference Guide

Version 1.0

This document describes the Lua user script programming interface related to script sensors. Script sensors are specific sensor values whose behavior can and must be programmed through Lua scripts.

[This page is intentionally left blank]

Introduction

With the iMSB Lua Widget scripting libraries user specific widgets can be programmed and attached to an iMSB display view. Widgets are essentially a graphic representation of a display element, such as a meter, used to display sensor values or any other graphic representation. A Lua Widget is attached to a display view the same way as any other pre-defined meter.

Lua Widgets are programmed using the lua scripting language. Lua is a powerful, efficient and embeddable multi-paradigm programming language designed as a scripting language. Detailed description and reference information is available at www.lua.org.

The Lua Widget library implements a subset of the iOS UIKit objects and functions. Basic display elements are taken from UIView, UILabel and UIImage. More advanced and efficient display elements taken from CAShapeLayer and CGPath are available. Most of these objects can be animated, in order to develop complex and rich display elements. Programmers having a knowledge of the Apple's UIKit will immediately feel at home. It is recommended that the user reads through the apple documentation before starting using the library. A good starting point is the "*View programming guide for iOS*" and "*Core animation programming guide*".

Lua Widget scripts can be used together with the Lua Sensor scripting library, to access sensor values and other attributes that need to be display in a Widget. It is important to note that the widget library solely handles graphic representations and animations. It does not provide any means to access sensor data. Useful sensor and value data need to be accessed via the sensor library.

In order to use Lua scripts the "*Expert mode*" must be enabled in the "*Data logging*" section of the application preferences.

Even if Lua is a quite efficient programming language, the use of widget scripts should be used moderately as it puts an additional strain on processing power and thus on battery drain of your device. Before using Lua you should insure that your required display features cannot be performed by the existing iMSB capabilities.

Lua language

Lua 5.3 has been integrated into iMSB and supports most of its language features. Before you start you should be familiar with the language concepts and syntax. Good references and support for Lua can be found on the official web site www.lua.org.

From the standard Lua libraries, the following have been integrated into iMSB:

Lua library	Included in iMSB
`Math`	Yes
`Table`	Yes
`String`	Yes
`I/O`	No
`Operating system`	`os.clock()`, `os.date()`, `os.difftime()`, `os.setlocale()`, `os.time()` only
`Debug`	No
`Package`	No

`Coroutine`	No

In addition, the following libraries form Lua Sensor scripting framework can be used:

- `SensorValue` class library: represents a sensor value object and exposes a set of methods to manipulate that value.
- `imsb` library: covers a set of function for handling announcements and notifications, as well as controlling application behavior.
- `device` library: provides access to the iOS device location and motion services in order to retrieve for instance the actual GPS position, altitude, as well as its orientation.

Please refer to the *"iMSB Lua Sensor Scripting Reference"* document for further details.

The Lua Widget Scritping Library provides the following classes as an extension to the language:

- `Widget` class: The class represents the canvas of the widget that is allocated in the display view. All drawing is performed in the widget and constrained to its bounds.
- `Box` class: `Box` are the fundamental building blocks of your `Widget`, and the `Box` class defines the behaviors that are common to all its subclasses. The `Box` class represents a rectangular zone that can be sized, moved, have a background and border color. It renders content within its bounds rectangle. In addition some interaction with the object can be performed. `Box` can nest inside other boxes to create view hierarchies, which offer a convenient way to organize related content.
- `Label` class: A `Label` is a display object that displays one of informational text. You can configure the overall appearance of a label's text, such as the font, its color and text alignment. In addition some predefined animation functions are provided for you. The `Label` class inherits from the `Box` class. Thus all functions defined for the `Box` class also apply to a `Label` object.
- `Image` class: The `Image` let you efficiently draw any image that can be specified using an image file stored on the iCloud or on the device. An image content can be changed dynamically and at any time during the script execution. In the current implementation only PNG images can be displayed. The `Image` class inherits from the `Box` class. Thus all functions defined for the `Box` class also apply to a `Label` object.
- `Layer` class: `Layer` objects are 2D surfaces organized in a 3D space and are at the heart of everything you do when performing complex animations or drawing. Like `Box` (and its subclasses), layers manage information about the geometry, content, and visual attributes of their surfaces. When performing animations you should always consider doing so with a `Layer` object instead of a `Box`, as system resources are handled much more efficiently.A `Layer` allows to draw complex drawing shapes with cubic Bezier splines. This is performed by defining a so called path that will be rendered at runtime. A path specifies a set of graphic primitives, such as lines, arcs, ellipses, etc. you use to construct your drawing. Each of these drawing primitives creates a subpath within a path. Attributes of you drawing such as line color, style, background colors, etc. may as well be specified. Constructing a path must be done within an `:beginPath` and `:applyPath` call in your code in order for the system to know which primitives belong to the path you wish to draw. Unlike other objects a path does not have an implicit animation. The path within a `Layer` may however be as well animated using a concrete animation function.

Layers can be used as a masking layer to mask or blend the content in its parent the layer hierarchy. A masking `Layer` has same drawing and layout properties of any other layer. A mask `Layer` is located relative to its parent. It is used in a similar way to a sublayer, but it does not appear as a normal sublayer. Instead of being drawn inside the parent, the mask `Layer`

defines the part of the parent layer that is visible. The `Layer`'s alpha channel determines how much of the layer's content and background shows through. Fully or partially opaque pixels.

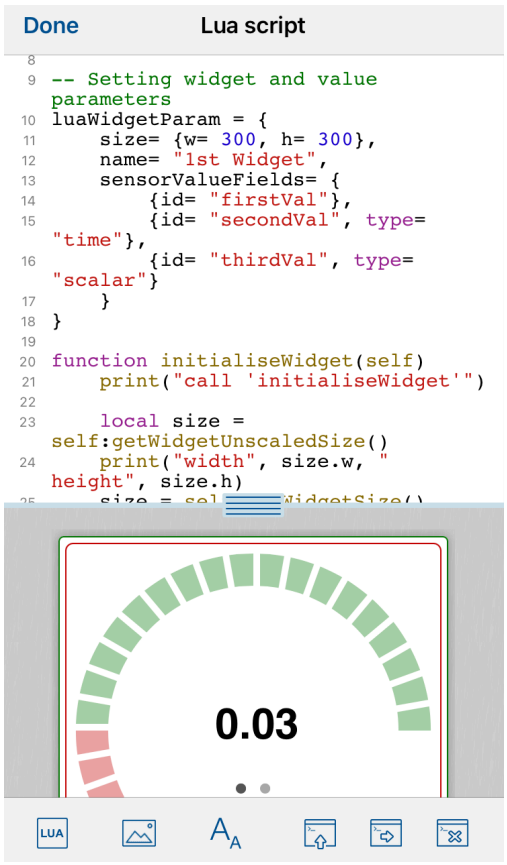
Lua scripts are directly stored in each display layout element. It is however possible to import or export a script file from iCloud. The editing of a Lua script is performed in iMSB with the editor that is available in the respective display view element. Particular debugging means are not provided, except for the console print function. A script console can however be used for displaying error messages and user log information. The console view is available as follows:

- In the lower pane of the script editor. This view allows to load and test the currently edited script.
- As a console display view accessible through the view tab bar that allows to display messages during a logging session. This view can be enabled or disabled in the "*Expert mode*" of the application settings.

Widget scripts

Setting up a widget script

Lua widget script are basic display elements that need to be added to a display layout. Therefore go into the layout editor and select the Lua widget display objets.



You may resize and position the element as usual.

In order to access the script editor simply tap the Lua Widget in the display layout view.

The upper pane of the view displays the editor and the Lua code. The lower pane a graphic representation of the widget once the script is loaded and running. Swiping the lower pane will additionally display a log view.

The lower toolbar allows to load or save a script, change the font size, as well as load a script, execute and terminate it.

Script anatomy

A Widget script must always contain at least the following two functions that are called and executed automatically by iMSB:

```
function initialiseWidget(self)
```

This function is called only once at the start of a real-time or playback logging session. It is typically used to perform all necessary initialisations required prior to start displaying data.

```
function updateWidget(self)
```

This function gets called on each display update cycle, at the same interval as defined in the "*Telemetry data update*" rate you can select in the application settings. This is where display and data updates will typically to be performed.

It is important that these two functions are defined, with the exact syntax as depicted above, within the script in order to insure proper operation and avoid errors.

The provided attribute self holds the instance of the Widget object (see below for further details).

Widget parameters

Within the widget scrip a global parameter variable may be defined to export general setting data to the application. Typically the name of the widget and the references to sensor values can be exported to the display layout configuration screen. Settings parameters are defined via a global table variable that must be named `luaWidgetParam` and has the following definition:

```
luaWidgetParam = {
    name= <name>,
    size= {w= ,h= },
    sensorValueFields= {
        {id= <id> [, type= <type>]},
        {id= <id> [, type= <type>]},
         etc…
    }
}
```

Where:

- `name`: (string) is the name given to the widget, that will be displayed in the display layout configuration.
- `size`: (table) is the normalised size (width and height) given to the widget. When defining a widget you always need to provide a default, i.e. called normalised size. This represents the bounds and coordinates in which the programmer will place its display elements. Note that the actual displayed widget may have any size chosen by the user. The iMSB application will in these cases take care to apply the required scaling. As a programmer you only need to care about the normalised size.

- `sensorValueFields`: (table, optional) are used to expose sensor value fields to the display settings. For each field the user may specify a sensor value. Sensor values allocated to each field can the retrieved via the `self:sensorValueForField(id)` function , where `self` is the instance of the widget and `id` the reference to the value field.
  Each value field is defined as a table of:
  - `id`: (string) the name given to the value field, as it will appear in the display settings.
  - `type`: (string, optional) is the type of the sensor value allowed. Possible values are:
    - `"any"`
    - `"scalar"`
    - `"time"`
    - `"string"`
    - `"coordinate"`
    - `"date"`
    - `"boolean"`

Note that in order for widget parameters to be populated to the app, the script must at least be run once.

Dark mode support

The iMSB app supports the iOS dark mode. Generally speaking it is the programmer's responsibility to adapt the colors used in your widget to correctly display in dark mode. You may identify if dark mode is enabled or not by the means of the widget's `:isDarkMode` method.

The Widget library provides however a standard predefined set of colors available via the `:getColor` method. These colors already have their dark mode counter part. Thus no particular measures need to be taken.

iMSB Lua Widget function reference

The Widget class

The Box class

The Label class

The Image class

:setAlarmImage(imageName [{r=, g=, b=, a=}])

The Layer class

Basic Layer functions

Layer:new(parent, {x=, y=, w=, h=})
Layer:newMask(parent)
:delete()
:reorder(index)
:setFrame({x=, y=, w=, h=})
:frame()
:setBounds({x=, y=, w=, h=})
:bounds()
:setPosition({x=, y=})
:position()
:setAnchor({x=, y=)
:anchor()
:hide(hidden)
:setBackgroundColor({r=, g=, b=, a=})
:setOpacity(opacity)
:setBorderWidth(lineWidth)
:setCornerRadius(cornerRadius)
:setBorderColor({r=, g=, b=, a=})
:setShadow({r=, g=, b=, a=}, opacity, radius [,{w=, h=})

Layer transformation and animation functions

:rotate(angle)
:move(dx, dy)
:scale(sx, sy)
:doAnimation(func, animationDur [, completionFct])

Defining Layer path

:beginPath()
:applyPath([invert])
:applyShadowPath()
:moveToPoint({x=, y=})
:lineToPoint({x=, y=} [,rotation, [{x=, y=}]])
:rectangle({x=, y=, w=, h=} [, roundn, [,rotation, [{x=, y=}]]])
:ellipseInRect({x=, y=, w=, h=}[,rotation, [{x=, y=}]])
:arc({x=, y=}, r, startAngle, endAngle, clockwise)
:arcToPoint(p1{x=, y=}, p2{x=, y=}, r)
:curveToPoint(cp1{x=, y=}, cp2{x=, y=}, {x=, y=})
:closePath()

Setting path attributes

:setStrokeColor({r=, g=, b=, a=})

:setLineWidth(width)
:setStrokeStart(start)
:setStrokeEnd(end)
:setFillColor({x=, y=)
:setLineDashPattern({ })
:setLineJoin(joinType)

The Widget class

The `Widget` class represents the canvas of the widget that is allocated in the display view. All drawing is performed in the widget and constrained to its bounds.

You do not explicitly create a widget object as this is done automatically for you when the script is loaded. The reference to the widget object is provided by the self variable provided in the `initialiseWidget` and `updateWidget` script functions.

:getWidgetSize()

Gets the normalised size in points of the widget as specified in the `luaWidgetParam` global variable. If the widget size was not specified a default (300, 300) size is returned.

Parameters:

- `n/a`

Returns:

- `{w=, h=}`: (table) the normalised size of the widget

Discussion:

Programmers should always consider the normalised size when manipulating a widget and its components. The scaling of the display elements to the actual widget as defined on the layout is automatically performed for you.

Usage:

```
local size
size = self:getWidgetSize()
```

:getWidgetUnscaledSize()

Gets the actual size in points of the widget as allocated on the display view.

Parameters:

- `val`: (number, string) any numerical or string value
- `{w=, h=}`: (table) the actua size of the widget

Returns:

- `n/a`

Discussion:

This method is provided for convenience if the actual widget size is required. As a programmer you should never use the actual widget size for setting up display elements, as the scaling would not be handled correctly anymore.

Usage:

```
local size
size = self:getWidgetUnscaledSize()
```

:sensorValueForField(fieldId)

Returns an instance of a `SensorValue` identified by its field `id` specified in the `sensorValueFields` section of the global `luaWidgetParam` variable.

Parameters:

- `fieldId`: (string) the name of the field used to reference a sensor value as defined in the `sensorValueFields` section of the global `luaWidgetParam` variable

Returns:

- `instance`: (userdata) the instance of the iMSB sensor value of class `SensorValue`
- `null` if the value does not exist or has not been set by the user

Discussion:

If the `sensorValueFields fieldId` references an existing iMSB sensor value you can use this method to directly retrieve an instance of the value of class `SensorValue`. You can then use all methods to access the required parameters of this value.

Please refer to the *"iMSB Lua Sensor Scripting Reference"* document for further details on the `SensorValue` class and its methods.

Usage:

```
local firstSensorValue
firstSensorValue = self:sensorValueForField("firstValue")
print(firstSensorValue:getName())
```

:setMeterView(meterName)

Sets a standard iMSB meter or panell as the widget background.

Parameters:

- `meterName`: (string) a string value indicating which meter type is to be applied as a background. The following meter types are available:
  - "`roundMeter`": displays the standard iMSB round meter canvas
  - "`lcdPanel`": displays the standard iMSB LCD panel
  - "`none`": does not apply any background (default)

Returns:

- `n/a`

Discussion:

For your convenience the standard round meter or lcd panel canvas can be used as background for your Widget if required. By default no specific background is applied to the Widget.

The colors of the view components can be adapted with the :setMeterViewColors method.

Usage:

```
self:setMeterView("roundMeter")
```

:setMeterViewColors({r=, g=, b=, a=}, {r=, g=, b=, a=} [,{r=, g=, b=, a=}])

Sets the border and panel color of a standard meter view. Optionally the intermediate color can be specified as well.

Parameters:

- `{r=, g=, b=, a=}`: (table) the color of the outer border expressed with their red, green and blue color components, as well as the alpha (transparency)

- `{r=, g=, b=, a=}`: (table) the color of the inner panel expressed with their red, green and blue color components, as well as the alpha (transparency)
- `{r=, g=, b=, a=}`: (table) the intermediate color expressed with their red, green and blue color components, as well as the alpha (transparency)

Returns:
- `n/a`

Discussion:

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the :`getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Usage:
```
self:setMeterViewColors(self:getColor("orangeColor"),
                        self:getColor("lightGrayColor"))
```

:getColor(colorName)

Convenience method to return a color representation given by its name.

Within the Widget scripting library colours are generically specified by a `{r=, g=, b=, a=}`: table. A set of standard colours have been predefined which can be retrieved via this function.

Parameters:
- `colorName`: (string) name of the color

Returns:
- `{r=, g=, b=, a=}`: (table) the RGB and alpha representation of the color

Discussion:

Color and alpha (transparency) components are expressed as a number between `0.0` and `1.0`.

An error message is returned if no color representation could be found for the given color name.

The following colors are predefined and can be retrieved:
- `“clearColor”`
- `“whiteColor”`
- `“blackColor”`
- `“greenColor”`
- `“greenColor2”`
- `“redColor”`
- `“darkBlueColor”`
- `“blueColor”`
- `“magentaColor”`
- `“yellowColor”`
- `“lightOrangeColor”`
- `“darkOrangeColor”`
- `“grayColor”`
- `“grayColor2”`

- “grayColor3”
- “grayColor4”
- “grayColor5”
- “grayColor6”
- “labelColor”
- “secondaryLabelColor”
- “tertiaryLabelColor”
- “alarmColor”
- “warningColor”
- “tintColor”
- “lcdColor”

Usage:

```
self:getColor(“lcdColor")
```

:isDarkMode()

Convenience method that indicates if the device is in dark mode or not.

Parameters:

- n/a

Returns:

- darkMode: (boolean) true if the device is in dark mode. False otherwise

Discussion:

When the device is in dark mode you may need to adapt your color definitions to suit this mode.

Note that all predefined colors return by the method :getColor already have their dark mode counter part. Thus no particular measures need to be taken.

Usage:

```
self:isDarkMode()
```

The Box class

Box are the fundamental building blocks of your Widget, and the Box class defines the behaviors that are common to all its subclasses. The Box class represents a rectangular zone that can be sized, moved, have a background and border color. It renders content within its bounds rectangle. In addition some interaction with the object can be performed.

Box can nest inside other boxes to create view hierarchies, which offer a convenient way to organize related content. Nesting a Box creates a parent-child relationship between the nested child boxes and the parent When a subview's visible area extends outside of the bounds of its superview, a clipping is applied.

The frame and bounds properties define the geometry of each Box. The frame property defines the origin and dimensions of the Box in the coordinate system of its superview. The bounds property defines the internal dimensions of the Box as it sees them, and its use is almost exclusive to custom drawing code.

Changes to several Box properties can be animated—that is, changing the property creates an animation starting at the current value and ending at the new value that you specify.

Box:new([parent,] {x=, y=, w=, h=})

Creates and returns a newly allocated Box object with the specified frame rectangle.

Parameters:

- parent: (userdata, optional): the parent object to which the current Box is to be added as a child view
- {x=, y=, w=, h=}: (table) the frame rectangle, i.e. position and size of the box. The origin of the frame is relative to the superview in which the box is nested

Returns:

- instance: (userdata) the instance of the created box

Discussion:

The new Box object must be inserted into the view hierarchy before it can be used. The parent attribute specifies to which object the Box is to be added as a child. If a parent is not provided, then it will be added directly to the widget.

Usage:

```
aBox = Box:new({x=10, y=10, w=100, h=100})
```

:setFrame({x=, y=, w=, h=} [, animationDur])

Sets the frame rectangle, which describes the Box location and size in its superview's coordinate system.

Parameters:

- {x=, y=, w=, h=}: (table) the frame rectangle, i.e. position and size of the box. The origin of the frame is relative to the superview in which the box is nested
- animationDur: (number, optional) the duration of the animation transition from the original to the destination frame

Returns:

- n/a

Discussion:

Changing the frame rectangle automatically redisplays the `Box`.

Changes to the frame property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aBox:setFrame({x=20, y=20, w=150, h=150}, 0.3)
```

`:frame()`

Returns the frame rectangle of a `Box` instance, which describes the `Box` location and size in its superview's coordinate system.

Parameters:

- `n/a`

Returns:

- `{x=, y=, w=, h=}`: (table) the frame rectangle, i.e. position and size of the box. The origin of the frame is relative to the superview in which the box is nested

Usage:

```
local myBoxRect
myBoxRect = aBox:frame()
```

`:bounds()`

Returns the bounds rectangle of a `Box` instance, which describes the view's location and size in its own coordinate system.

Parameters:

- `n/a`

Returns:

- `{x=, y=, w=, h=}`: (table) the bounds rectangle

Discussion:

The default `bounds` origin is (0,0) and the size is the same as the size of the rectangle in the `frame` property.

Usage:

```
local myBoxBounds
myBoxBounds = aBox:bounds()
```

`:setCenter({x=, y=} [, animationDur])`

Sets the center point of the `Box`'s frame rectangle

Parameters:

- `{x=, y=}`: (table) the coordinate of the center point, relative to the superview's coordinate system

- `animationDur`: (number, optional) the duration of the animation transition from the original to the destination center

Returns:

- `n/a`

Discussion:

The center point is specified in points in the coordinate system of its superview. Setting this property updates the origin of the rectangle in the `frame` property appropriately. Use this property, instead of the `frame` property, when you want to change the position of a Box. The center point is always valid, even when scaling or rotation factors are applied to the view's transform.

Changes to the center property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aBox:setCenter({x=20, y=20}, 0.3)
```

:center()

Returns the center point of the `Box`'s frame rectangle.

Parameters:

- `n/a`

Returns:

- `{x=, y=}`: (table) the coordinate of the center point, relative to the superview's coordinate system

Discussion:

The center point is specified in points in the coordinate system of its superview.

Usage:

```
local myBoxCenter
myBoxCenter = aBox:center()
```

:rotate(angle [, animationDur])

Rotates a `Box` around its center point by a given angle.

Parameters:

- `angle`: (number) the angle of rotation given in degrees
- `animationDur`: (number, optional) the duration of the animation transition from the original to the destination angle

Returns:

- `n/a`

Discussion:

The angle of rotation is expressed in degrees. A positive value specifies counterclockwise rotation and a negative value specifies clockwise rotation.

Changes to the angle property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aBox:rotate(30, 0.3)
```

:move(dx, dy [, animationDur])

Translates a `Box` instance by a given amount on the x and y axis of the coordinate system.

Parameters:

- `dx`: (number) the translation direction on the x axis expressed in points
- `dy`: (number) the translation direction on the y axis expressed in points
- `animationDur`: (number, optional) the duration of the animation transition from the original to the destination position

Returns:

- `n/a`

Discussion:

Changes to the angle property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aBox:move(10, 0, 0.3)
```

:scale(sx, sy [, animationDur])

Scales the size of a `Box` instance by a given scale factor expressed on the x and y axis of the coordinate system.

Parameters:

- `sx`: (number) the factor by which to scale the x-axis of the coordinate system
- `sy`: (number) the factor by which to scale the y-axis of the coordinate system
- `animationDur`: (number, optional) the duration of the animation transition from the original to the destination frame

Returns:

- `n/a`

Discussion:

Changes to the angle property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aBox:scale(0.8, 1.2, 0.3)
```

:setBackgroundColor({r=, g=, b=, a=} [, animationDur])

Sets the background color of `Box` instance.

Parameters:

- `{r=, g=, b=, a=}`: (table) the color expressed with their red, green and blue color components, as well as the alpha (transparency)
- `animationDur`: (number, optional) the duration of the animation transition from the original to the destination color

Returns:

- `n/a`

Discussion:

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the :`getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Changes to the color property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aBox:setBackgroundColor({r=255, g=128, b=152, a=1.0})
aBox:setBackgroundColor(self:getColor("whiteColor"))
```

`:setAlpha(alpha [, animationDur])`

Sets the alpha (transparency component) of a `Box` instance background color.

Parameters:

- `alpha`: (number) the alpha component to be applied to the background color
- `animationDur`: (number, optional) the duration of the animation transition from the original to the destination alpha

Returns:

- `n/a`

Discussion:

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

Changes to the color property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aBox:setAlpha(0.8)
```

`:setBorder(lineWidth [, cornerRadius])`

Draws a border line of a given width around a Box instance. Optionally rounds the corners of the shape by the provided radius.

Parameters:

- `lineWidth`: (number) the width of the border line in points to be drawn
- `cornerRadius`: (number, optional) the corner radius expressed in points if provided

Returns:

- n/a

Discussion:

By default the color of the border line is drawn with a black color. The border line color can be changed with the `:setBorderColor` method.

Usage:

```
aBox:setBorder(2, 10)
```

:setBorderColor({r=, g=, b=, a=} [, animationDur])

Sets the color of a `Box` border line.

Parameters:

- `{r=, g=, b=, a=}`: (table) the color expressed with their red, green and blue color components, as well as the alpha (transparency)

Returns:

- n/a

Discussion:

Before using this method a border line must be draw around the `Box` with the `:setBorder` method.

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the :`getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Usage:

```
aBox:setBorderColor({r=255, g=128, b=152, a=1.0})
aBox:setBorderColor(self:getColor("redColor"))
```

:setShadow({r=, g=, b=, a=}, opacity, radius [,{w=, h=})

Draws a shadow around a `Box` instance. Properties like the shadow color, its opacity and blur radius can be set, as well as an optional offset.

Parameters:

- `{r=, g=, b=, a=}`: (table) the shadow color expressed with their red, green and blue color components, as well as the alpha (transparency)
- `opacity`: (number) set the shadow opacity. Opacity is expressed by a number between `0` and `1`
- `radius`: (number) the blur radius (in points) used to render the layer's shadow
- `{w=, h=}`: (table, optional) the offset in points of the shadow. The default value of this property is `{w=0.0, h=0.0}`

Returns:

- n/a

Discussion:

Note that the alpha component does not have any effect on the shadow opacity. You need to set the `opacity` attribute in order the control the opacity.

By default the shadow offset is set to `{w=0.0, h=0.0}`, which means that it will be spread evenly around the `Box` shape.

Drawing a shadow can be animated. However, you should consider performing shadow animation by the means of the `:doAnimation` method.

Usage:

```
aBox:setShadow(self:getColor("blackColor"), 0.8, 5, {w=3.0, h=3.0})
```

:hide(hidden [, animationDur])

Allows to hide or show a shape.

Parameters:

- `hidden`: (boolean) a Boolean value that determines whether the view is hidden
- `animationDur`: (number, optional) the duration of the animation transition from the original to the destination frame

Returns:

- `n/a`

Discussion:

Changes to the hidden property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aBox:hide(true)
```

:doAnimation(func, animationDur [, completionFct])

This function allows to conveniently perform a batch animation of several attributes of a `Box` instance. Calls a user provided function and animates its execution. Optionnaly a callback function can be executed at the end of the animation.

Parameters:

- `func`: (function) the function you provide that will be executed in an animation
- animationDur: (number) the duration of the animation for the function execution
- `completionFct`: (function, optional) an optional function that will be called at the end of an animation

Returns:

- `n/a`

Discussion:

If you wish to animate several attributes of a `Box` instance you might consider doing this in a specific animation function you provided to this method.

After the animation has been performed an optional completion function is called.

Usage:

```
function myAnimation()
```

```
  aBox:rotate(45)
  aBox:move(30, 30)
  aBox:setBackgroundColor(self:getColor("greenColor"))
end

function myCallback()
  print("animation ended")
end

aBox:doAnimation(myAnimation, 4, myCallback)
```

:addTapHandler(handlerFct)

Adds a user provided tap handler function to the Box frame. When the user taps the Box within its frame the handler function gets called.

Parameters:

- handlerFct: (function) the function you provide that will be executed when the user taps on the object

Returns:

- n/a

Discussion:

A tap handler is typically used if you want to execute an action on the widget, such as for instance a sensor value announcement when the user taps on a Box object. It is your responsibility to handle user interaction with you widget.

The tap handler is limited and only active on the frame of the Box. That means that when the user taps the object somewhere within its frame the handler function will be called.

You can define only tap handler per Box instance. Each Box instance can however have its own dedicated tap handler.

Usage:

```
function myCallback()
  print(“user tapped on object")
end

aBox:addTapHandler(myCallback)
```

The Label class

A `Label` is a display object that displays one of informational text.

You can configure the overall appearance of a label's text, such as the font, its color and text alignment. In addition some predefined animation functions are provided for you.

The `Label` class inherits from the `Box` class. Thus all functions defined for the `Box` class also apply to a `Label` object.

`Label:new([parent,] {x=, y=, w=, h=})`

Creates and returns a newly allocated `Label` object with the specified frame rectangle.

Parameters:

- `parent`: (userdata, optional): the parent object to which the current `Label` is to be added as a child view
- `{x=, y=, w=, h=}`: (table) the frame rectangle, i.e. position and size of the `Label`. The origin of the frame is relative to the superview in which the `Label` is nested

Returns:

- `instance`: (userdata) the instance of the created `Label`

Discussion:

The new `Label` object must be inserted into the view hierarchy before it can be used. The `parent` attribute specifies to which object the `Label` is to be added as a child. If a `parent` is not provided, then it will be added directly to the widget.

Usage:

```
aLabel = Label:new({x=10, y=10, w=100, h=100})
```

`:setText(text)`

Sets as text string to the label

Parameters:

- `text`: (string): the text that the label displays

Returns:

- `n/a`

Usage:

```
aLabel:setText(“Hello world”)
```

`:setFont(fontName, fontSize)`

Assigns a font by its name and sets the size of the font to the `Label`.

Parameters:

- `fontName`: (string): the name of the font as defined in the iOS system
- `fontSize`: (number) the size in points to be given to the font

Returns:

- `n/a`

Discussion:

In general all fonts that are available as standard in iOS can be used. The font name must however follow the iOS naming convention. There are a couple of references in the web displaying the existing iOS fonts, such as http://iosfonts.com for instance.

The lcd font that is used in iMSB is a non standard custom font that is named `"lcd"` and can be accessed via this name.

Throughout iMSB `"Helvetica"` is the font generally used for standard text. You might need to consider this one to harmonise with the iMSB general look and fell.

Usage:

```
aBox:setFont("lcd", 17)
aBox:setFont("Helvetica-Bold", 32)
```

:setTextColor({r=, g=, b=, a=})

Sets the foreground color of the text in a `Label`

Parameters:

- `{r=, g=, b=, a=}`: (table) the text color expressed with its red, green and blue color components, as well as the alpha (transparency)

Returns:

- `n/a`

Discussion:

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the :`getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Changes to the color property can be animated. The duration of the animation is set via the `animationDur` parameter.

Usage:

```
aLabel:setTextColor({r=255, g=128, b=152, a=1.0})
aLabel:setTextColor(self:getColor("blackColor"))
```

:setTextAlignment(alignment)

Sets the alignment of the text within the `Label`'s frame rectangle.

Parameters:

- `alignment`: (string): a string defining the text alignment. One of the following attributes must be used:
  - `"left"`: text is visually left aligned
  - `"right"`: text is visually right aligned
  - `"center"`: text is centred within the bounding frame
  - `"justified"`: text is justified

Returns:

- `n/a`

Discussion:

By default if nothing is defined, the text alignment is always `“left”`.

Usage:

```
aLebel:setTextAlignment(“center”)
```

:setAlarmed(isAlarmed)

Enables or disables the alarm mode of a text `Label`. When a `Label` is in alarm mode it alternatively blinks between its foreground color and the alarm color.

Parameters:

- `isAlarmed`: (boolean): enables or disables the alarmed mode

Returns:

- `n/a`

Discussion:

By default the alarm color is set to the standard red `“alarmColor”`. The alarm color can however be changed with the `:setAlarmColor` method.

Usage:

```
aLable:setAlarmed(true)
```

:setAlarmColor({r=, g=, b=, a=})

Sets the alternate alarm color when a `Label` is blinking in alarm mode.

Parameters:

- `{r=, g=, b=, a=}`: (table) the text alarm color expressed with its red, green and blue color components, as well as the alpha (transparency)

Returns:

- `n/a`

Discussion:

The color attributes are only relevant when a Label ist set in alarm mode with the `:setAlarmed` method.

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the :`getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Usage:

```
aLabel:setAlarmColor({r=255, g=128, b=152, a=1.0})
aLabel:setAlarmColor(self:getColor("warningColor"))
```

The Image class

The `Image` let you efficiently draw any image that can be specified using an image file stored on the iCloud or on the device. An image content can be changed dynamically and at any time during the script execution.

An `Image` uses `setContentMode` method and the configuration of the image itself to determine how to display the image. It is best to specify images whose dimensions match the dimensions of the `Image` exactly, but it can scale your images to fit all or some of the available space. If the size of the `Image` itself changes, it automatically scales the image as needed. Images are composited onto the `Image`'s background. Any transparency in the image allows the `Image` background to show through.

In the current implementation only PNG images can be displayed.

The `Image` class inherits from the `Box` class. Thus all functions defined for the `Box` class also apply to a `Label` object.

Image files you use in your script may be stored on the iCloud or on the device itself if iCloud is not used. On the iCloud the files must reside in the *"iMSB"* app folder in a directory named *"images".* If this folder does not exist or is not named correctly the script will not be able to see them. Within the image root folder you may organise your image files in subfolder according to your liking. On the iMSB app's widget script editor you find a tool, that you can access via the tool bar to help you, copy and organise your files, as well as create nested subfolders. When iCloud is not used for storing data you need to use the iMSB image organiser to copy your files from their original to the correct location.

`Image:new([parent,] {x=, y=, w=, h=})`

Creates and returns a newly allocated `Image` object with the specified frame rectangle.

Parameters:

- `parent`: (userdata, optional): the parent object to which the current `Image` is to be added as a child view
- `{x=, y=, w=, h=}`: (table) the frame rectangle, i.e. position and size of the `Image`. The origin of the frame is relative to the superview in which the `Image` is nested

Returns:

- `instance`: (userdata) the instance of the created `Image`

Discussion:

The new `Image` object must be inserted into the view hierarchy before it can be used. The `parent` attribute specifies to which object the `Image` is to be added as a child. If a `parent` is not provided, then it will be added directly to the widget.

Usage:

```
anImage = Image:new(aBox, {x=10, y=10, w=100, h=100})
```

`:setImage(imageName)`

Sets the image to display provided by its file name into an `Image` instance.

Parameters:

- `imageName`: (string, optional): the file path of the image to be loaded relative to the image store root directory

Returns:
- n/a

Discussion:

Image files you use need be stored on the iCloud or on the device itself if iCloud is not used. On the iCloud the files must reside in the *"iMSB"* app folder in a directory named *"images".* When iCloud is not used you need to use the iMSB image organiser to copy your files from their original to the correct location

The path to the image file you provide to this method must be specified relative to the root image file folder.

In the current implementation only PNG images can be displayed.

Usage:

```
anImage:setImage("subFolder/testImage.png")
```

:setContentMode(mode)

Sets the option to specify how an `Image` adjusts its content when displayed or its size is changed. Provides layout behavior for the `Image` content, as opposed to the frame of the view. This property also affects how the content is scaled to fit the view.

Parameters:
- `mode`: (string):preferred content behaviour. The following content modes can be specified:
  - `"scaleToFill"`: option to scale the content to fit the size of itself by changing the aspect ratio of the content if necessary.
  - `"aspectFit"`: option to scale the content to fit the size of the view by maintaining the aspect ratio. Any remaining area of the view's bounds is transparent.
  - `"aspectFill"`: option to scale the content to fill the size of the view. Some portion of the content may be clipped to fill the view's bounds.
  - `"center"`: option to center the content in the view's bounds, keeping the proportions the same.
  - `"top"`: option to center the content aligned at the top in the view's bounds.
  - `"bottom"`: option to center the content aligned at the bottom in the view's bounds.
  - `"left"`: option to align the content on the left of the view.
  - `"right"`: option to align the content on the right of the view.

Returns:
- n/a

Discussion:

By default when an `Image` is created its content scaling is set to `"aspectFit"`.

Usage:

```
anImage:setContentMode("aspectFill")
```

:setAlarmed(isAlarmed)

Enables or disables the alarm mode of an `Image`. When an `Image` is in alarm mode it alternatively blinks between its standard image and an alternate image.

Parameters:
- `isAlarmed`: (boolean): enables or disables the alarmed mode

Returns:
- `n/a`

Discussion:

You need to explicitly set an alternate image for the alarmed state using method `:setAlarmImage`. By default the alarmed image is empty.

Usage:

```
anImage:setAlarmed(true)
```

`:setAlarmImage(imageName [{r=, g=, b=, a=}])`

Sets the alternate alarm image when an `Image` is blinking in alarm mode. Optionally sets a specific view background color for the alarm mode.

Parameters:
- `imageName`: (string, optional): the file path of the alarm image to be loaded relative to the image store root directory
- `{r=, g=, b=, a=}`: (table, optional) the background color you wish to apply to the view in the alarmed mode. The color expressed with its red, green and blue color components, as well as the alpha (transparency)

Returns:
- `n/a`

Discussion:

When setting an alarm image, the image will be shown in alternation with the standard image. By default the alarm image is empty. Not specifying an alarm image will alternately just display the normal image.

Image files you use need be stored on the iCloud or on the device itself if iCloud is not used. On the iCloud the files must reside in the *“iMSB”* app folder in a directory named *“images”.* When iCloud is not used you need to use the iMSB image organiser to copy your files from their original to the correct location. The path to the image file you provide to this method must be specified relative to the root image file folder.

In addition an alternate background color can be set.for the alarm mode. If no alternate color is specified the standard background color will be used instead.

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the :`getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Usage:

```
anImage:setAlarmImage(“testImageRed.png”)
```

The Layer class

`Layer` objects are 2D surfaces organized in a 3D space and are at the heart of everything you do when performing complex animations or drawing. Like `Box` (and its subclasses), layers manage information about the geometry, content, and visual attributes of their surfaces. Unlike views, layers do not define their own appearance. A layer merely manages the state information surrounding a bitmap. Layers do not necessarily do any actual drawing in your script. Instead, a layer captures the content your script provides and caches it in a bitmap. When you subsequently change a property of the layer, all you are doing is changing the state information associated with the layer object. Thus when performing animations you should always consider doing so with a `Layer` object instead of a `Box`, as system resources are handled much more efficiently.

A `Layer` object cannot be created and reside on its own in a script, but must be at least added to a `Box` (or one if its subclasses). A `Layer` can be added as a sublayer to an existing one. You can as such construct a whole layer hierarchy in a `Box` or a `Layer` who's ordering can be manipulated in script runtime.

When performing animations you should consider keeping the content of your layers simple and add simple drawing objects in separate layers. Layers share very similar attributes to a `Box`, such as setting background colors, borders, positioning and resizing, moving or scaling. When performing animation on such attributes you should consider doing this with a `Layer` instead of directly with a `Box`, although both methods are possible. Animatable `Layer` attributes do not provide a way to control the animation duration directly on the attribute itself, but have an implicit animation built in already that allows for a smooth transition from one state to the other. If more fine grained animation is required you should consider doing so with the `:doAnimation` method.

A `Layer` allows to draw complex drawing shapes with cubic Bezier splines. This is performed by defining a so called path that will be rendered at runtime. A path specifies a set of graphic primitives, such as lines, arcs, ellipses, etc. you use to construct your drawing. Each of these drawing primitives creates a subpath within a path. Attributes of you drawing such as line color, style, background colors, etc. may as well be specified. Constructing a path must be done within an `:beginPath` and `:applyPath` call in your code in order for the system to know which primitives belong to the path you wish to draw. Unlike other objects a path does not have an implicit animation. The path within a `Layer` may however be as well animated using a concrete animation function.

A shape path may also be used for creating complex or special effect shadows. Constructing a shadow path must be done between a `:beginPath` and `:applyShadowPath` call. If you specify a shadow path, the layer creates its shadow using the specified path instead of the layer's composited alpha channel. The path you provide defines the outline of the shadow. It is filled using the current shadow color, opacity, and blur radius.

Layers can be used as a masking layer to mask or blend the content in its parent the layer hierarchy. A masking `Layer` has same drawing and layout properties of any other layer. A mask `Layer` is located relative to its parent. It is used in a similar way to a sublayer, but it does not appear as a normal sublayer. Instead of being drawn inside the parent, the mask `Layer` defines the part of the parent layer that is visible. The `Layer`'s alpha channel determines how much of the layer's content and background shows through. Fully or partially opaque pixels allow the underlying content to show through, but fully transparent pixels block that content. If the mask `Layer` is smaller than the parent layer, only the parts of the parent that intersect the mask will be visible.

When creating a masking `Layer` you need to define a path that will determine the shape used for masking your content. Not associating a shape to a masking layer will completely render the layers in the hierarchy invisible.

Basic Layer functions

`Layer:new(parent, {x=, y=, w=, h=})`

Creates and returns a newly allocated `Layer` object with the specified frame rectangle. The `Layer` is allocated added to a `Box` or as a sublayer of an existing `Layer`.

Parameters:

- `parent`: (userdata): the parent object to which the current `Layer` is to be added as a child. The parent must be a `Box` (or one of its subclasses) or a `Layer`
- `{x=, y=, w=, h=}`: (table) the frame rectangle, i.e. position and size of the `Layering` points. The origin of the frame is relative to the superview or parent layer in which the `Layer` is nested

Returns:

- `instance`: (userdata) the instance of the created `Layer`

Discussion:

The created `Layer` is always added as the last object, i.e. top object in the layer hierarchy.

Usage:

```
aLayer = Layer:new(aBox, {x=10, y=10, w=100, h=100})
```

`Layer:newMask(parent)`

Creates and returns a newly allocated masking `Layer` object whose alpha channel is used to mask the visible layer's content. The `Layer` is allocated added to a `Box` or as a sublayer of an existing `Layer`.

Parameters:

- `parent`: (userdata): the parent object to which the current `Image` is to be added as a child view

Returns:

- `instance`: (userdata) the instance of the created `masking` layer

Discussion:

The `Layer`'s alpha channel determines how much of the layer's content and background shows through. Fully or partially opaque pixels allow the underlying content to show through, but fully transparent pixels block that content. If the mask `Layer` is smaller than the parent layer, only the parts of the parent that intersect the mask will be visible.

By default a masking `Layer` has the same bounds as its parent. The Layer's frame can however be adapted with the `:setFrame` method.

When creating a masking `Layer` you need to define a path that will determine the shape used for masking your content. Not associating a shape to a masking layer will completely render the layers in the hierarchy invisible.

A masking layer can be inverted by setting the the `invert` parameter of the `:applyPath` method to `true`.

Usage:

```
aMaskingLayer = Layer:newMask(backgroundLayer)
aMaskingLayer:setOpacity(0.8)
aMaskingLayer:beginPath()
  aMaskingLayer:ellipseInRect({x=20, y=30, w=60, h=40})
aMaskingLayer:applyPath(true)
```

:delete()

Removes the `Layer` from the layer hierarchy and deallocates it.

Parameters:

- `n/a`

Returns:

- `n/a`

Usage:

```
aLayer:delete()
```

:reorder(index)

Moves the specified `Layer` into the receiver's list of sublayers at the specified index.

Parameters:

- `index`: (number): the index of the position in the layer hierarchy where the layer is to be moved

Returns:

- `n/a`

Discussion:

The `index` must be a valid `0`-based index into the sublayers array. An `index 0` represents the bottom layer whereas index `n` is the topmost layer.

Usage:

```
aLayer:reorder(2)
```

:setFrame({x=, y=, w=, h=})

Sets the frame rectangle, which describes the `Layer` location and size in its superview's or parent layer coordinate system.

Parameters:

- `{x=, y=, w=, h=}`: (table) the frame rectangle, i.e. position and size of the `Layer`. The origin of the frame is relative to the superview or parent layer in which the layer is nested

Returns:

- `n/a`

Discussion:

Changing the frame rectangle automatically redisplays the `Layer` with an implicit animation.

Usage:

```
aLayer:setFrame({x=20, y=20, w=150, h=150})
```

:frame()

Returns the frame rectangle of a `Layer` instance, which describes the `Layer` location and size in its superview's or parent layer coordinate system.

Parameters:

- `n/a`

Returns:

- `{x=, y=, w=, h=}`: (table) the frame rectangle, i.e. position and size of the layer. The origin of the frame is relative to the superview or parent layer in which the layer is nested

Usage:

```
local myLayerRect
myLayerRect = aLayer:frame()
```

:setBounds({x=, y=, w=, h=})

Sets the bounding rectangle, which describes the `Layer` location and size in its superview's or parent layer coordinate system.

Parameters:

- `{x=, y=, w=, h=}`: (table) the frame bounds, i.e. location and size of the `Layer`. The origin of the frame is relative to the superview or parent layer in which the layer is nested

Returns:

- `n/a`

Discussion:

The bounds rectangle is the origin and size of the layer in its own coordinate space. The default `bounds` origin is (0,0) and the size is the same as the size of the rectangle in the `frame` property.

Changing the bounds rectangle automatically redisplays the `Layer` with an implicit animation.

Usage:

```
aLayer:setBounds({x=0, y=0, w=150, h=150})
```

:bounds()

Returns the bounds rectangle of a `Layer` instance, which describes the `Layer`'s location and size in its own coordinate system.

Parameters:

- `n/a`

Returns:

- `{x=, y=, w=, h=}`: (table) the bounds rectangle

Discussion:

The default `bounds` origin is (0,0) and the size is the same as the size of the rectangle in the `frame` property.

Usage:

```
local myLayerBounds
myLayerBounds = aLayer:bounds()
```

:setPosition({x=, y=})

The `Layer`'s position in its superlayer's coordinate space.

Parameters:

- `{x=, y=}`: (table) the coordinate of the position point, relative to the superview's coordinate system

Returns:

- `n/a`

Discussion:

The value of this property is specified in points and is always specified relative to the value in the `anchor` property, i.e. its location within the Layer is depending on the anchor.

Setting this property updates the origin of the rectangle in the `frame` property appropriately. Use this property, instead of the `frame` property, when you want to change the position of a `Layer`.

Changing this property automatically redisplays the `Layer` with an implicit animation.

Usage:

```
aLayer:setPosition({x=20, y=20})
```

:position()

Returns the `Layer`'s position in its superlayer's coordinate space.

Parameters:

- `n/a`

Returns:

- `{x=, y=}`: (table) the coordinate of the position, relative to the superview's or parent layer coordinate system

Discussion:

The value of this property is specified in points and is always specified relative to the value in the `anchor` property, i.e. its location within the Layer is depending on the anchor.

Usage:

```
local myLayerPosition
myLayerPosition = aLayer:position()
```

:setAnchor({x=, y=)

Defines the anchor point of the layer's bounds rectangle. The anchor point is always expressed relative to the layer's width and height property.

Parameters:

- `{x=, y=}`: (table) the position of the anchor point relative to the layer bounding box. The value of the `x` and `y` coordinates must be between `0` and `1`

Returns:

- `instance`: (userdata) the instance of the created `Image`

Discussion:

You specify the value for this property using the unit coordinate space. The default value of this property is (0.5, 0.5), which represents the center of the layer's bounds rectangle. All geometric manipulations to the view occur about the specified point. For example, applying a rotation transform to a layer with the default anchor point causes the layer to rotate around its center. Changing the anchor point to a different location would cause the layer to rotate around that new point.

The value of the `x` and `y` coordinates must be between `0` and `1`.

Modifying the anchor points does not have any effect on the `Layer`'s frame rectangle. The value of the `position` attribute will however be updated to reflect the anchor change.

Usage:

```
aLayer:setAnchor({x=0.0, y=0.0)
```

:anchor()

Returns the anchor point of the layer's bounds rectangle. The anchor point is always expressed relative to the layer's width and height property.

Parameters:

- `n/a`

Returns:

- `{x=, y=}`: (table) the position of the anchor point relative to the layer bounding box. The value of the `x` and `y` coordinates must be between `0` and `1`

Discussion:

The default value of this property is (0.5, 0.5), which represents the center of the layer's bounds rectangle. All geometric manipulations to the view occur about the specified point. For example, applying a rotation transform to a layer with the default anchor point causes the layer to rotate around its center. Changing the anchor point to a different location would cause the layer to rotate around that new point.

The value of the `x` and `y` coordinates are expressed between `0` and `1`.

Usage:

```
local myLayerAnchor
myLayerAnchor = aLayer:anchor()
```

:hide(hidden)

Allows to hide or show a Layer.

Parameters:

- hidden: (boolean) a Boolean value that determines whether the layer is hidden

Returns:

- n/a

Discussion:

Changing this property hides or displays the Layer with an implicit animation.

Usage:

```
aLaywer:hide(true)
```

:setBackgroundColor({r=, g=, b=, a=})

Sets the background color of Layer instance.

Parameters:

- {r=, g=, b=, a=}: (table) the color expressed with their red, green and blue color components, as well as the alpha (transparency)

Returns:

- n/a

Discussion:

Color components and alpha (transparency) are expressed as a number between 0.0 and 1.0.

As an alternative you can use the :getColor method from the widget class to determine the color components using on of the predefined colors.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setBackgroundColor({r=255, g=128, b=152, a=1.0})
aLayer:setBackgroundColor(self:getColor("whiteColor"))
```

:setOpacity(opacity)

Sets the opacity of a Layer instance background color.

Parameters:

- opacity: (number) the opacity component to be applied to the background color

Returns:

- n/a

Discussion:

The value of this property must be in the range 0.0 (transparent) to 1.0 (opaque). Values outside that range are clamped to the minimum or maximum. The default value of this property is 1.0.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setOpacity(0.8)
```

:setBorderWidth(lineWidth)

Draws a border line of a given width around a `Layer` instance.

Parameters:

- `lineWidth`: (number) the width of the border line in points to be drawn

Returns:

- `n/a`

Discussion:

By default the color of the border line is drawn with a black color. The border line color can be changed with the `:setBorderColor` method. The corner radius can be changed with the `:setCornerRadius` method.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setBorderWidth(2, 10)
```

:setCornerRadius(cornerRadius)

Rounds the corners of a `Layer` bounding box by the provided radius.

Parameters:

- `cornerRadius`: (number) the corner radius expressed in points

Returns:

- `n/a`

Discussion:

The border line color can be changed with the `:setBorderColor` method. The line width can be changed with the `:setBorderWidth` method.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setCornerRadius(4)
```

:setBorderColor({r=, g=, b=, a=})

Sets the color of a `Layer` border line.

Parameters:

- `{r=, g=, b=, a=}`: (table) the color expressed with their red, green and blue color components, as well as the alpha (transparency)

Returns:

- `n/a`

Discussion:

Before using this method a border line must be draw around the `Layer` with the `:setBorderWidth` method.

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the :`getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setBorderColor({r=255, g=128, b=152, a=1.0})
aLayer:setBorderColor(self:getColor("redColor"))
```

:setShadow({r=, g=, b=, a=}, opacity, radius [,{w=, h=})

Draws a shadow around the bounds of a `Layer` instance. Properties like the shadow color, its opacity and blur radius can be set, as well as an optional offset.

Parameters:

- `{r=, g=, b=, a=}`: (table) the shadow color expressed with their red, green and blue color components, as well as the alpha (transparency)
- `opacity`: (number) set the shadow opacity. Opacity is expressed by a number between `0` and `1`
- `radius`: (number) the blur radius (in points) used to render the `Layer`'s shadow
- `{w=, h=}`: (table, optional) the offset in points of the shadow. The default value of this property is `{w=0.0, h=0.0}`

Returns:

- `n/a`

Discussion:

Note that the alpha component does not have any effect on the shadow opacity. You need to set the `opacity` attribute in order the control the opacity.

By default the shadow offset is set to `{w=0.0, h=0.0}`, which means that it will be spread evenly around the `Box` shape.

Alternatively you may draw a more complex shadow around you `Layer` by using a path shape. If you specify a drawing path an apply it with `:applyShapePath`, the layer creates its shadow using the specified path instead of the layer's composited alpha channel.

Usage:

```
aLayer:setShadow(self:getColor("blackColor"), 0.8, 5, {w=3.0, h=3.0})
```

Layer transformation and animation functions

:rotate(angle)

Rotates a `Layer` around its anchor point by a given angle.

Parameters:

- `angle`: (number) the angle of rotation given in degrees

Returns:

- `n/a`

Discussion:

The angle of rotation is expressed in degrees. A positive value specifies counterclockwise rotation and a negative value specifies clockwise rotation.

The rotation center is dependent on the location of the anchor point.

Applying a rotation to a `Layer` is performed with an implicit animation.

Usage:

```
aLabel:rotate(30)
```

:move(dx, dy)

Translates a `Layer` instance by a given amount on the x and y axis of the coordinate system.

Parameters:

- `dx`: (number) the translation direction on the x axis expressed in points
- `dy`: (number) the translation direction on the y axis expressed in points

Returns:

- `n/a`

Discussion:

Applying a translation to a `Layer` is performed with an implicit animation.

Usage:

```
aLabel:move(10,0)
```

:scale(sx, sy)

Scales the size of a `Layer` instance by a given scale factor expressed on the x and y axis of the coordinate system.

Parameters:

- `sx`: (number) the factor by which to scale the x-axis of the coordinate system
- `sy`: (number) the factor by which to scale the y-axis of the coordinate system

Returns:

- `n/a`

Discussion:

Applying a scale factor to a `Layer` is performed with an implicit animation.

Usage:

```
aLayer:scale(0.8, 1.2)
```

`:doAnimation(func, animationDur [, completionFct])`

This function allows to conveniently perform a batch animation of several attributes of a `Layer` instance. Calls a user provided function and animates its execution. Optionally a callback function can be executed at the end of the animation.

Parameters:

- `func`: (function) the function you provide that will be executed in an animation
- animationDur: (number) the duration of the animation for the function execution
- `completionFct`: (function, optional) an optional function that will be called at the end of an animation

Returns:

- `n/a`

Discussion:

If you wish to animate several attributes of a `Layer` instance you might consider doing this in a specific animation function you provided to this method.

After the animation has been performed an optional completion function is called.

Usage:

```
function myAnimation()
  aLayer:rotate(45)
  aLayer:move(30, 30)
  aLayer:setBackgroundColor(self:getColor("greenColor"))
end

function myCallback()
  print("animation ended")
end

aLayer:doAnimation(myAnimation, 4, myCallback)
```

Defining Layer path

`:beginPath()`

Starts the definition of a new drawing path that will be applied to a `Layer` instance.

Parameters:

- `n/a`

Returns:

- `n/a`

Discussion:

A path specifies a set of mathematical graphic primitives, such as lines, arcs, ellipses, etc. you use to construct your complex drawing. The `:beginPath` instruction tells the app that subsequent drawing primitives belong to the started path.

Constructing a path must always be done within an `:beginPath` and `:applyPath` call in your code in order for the system to know which primitives belong to the path you wish to draw.

You can only define multiple drawing path per `Layer` instance, although this is not recommended.

Unlike other objects a path does not have an implicit animation.

Usage:

```
aLayer:beginPath()
  aLayer:moveToPoint({x=10, y=10})
  aLayer:lineToPoint({x=90, y=10})
  aLayer:lineToPoint({x=50, y=80})
  aLayer:closePath()
aLayer:applyPath()
aLayer:setFillColor(self:getColor("redColor"))
```

`:applyPath([invert])`

Ends the definition of a new drawing path and applies it to a `Layer` instance.

Parameters:

- `invert:` (boolean, optional) set the fill rule to inverted when filling the multiple crossing closed path of a `Layer`

Returns:

- `n/a`

Discussion:

A path specifies a set of mathematical graphic primitives, such as lines, arcs, ellipses, etc. you use to construct your complex drawing. The `:endPath` applies the drawing primitives you defined to your `Layer` for rendering.

Constructing a path must be done within an `:beginPath` and `:applyPath` call in your code in order for the system to know which primitives belong to the path you wish to draw.

. You can only define multiple drawing path per `Layer` instance, although this is not recommended.

Unlike other objects a path does not have an implicit animation.

When the `invert` parameter is set to true an even-odd winding rule is applied when rendering a path. It counts the total number of path crossings. If the number of crossings is even, the point is outside the path. If the number of crossings is odd, the point is inside the path and the region containing it should be filled.

Usage:

```
aLayer:beginPath()
  aLayer:ellipseInRect({x=10, y=40, w=80, h=20})
  aLayer:ellipseInRect({x=10, y=40, w=80, h=20}, -90, {x=50, y=50})
aLayer:applyPath(true)
aLayer:setStrokeColor(self:getColor(“blueColor"))
```

:applyShadowPath()

Ends the definition of an explicit shadow path and applies it to a `Layer`'s shadow.

Parameters:

- `n/a`

Returns:

- `n/a`

Discussion:

A shape path may be used for creating a complex or a special effect shadow on a `Layer`. When doing so you must explicitly add the created shape to its shadow path property. .Constructing a shadow path must be done between a `:beginPath` and `:applyShadowPath` call. If you specify a shadow path, the layer creates its shadow using the specified path instead of the layer's composited alpha channel. The path you provide defines the outline of the shadow. It is filled using the current shadow color, opacity, and blur radius.

Usage:

```
aLayer = Layer:new(box, {x=75, y=75, w=150, h=150});
aLayer:setBackgroundColor(self:getColor("blueColor"))
aLayer:setShadow(self:getColor("blackColor"), 0.8, 5, {w=3.0, h=3.0})
aLayer:beginPath()
  aLayer:ellipseInRect({x=-20, y=150-10, w=150+40, h=20})
aLayer:applyShadowPath()
```

:moveToPoint({x=, y=})

Starts a path at a specified location given its coordinates.

Parameters:

- `{x=, y=}`: (table) the position to move the starting point of the path

Returns:

- `n/a`

Usage:

```
aLayer:moveToPoint({x=10, y=10})
```

:lineToPoint({x=, y=} [,rotation, [{x=, y=}]])

Draws a line from the last position to the specified location. Optionally the line can be rotated around a given rotation point.

Parameters:

- `{x=, y=}`: (table) the coordinates of the location to which to draw the line
- `rotation`: (number, optional) an optional rotation angle to apply to the line
- `{x=, y=}`: (table, optional) the coordinates of the rotation point

Returns:

- `n/a`

Discussion:

The origin of the line is set at the end point of the last specified drawing primitive. You may reposition the origin with the `:moveToPoint` method.

A drawn line can be optionally rotated around a defined point. Per default the rotation point is set at coordinates (0, 0) in the `Layer` coordinate system.

The angle of rotation is expressed in degrees. A positive value specifies counterclockwise rotation and a negative value specifies clockwise rotation.

Usage:

```
aLayer:lineToPoint({x=80, y=10}, 20, {x=10, y=10})
```

:rectangle({x=, y=, w=, h=} [, roundn, [,rotation, [{x=, y=}]]])

Draws a rectangle shape given its bound box. Optionally corner roundness can be set, as well as rotation around a given rotation point.

Parameters:

- `{x=, y=, w=, h=}`: (table) the bounding box coordinates in which the rectangle will be drawn
- `roundn`: (number, optional) the corner radius expressed in points if provided
- `rotation`: (number, optional) an optional rotation angle to apply to the bounding box
- `{x=, y=}`: (table, optional) the coordinates of the rotation point

Returns:

- `n/a`

Discussion:

A drawn rectangle can be optionally rotated around a defined point. Per default the rotation point is set at coordinates (0, 0) in the `Layer` coordinate system.

The angle of rotation is expressed in degrees. A positive value specifies counterclockwise rotation and a negative value specifies clockwise rotation.

Usage:

```
aLayer:rectangle({x=20, y=40, w=80, h=20}, 4, 30, {x=40, y=50})
```

:ellipseInRect({x=, y=, w=, h=}[,rotation, [{x=, y=}]])

Draws an ellipse shape given its bound box. Optionally the ellipse can be rotated around a given rotation point.

Parameters:

- `{x=, y=, w=, h=}`: (table) the bounding box coordinates in which the ellipse will be drawn
- `rotation`: (number, optional) an optional rotation angle to apply to the bounding box
- `{x=, y=}`: (table, optional) the coordinates of the rotation point

Returns:

- `n/a`

Discussion:

A drawn ellipse can be optionally rotated around a defined point. Per default the rotation point is set at coordinates (0, 0) in the `Layer` coordinate system.

The optional angle of rotation is expressed in degrees. A positive value specifies counterclockwise rotation and a negative value specifies clockwise rotation.

Usage:

```
aLayer:ellipseInRect({x=10, y=10, w=80, h=60})
```

:arc({x=, y=}, r, startAngle, endAngle, clockwise)

Draws an arc to a graphics path, possibly preceded by a straight line segment.

Parameters:

- `{x=, y=}`: (table) the coordinates of the center point of the arc
- r: (number) the radius of the arc expressed in points
- `startAngle`: (number) the angle (in degrees) that determines the starting point of the arc, measured from the x-axis in the current `Layer` space
- `endAngle`: (number) the angle (in degrees) that determines the ending point of the arc, measured from the x-axis in the current `Layer` space
- `clockwise`: (number) a Boolean value that specifies whether or not to draw the arc in the clockwise direction

Returns:

- `n/a`

Discussion:

An arc is a segment of a circle with radius `r` centered at a point (x,y). When you call this method, you provide the center point, radius, and two angles in degrees. The app uses this information to determine the end points of the arc, and then approximates the new arc using a sequence of cubic Bézier curves. The `clockwise` parameter determines the direction in which the arc is created.

If the specified path already contains a drawing, the app implicitly adds a line connecting the subpath's current point to the beginning of the arc. If the path is empty, the app creates a new subpath with a starting point set to the starting point of the arc.

The ending point of the arc becomes the new current point of the path.

For another way to draw an arc in a path, `:arcToPoint`.

Usage:

```
aLayer:arc({x=50, y=50}, 40, 0, 90, false)
```

:arcToPoint(p1{x=, y=}, p2{x=, y=}, r)

Draws an arc to a graphics path, possibly preceded by a straight line segment.

Parameters:

- p1`{x=, y=}`: (table) the coordinate of the `Layer` space for the end point of the first tangent line. The first tangent line is drawn from the current point to (x1,y1)
- p2`{x=, y=}`: (table) the coordinate of the `Layer` space for the end point of the second tangent line. The second tangent line is drawn from (x1,y1) to (x2,y2)
- r: (number) the radius of the arc expressed in points

Returns:

- `n/a`

Discussion:

This method uses a sequence of cubic Bézier curves to create an arc that is tangent to the line from the current point to (x1,y1) and to the line from (x1,y1) to (x2,y2). The start and end points of the arc are located on the first and second tangent lines, respectively. The start and end points of the arc are also the “tangent points” of the lines.

If the current point and the first tangent point of the arc (the starting point) are not equal, the app appends a straight line segment from the current point to the first tangent point.

The ending point of the arc becomes the new current point of the path.

For another way to draw an arc in a path, `:arc`.

Usage:

```
aLayer:moveToPoint({x=10, y=10})
aLayer:arcToPoint({x=80, y=50}, {x=10, y=80}, 20)
```

`:curveToPoint(cp1{x=, y=}, cp2{x=, y=}, {x=, y=})`

Draws a cubic Bézier curve to a graphics path.

Parameters:

- cp1`{x=, y=}`: (table) the coordinate of the first control point
- cp2`{x=, y=}`: (table) the coordinate of the second control point
- `{x=, y=}`: (table) the coordinate of the end point of the curve

Returns:

- `n/a`

Discussion:

Draws a cubic Bézier curve from the current point in a path to the specified location using two control points. Before returning, this function updates the current point to the specified location (x,y).

Usage:

```
aLayer:moveToPoint({x=40, y=10})
aLayer:curveToPoint({x=80, y=50}, {x=10, y=80}, {x=10, y=80})
```

`:closePath()`

Closes and completes a subpath in a graphics path, by appending a line to the first point in the path.

Parameters:

- `n/a`

Returns:

- `n/a`

Discussion:

Appends a line from the current point to the starting point of the current path and closes the subpath.

After closing the subpath, your script can begin a new subpath without first calling `:moveToPoint`. In this case, a new subpath is implicitly created with a starting and current point equal to the previous subpath's starting point.

Usage:

```
aLayer:moveToPoint({x=10, y=10})
aLayer:lineToPoint({x=90, y=10})
aLayer:lineToPoint({x=50, y=80})
aLayer:closePath()
```

Setting path attributes

`:setStrokeColor({r=, g=, b=, a=})`

The color used to stroke the `Layer`'s path.

Parameters:

- `{r=, g=, b=, a=}`: (table) the color expressed with their red, green and blue color components, as well as the alpha (transparency)

Returns:

- `n/a`

Discussion:

This method has only an effect on the stroke color of the drawing path specified for the `Layer`.

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the `:getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setStrokeColor(self:getColor("blueColor"))
```

`:setLineWidth(width)`

Specifies the line width of the `Layer`'s path.

Parameters:

- `width`: (number) the line width in point when stroking the path

Returns:

- `n/a`

Discussion:

This method has only an effect on the stroke width of the drawing path specified for the `Layer`.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setAnchor({x=0.0, y=0.0)
```

:setStrokeStart(start)

The relative location at which to begin stroking the path of a `Layer`.

Parameters:

- `start`: (number) the value of relative start or the stroke with respect to its total length. The value must be between `0` and `1`

Returns:

- `n/a`

Discussion:

The value of this property must be in the range `0.0` to `1.0`. The default value of this property is `0.0`.

Combined with the `:setStrokeEnd` method, this property defines the subregion of the path to stroke. The value in this property indicates the relative point along the path at which to begin stroking while `:setStrokeEnd` defines the end point. A value of `0.0` represents the beginning of the path while a value of `1.0` represents the end of the path. Values in between are interpreted linearly along the path length.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setStrokeStart(0.2)
```

:setStrokeEnd(end)

The relative location at which to stop stroking the path of a `Layer`.

Parameters:

- `end`: (number) the value of relative end or the stroke with respect to its total length. The value must be between `0` and `1`

Returns:

- `n/a`

Discussion:

The value of this property must be in the range `0.0` to `1.0`. The default value of this property is `0.0`.

Combined with the `:setStrokeStart` method, this property defines the subregion of the path to stroke. The value in this property indicates the relative point along the path at which to finish stroking while `:setStrokeStart` defines the start point. A value of `0.0` represents the beginning of the path while a value of `1.0` represents the end of the path. Values in between are interpreted linearly along the path length.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setStrokeEnd(0.8)
```

:setFillColor({x=, y=)

The color used to fill the `Layer`'s path.

Parameters:

- `{r=, g=, b=, a=}`: (table) the color expressed with their red, green and blue color components, as well as the alpha (transparency)

Returns:

- `n/a`

Discussion:

This method has only an effect on the fill color of the drawing path specified for the `Layer`.

Color components and alpha (transparency) are expressed as a number between `0.0` and `1.0`.

As an alternative you can use the :`getColor` method from the `widget` class to determine the color components using on of the predefined colors.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setFillColor(self:getColor("blueColor"))
```

:setLineDashPattern({ })

The dash pattern applied to the `Layers`'s path when stroked.

Parameters:

- `{ }`: (table) the position of the anchor point relative to the layer bounding box. The value of the `x` and `y` coordinates must be between `0` and `1`

Returns:

- `n/a`

Discussion:

The dash pattern is specified as an array of numbers that specify the lengths of the painted segments and unpainted segments, respectively, of the dash pattern.

For example, passing an array with the values `{2, 3}` sets a dash pattern that alternates between a 2-user-space-unit-long painted segment and a 3-user-space-unit-long unpainted segment. Passing the values `{10, 5, 5, 5}` sets the pattern to a 10-unit painted segment, a 5-unit unpainted segment, a 5-unit painted segment, and a 5-unit unpainted segment.

The default value if this property is a solid line. Calling the method with an empty argument set the dash pattern to solid.

Changing this property is performed with an implicit animation.

Usage:

```
aLayer:setLineDashPattern({2,4})
aLayer:setLineDashPattern()
```

`:setLineJoin(joinType)`

Specifies the line join style for the `Layers`'s path.

Parameters:

- `joinType`: (number) a number indicating the line join type. Following line join ties can be used:
  - `1`: specifies a miter line shape of the joints between connected segments of a stroked path
  - `2`: specifies a round line shape of the joints between connected segments of a stroked path
  - `3`: Specifies a bevel line shape of the joints between connected segments of a stroked path

Returns:

- `n/a`

Discussion:

The line join style specifies the shape of the joints between connected segments of a stroked path.

The default value of this property is 1 (miter).

Usage:

```
aLayer:setLineJoin(2)
```

Example scripts

LipoMeter.lua

```
--[[  LipoMeter.lua
A demo widget displaying the lipo pack voltage and its individual
cells in a bar graph.
Up to 4 cells can be displayed along the pack voltage. Individual
sensor values are
configured in the display settings view.
--]]



-- Setting widget and value parameters
luaWidgetParam = {
    size= {w= 300, h= 230},
    name= "LiPo meter",
    sensorValueFields= {
        {id= "Pack voltage", type = "scalar"},
        {id= "Cell 1", type= "scalar"},
        {id= "Cell 2", type= "scalar"},
        {id= "Cell 3", type= "scalar"},
        {id= "Cell 4", type= "scalar"}
    }
}

local valueCell1, valueCell2, valueCell3, valueCell4
local valuePack
local cellBarSteps = 7
local packBarSteps = 10



function setupSensorValue(instance, steps)
    local data = {}
    data.instance = instance
    data.steps = steps
    if instance then
        data.name = instance:getName()
        data.unit = instance:getUnit()
```

```
        if data.instance:getAlarmEnabled("min") then
            data.minAlarm = instance:getAlarmValue("min")
        end
        if data.instance:getAlarmEnabled("max") then
            data.maxAlarm = instance:getAlarmValue("max")
        end
        data.minDisp, data.maxDisp = instance:getDisplayRange()
        if not data.minDisp then
            data.minDisp = 0
        end
        data.dispRange = data.maxDisp - data.minDisp
    end

    return data
end


function tapHandlerCell1()
    if valueCell1.instance then
        valueCell1.instance:announceValue()
    end
end


function tapHandlerCell2()
    if valueCell2.instance then
        valueCell2.instance:announceValue()
    end
end


function tapHandlerCell3()
    if valueCell3.instance then
        valueCell3.instance:announceValue()
    end
end


function tapHandlerCell4()
```

```
    if valueCell4.instance then
        valueCell4.instance:announceValue()
    end
end


function tapHandlerPack()
    if valuePack.instance then
        valuePack.instance:announceValue()
    end
end


function setupCellBars(frame, barFrame, valueData, title)
    local barView = Box:new(frame)
    barView:setBackgroundColor(self:getColor("clearColor"))

    -- create name label
    if title then
        local lbl = Label:new(barView, {x=5, y=8, w=38, h=15})
        lbl:setFont("Helvetica", 10)
        lbl:setTextAlignment("center")
        lbl:setText(title)
    end

    local barLayer = Layer:new(barView, barView:bounds())
    barLayer:setBackgroundColor(self:getColor("clearColor"))

    --local barFrame = bframe
    -- create bar shape
    local meterBar = Layer:new(barLayer, barFrame);
    meterBar:setBackgroundColor(self:getColor("clearColor"))
    meterBar:beginPath()
        meterBar:moveToPoint({x=barFrame.w/2, y=barFrame.h})
        meterBar:lineToPoint({x=barFrame.w/2, y=0})
    meterBar:applyPath()
    meterBar:setStrokeColor(self:getColor("grayColor"))
    meterBar:setLineWidth(barFrame.w)
```

```
    -- create green shape
    local greenBar = Layer:new(barLayer, barFrame);
    greenBar:setBackgroundColor(self:getColor("clearColor"))
    greenBar:beginPath()
        greenBar:moveToPoint({x=barFrame.w/2, y=barFrame.h})
        greenBar:lineToPoint({x=barFrame.w/2, y=0})
    greenBar:applyPath()
    greenBar:setStrokeColor(self:getColor("greenColor"))
    greenBar:setLineWidth(barFrame.w)
    if not valueData.instance then
        greenBar:setStrokeEnd(0)
    end

    -- create red alarm shape
    local redBar = Layer:new(barLayer, barFrame);
    redBar:setBackgroundColor(self:getColor("clearColor"))
    redBar:beginPath()
        redBar:moveToPoint({x=barFrame.w/2, y=barFrame.h})
        redBar:lineToPoint({x=barFrame.w/2, y=0})
    redBar:applyPath()
    redBar:setStrokeColor(self:getColor("alarmColor"))
    redBar:setLineWidth(barFrame.w)
    local s = 0
    if valueData.minAlarm then
        s = (valueData.minAlarm - valueData.minDisp) /
valueData.dispRange
        s = math.floor((s * valueData.steps) + 0.5) / valueData.steps
    end
    redBar:setStrokeEnd(s)

    -- create the opaque bar
    local valueBar = Layer:new(barLayer, barFrame);
    valueBar:setBackgroundColor(self:getColor("clearColor"))
    valueBar:beginPath()
    valueBar:moveToPoint({x=barFrame.w/2, y=0})
    valueBar:lineToPoint({x=barFrame.w/2, y=barFrame.h})
    valueBar:applyPath()
    local clValBar = self:getColor("whiteColor"); clValBar.a = 0.65
    valueBar:setStrokeColor(clValBar);
```

```
    valueBar:setLineWidth(barFrame.w)
    valueData["valueBar"] = valueBar

    -- create masking shape
    local maskShape = Layer:newMask(barLayer);
    local maskFrame = barFrame; maskFrame.y = maskFrame.y + 12;
maskFrame.h = 4
    maskShape:beginPath()
    for i = 0, valueData.steps, 1 do
        maskShape:rectangle(maskFrame)
        maskFrame.y = maskFrame.y + 16
    end
    maskShape:applyPath(true)



    -- create the label
    local lblValue
    if title then     -- cell value label
        lblValue = Label:new(barView, {x=0, y=150, w=40, h=20})
        lblValue:setFont("lcd", 15)
        lblValue:setTextAlignment("right")
        lblValue:setText"---"
        if not valueData.instance then
            lblValue:setTextColor(self:getColor("grayColor"))
        end
        valueData["lblValue"] = lblValue

    else   -- pack value label
        lblValue = Label:new(barView, {x=0, y=176, w=62, h=32})
        lblValue:setFont("lcd", 26)
        lblValue:setTextAlignment("right")
        lblValue:setText("---")
        valueData["lblValue"] = lblValue

        local lblUnit = Label:new(barView, {x=66, y=183, w=18, h=19})
        lblUnit:setFont("Helvetica", 17)
        lblUnit:setText(valueData.unit)
        if not valueData.instance then
            lblValue:setTextColor(self:getColor("grayColor"))
```

```
            lblUnit:setTextColor(self:getColor("grayColor"))
        end
    end

    return barView
end



function displayValue(valueData)
    if not valueData.instance then
        return
    end

    -- display sensor value
    valueData.lblValue:setText(valueData.instance:getValueStr())

    -- set value label & check alarm
    valueData.lblValue:setAlarmed(valueData.instance:isValueAlarmed())

    -- set bar graph
    local value = valueData.instance:getValue()
    local s = 0
    s = (value - valueData.minDisp) / valueData.dispRange;
    if s > 1 then s = 1 elseif s < 0 then s = 0 end
    s = math.floor((s * valueData.steps) + 0.5) / valueData.steps;
    valueData.valueBar:setStrokeEnd(1 - s)
end



function initialiseWidget(self)
    -- get the sensor values
    valuePack = setupSensorValue(self:sensorValueForField("Pack
voltage"), packBarSteps)
    valueCell1 = setupSensorValue(self:sensorValueForField("Cell 1"),
cellBarSteps)
    valueCell2 = setupSensorValue(self:sensorValueForField("Cell 2"),
cellBarSteps)
    valueCell3 = setupSensorValue(self:sensorValueForField("Cell 3"),
cellBarSteps)
```

```
    valueCell4 = setupSensorValue(self:sensorValueForField("Cell 4"),
cellBarSteps)

    self:setMeterView("lcdPanel")
    local view
    -- create value label
    view = Label:new({x=16, y=13, w=180, h=15})
    view:setFont("Helvetica", 11)
    if valuePack.instance then
        view:setText(valuePack.instance:getName())
    else
        view:setText("<no value set>")
        view:setTextColor(self:getColor("grayColor"))
    end

    -- setup the cell bars
    local boundingFrame
    local barFrame
    -- cell 1
    boundingFrame = {x=12, y=42, w=48, h=177}
    barFrame = {x=9, y=29, w=30, h=109}
    view = setupCellBars(boundingFrame, barFrame, valueCell1, "Cell
1")
    view:addTapHandler(tapHandlerCell1)

    -- cell 2
    boundingFrame.x = boundingFrame.x + boundingFrame.w
    barFrame = {x=9, y=29, w=30, h=109}
    view = setupCellBars(boundingFrame, barFrame, valueCell2, "Cell
2")
    view:addTapHandler(tapHandlerCell2)

    -- cell 3
    boundingFrame.x = boundingFrame.x + boundingFrame.w
    barFrame = {x=9, y=29, w=30, h=109}
    view = setupCellBars(boundingFrame, barFrame, valueCell3, "Cell
3")
    view:addTapHandler(tapHandlerCell3)

    -- cell 4
```

```
    boundingFrame.x = boundingFrame.x + boundingFrame.w
    barFrame = {x=9, y=29, w=30, h=109}
    view = setupCellBars(boundingFrame, barFrame, valueCell4, "Cell
4")
    view:addTapHandler(tapHandlerCell4)

    -- pack
    boundingFrame = {x=206, y=10, w=84, h=210}
    barFrame = {x=11, y=13, w=64, h=158}
    view = setupCellBars(boundingFrame, barFrame, valuePack, nil)
    view:addTapHandler(tapHandlerPack)

end

function updateWidget(self)
    displayValue(valuePack)
    displayValue(valueCell1)
    displayValue(valueCell2)
    displayValue(valueCell3)
    displayValue(valueCell4)
end
```

Change history

Document version	Date	Status	Short description of the modification
1.0	28.04.2021	Release	Initial version