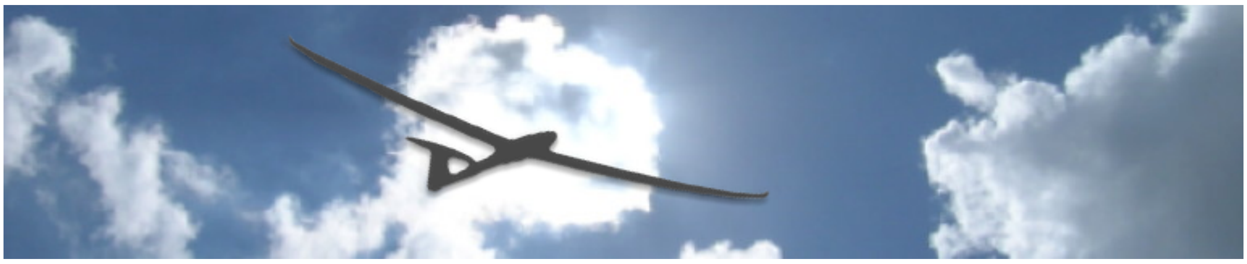

iMSB Lua Sensor Scripting

Reference Guide

Version 1.5

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 libraries user scripts can be attached to dedicated sensor values. 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.

iMSB supports sensor value based scripts. This means that a script can be associated to a calculated sensor value that will be evaluated on a regular basis during data logging. Such a value behaves as any other value. It can thus be visualized, announced and logged by the standard iMSB functionalities.

In order to use Lua scripts the following steps must be performed:

- The "*Expert mode*" must be enabled in the "*Data logging*" section of the application preferences
- A "Script sensor" must be added to your model

Lua scripted values are only available in a script sensor that contains eight values to which a Lua script can be attached. You can of course instantiate several occurrences of this sensor.

Even if Lua is a quite efficient programming language, the use of value 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 function cannot be performed by the existing iMSB capabilities, cannot be substituted by a generic sensor value or be automated with transmitter tasks and events.

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, Lua has been extended with several dedicated iMSB libraries:

- `SensorValue` class library: represents a sensor value object and exposes a set of methods to manipulate that value
- `TransmitterEvent` class library: allows for the handling of trigger and conditional transmitter events

- `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

Lua scripts are directly stored in each scripted sensor values. 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 value settings. 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

Sensor value scripts

Value setup

A scripted value has essentially the same attributes as a measured value. Thus it can have a specific name, a unit, has statistics, alarms, etc., that can be configured as usual in the iMSB application.

One important aspect that has to be specified in the value settings is its data type, as this has to be handled consequently in your script logic. Possible data types are:

- *Numerical* (scalar) value that is always represented by a floating-point number. These values can be displayed with zero, one or two decimal places
- *Time* value, expressed internally in seconds
- *String* value that is essentially a text. Note that two supplemental display fields are available in order to present this type of sensor value
- *Coordinate* value, expressed as a latitude, longitude value tuple

Script anatomy

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

```
function initialiseSensorValue(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 acquiring sensor data.

```
function updateSensorValue(self)
```

This function gets called on each value update cycle, at the same interval as defined in the "*Telemetry data update*" rate you can select in the application settings. This is where value calculations 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.

Value parameters

Within a sensor script there is the possibility to define parameter variables that can be used to control various aspects of the script. Defined value parameters are populated into the value configuration screen where the user may freely define these variables without the need to modify the script code. In order to populate the defined parameters the script must at least be run once.

Value parameters are defined via a global table variable that must be named `luaValueParam` and has the following definition:

```
luaValueParam = {
    {id= <id> [, name= <name>] [, type= <type>], default= <defValue>},
    {id= <id> [, name= <name>] [, type= <type>], default= <defValue>},
     etc…
}
```

Where:

- `id`: (number or string) is a unique variable identification. The `id` is used to retrieve the parameter via the `:getLuaValueParam()` function call
- `name`: (string, optional) the name given to the parameter and that will be displayed in the value configuration view. If omitted the `id` will be used as the parameter name
- `type`: (string, optional) is the type given to the variable (see below for a list of available types). If omitted will be derived from the format of the default value
- `default`: a default, i.e. initial value given to the variable. The value given here must match with the type

Possible values for the type tag are:

- `"float"`: a floating point value
- `"integer"`: an integer value
- `"string"`: a string value freely definable by the user
- `"boolean"`: can be true or false and is shown as a switch to the user
- `"time"`: expressed in seconds (float number), but displayed in time format to the user
- `"sound"`: the user may select a system sound from the picker

Defined value parameters can be accessed via the `SensorValue:getLuaValueParam()` function call (see the SensorValue class definition)

Example:

```
luaValueParam = { {id="Parameter 1", type= "boolean", default= true},
      {id= 2, name= "Alarm message", default= "An alarm occurred"},
      {id= 3, name= "Timeout time", type= "time", default= 60} }
```

The SensorValue class

The `SensorValue` class mostly defines setter and getter methods that allow retrieving or setting specific value information. Within a value script you can read attributes of every sensor value that is available in your model, such as its value, statistics minimum or maximum data, alarm values and configuration, etc. The functions used to perform these tasks are getter methods. Writing data to a value (through setter methods) can obviously only be done to the value the current script is attached to.

The `SensorValue` class follows the pseudo object oriented approach available in Lua. Before you start, you should be familiar with the Lua's OO paradigm and know the basics of Lua object anatomy and method calls.

In order to get any value information, you need to instantiate the sensor value you want to manipulate. Value instantiation is performed via `SensorValue:get(name|(sensorAdr, valueAdr))`, where you specify either the name or a sensor/value address tuple of the value you want to retrieve. Once the value has been instantiated, you can call `SensorValue` methods on that value.

Important to know is that setter methods can only be called on the value your script is attached to or to another script value. Thus, you cannot modify values that is not directly handled by a script. An instance of that value does not need to be retrieved specifically. It has already been instantiated for you and is provided as the convenience argument `self` of both mandatory script functions.

Retrieving a sensor value can be done in several fashions:

- Referencing by ***value name***: as long as a value name is unique its string representation can be used to access its data. This method is however not very robust, as a name can be edited, which would require to change all references in your code
- Referencing through the ***sensor address*** and ***value address***: in iMSB each sensor and value form a unique address tuple, whether it is a measured or internally calculated value. You can always reference a value with a sensor address/value address tuple. In expert mode all sensor and value addresses can be looked up in the corresponding sensor and value setup screen. The following have to be considered however:

  iMSB – Mlink (Multiplex): in the Multiplex version a sensor does not have a dedicated address and thus must always be 0

  iMSB – FrSky: sensor addresses are displayed in decimal, whereas value addresses are provided in hexadecimal notation

  iMSB – HoTT (Graupner): sensor addresses are provided in hexadecimal, whereas value addresses are displayed in decimal notation

  iMSB – FASSTest (Futaba): a sensor does not have a dedicated address and must always be 0. The value address corresponds to the Slot Id of the value

The TransmitterEvent class

The `TransmitterEvent` class allows instantiation of iMSB trigger and conditional transmitter events, such as GPIO switch events, sensor value transitions and timer triggers. When instantiating a trigger event you need to specify a callback function that will be called whenever the trigger conditions are met. All trigger-based events that are available in the application can be accessed through this library.

Additionally GPIO switch conditions and Multiplex transmitter channel conditions can be specified and tested. Note that the Lua interface does only support testing of external events, as value or timer conditions can easily be programmed with the `SensorValue` class. It is also left to the programmer to combine the various trigger and conditional events into the required task.

The `TransmitterEvent` class follows the pseudo object oriented approach available in Lua.

The imsb library

The `imsb` library provides a set of functions that can be used within your script to control various aspects of the application that are not directly bound to a sensor or sensor value.

Currently three broad function categories are available:

- Notification and announcement functions: handle value announcements, custom notifications and event logging

- Application navigation functions: allow to automate application navigation and controlling app settings
- Miscellaneous functionsThe device library

The device library

The `device` library grants access to two kind of sensors available in your iOS device that can be used in any lua script.

Currently two sensors are made available via the device library:

- The internal GPS sensor, which provides the current, i.e. the viewer position, its altitude and orientation of the device relative to the north
- The motion (IMU) sensor, allowing retrieving positional attitude of the device, as well as motion rate

Note that these sensors only provide information about the iOS device (iPhone or iPad) position or attitude, and are not related in any way to the flying model.

iMSB Lua function reference

The SensorValue class

The TransmitterEvent class

`TransmitterEvent:newValueTransition(callbackFct, sensorValue, “above”|”below”|”both”, thresholdValue)`

`TransmitterEvent:newRepeatingTimerTrigger(callbackFct, timeInterval)`

`TransmitterEvent:newUniqueTimerTrigger(callbackFct, timeInterval)`

`TransmitterEvent:newGPIOSwitchCondition(“up”|”down”, portNum)`

`TransmitterEvent:newChannelCondition(“up”|”down”, channelNum)`

`TransmitterEvent:newControlCondition(callbackFct, “up”|”middle”|”down”, “on”|"off"", switchName)`

`:testCondition()`

`:setIsActive(onOff)`

`:getIsActive()`

`:reset()`

The imsb Library

`imsb.announceValue(value)`

`imsb.announceValueStat(value [,”max”|”min”|”avg”])`

`imsb.announceAlarms()`

`imsb.announceMessage(message)`

`imsb.announceView([viewName])`

`imsb.playSound(soundName)`

`imsb.displayNotification(type, notificationText)`

`imsb.writeLogEvent(type, title[, text])`

`imsb.showView(viewName)`

`imsb.showNextView()`

`imsb.showPreviousView()`

`imsb.varioPlayer(onOff)`

`imsb.voiceAnnouncements(onOff)`

`imsb.speechRecognition(onOff)`

`imsb.elapsedTime()`

`imsb.dataUpdateRate()`

`imsb.dataReaderMode()`

`imsb.stopLogger([stayOnView])`

The device Library

`device.startLocationUpdates(doOrientation [, callbackFct])`

`device.stopLocationUpdates()`

`device.getPosition()`

`device.getAltitude()`

`device.getOrientation()`

`device.startMotionUpdates(updateRate, resetAttitude [, callbackFct])`

`device.stopMotionUpdates()`

device.getAttitude()
device.getRotationRate()

The SensorValue class

The `SensorValue` class mostly defines setter and getter methods that allow retrieving or setting specific value information.

In order to get any value information, you need to instantiate the sensor value you want to manipulate. Value instantiation is performed via `SensorValue:get(name|(sensorAdr, valueAdr))`, where you specify either the name or a sensor/value address tuple of the value you want to retrieve. Once the value has been instantiated, you can call `SensorValue` methods on that value.

Important to know is that setter methods can only be called on the value the script is attached to. Thus, you cannot modify values that are not associated to your script. An instance of that value does not need to be retrieved specifically. It has already been instantiated for you and is provided as the convenience argument `self` of both mandatory script functions.

`SensorValue:get(name|(sensorAdr, valueAdr))`

Retrieves an instance of the iMSB sensor value specified by the arguments.

Parameters:
- `name`: (string) the name of the sensor to get the value from
- `sensorAdr, valueAdr`: (number) the sensor and value address that uniquely identifies the sensor

Returns:
- `instance`: (userdata) the instance of the sensor value
- `null` if the value does not exist

Discussion:
Before manipulating a sensor value, it must always be instantiated first.
A sensor value can be accessed either by its name or by specifying the sensor address and value address. For Multiplex M-Link and Futaba FASSTest the sensor address always has to be `0`.

Usage:
```
local aSensVal
aSensVal = SensorValue:get(“Receiver voltage”)
aSensVal = SensorValue:get(0, 6)          -- typically Multiplex
aSensVal = SensorValue:get(0x8A, 12)      -- typically HoTT
aSensVal = SensorValue:get(12, 0x0f30)    -- typically FrSky
aSensVal = SensorValue:get(0, 24)         -- typically FASSTest
```

`:setValue(val|{lat=, lon=})`

Sets the value of the actual sensor (`self`) or another script value.

Parameters:
- `val`: (number, string) any numerical or string value
- `{lat=, lon=}`: (table) in case of coordinate value a table with latitude and longitude coordinate (expressed in decimal degrees)

Returns:
- `n/a`

Discussion:
This method can only set the value the script is attached to or another script value.

If the `val` parameter is set to `null` the corresponding sensor value will have an undefined value.
Values are always expressed in the metric system, independently of the measurement system that was chosen in the application settings. Conversion into the nautical or imperial system is performed automatically by the application.

Usage:
```
self:setValue(12.4)
self:setValue({lat=46.50422833, lon=6.68691333})
```

:getValue()

Gets the value of the specified sensor.

Parameters:
- `n/a`

Returns:
- `val`: (number, string) the sensor scalar or string value (depending of its type)
- `{lat=, lon=}`: (table) a table containing latitude and longitude coordinate in case of a coordinate value (expressed in decimal degrees)
- `0` if the value is undefined

Discussion:
This method can be called on `self` or any value previously instantiated.

Usage:
```
v = self:getValue()
```

:getValueStr()

A formatted string representation of the sensor value.

Parameters:
- `n/a`

Returns:
- `str`: (string) the string representation of the value
- `nil`:if the value is undefined

Discussion:
This method can be called on `self` or any value previously instantiated.

Usage:
```
v = aSensVal:getValueStr()
```

:isValueDefined()

Informs whether the value has a defined state.

Parameters:
- `n/a`

Returns:
- `defined`: (boolean) indicates if the value is defined

Discussion:
This method can be called on `self` or any value previously instantiated.

When values have not yet been acquired, they typically do not have a defined state. In the iMSB display views they are displayed as "---".

Usage:
```
defined = self:isValueDefined()
defined = aSensVal:isValueDefined()
```

:getName()

Gets the name of the specified sensor value.

Parameters:
- `n/a`

Returns:
- `name`: (string) the sensor value name

Discussion:
This method can be called on `self` or value previously instantiated.

Usage:
```
name = self:getName()
name = aSensVal:getName()
```

:getAddress()

Gets the s.port sensor id and value address of the associated value.

Parameters:
- `n/a`

Returns:
- `sensId`: (number) the id (address) of the sensor the value belongs to
- `addr`: (number) the address of the sensor value

Discussion:
This method can be called on `self` or value previously instantiated.

Usage:
```
sensId, addr = self:getAddress()
sensId, addr = aSensVal:getAddress()
```

:setUnit(unit)

Sets the unit of the actual sensor.

Parameters:
- `unit`: (string) the string representation of the unit

Returns:
- `n/a`

Discussion:
A specific unit can only be set on a value whose unit was declared as a “custom unit”.
This method can only set the value the script is attached to or another script value.

Usage:
```
self.setUnit(“mAh”)
```

`:getUnit()`

Retrieves the unit string representation of a specified sensor value.

Parameters:
- `n/a`

Returns:
- `unit`: (string) unit string representation
- `null` if the unit is undefined

Discussion:
This method can be called on `self` or any value previously instantiated.

Usage:
```
unit = self:getUnit()
unit = aSensVal:getUnit()
```

`:getFlightStats()`

Returns the min, max and average statistics of the current flight session.

Parameters:
- `n/a`

Returns:
- `min, max, avg`: (number) min, max and average statistics value

Discussion:
This method can be called on `self` or any value previously instantiated.
This method always returns a min, max and avg triple.
If a statistic does not exist or if it is undefined, its value is set to `null`.

Usage:
```
min, max, avg = self:getFlightStats()
min, max, avg = aSensVal:getFlightStats()
```

`:getModelStats()`

Returns the min, max and average model statistics of a specified sensor value.

Parameters:
- `n/a`

Returns:
- `min, max, avg`: (number) min, max and average statistics value

Discussion:
This method always returns a min, max and avg triple.
If a statistic does not exist or if it is undefined, its value is set to `null`.

Usage:
```
min, max, avg = self:getModelStats()
min, max, avg = aSensVal:getModelStats()
```

`:setAlarmValue(val [,”min”|”max”])`

Sets the alarm threshold of the actual value.

Parameters:
- `val`: (number) the alarm threshold
- `“min”` or `“max”`: (string, optional) specifies whether the min or max alarm should be set. Can be omitted if only one alarm is defined for the value

Returns:
- `n/a`

Discussion:
This method can only set the alarm threshold of the value the script is attached to or another script value.

Usage:
```
self:setAlarmValue(45.6, “max”)
```

`:getAlarmValue([”min”|”max”])`

Returns the alarm threshold of a specified sensor value.

Parameters:
- `“min”` or `“max”`: (string, optional) specifies whether the min or max alarm should be returned. Can be omitted if only one alarm is defined for the value

Returns:
- `val`: (number) the value of the alarm value

Discussion:
This method can be called on `self` or any value previously instantiated.
If an alarm does not exist or if it is undefined, its value is set to `null`.

Usage:
```
val = self:getAlarmValue(“min”)
val = aSensVal:getAlarmValue()
val = sSensVal:getAlarmValue(“max”)
```

`:setAlarmMessage(msg [,”min”|”max”])`

Sets the alarm occurrence announcement message.

Parameters:
- `msg`: (string) the announcement message
- `“min”` or `“max”`: (string, optional) specifies whether the min or max alarm message should be set. Can be omitted if only one alarm is defined for the value

Returns:
- `n/a`

Discussion:
This method can only set the alarm occurrence message of the value the script is attached to or another script value.

Usage:
```
self:setAlarmMessage(“Value exceeded $VALUE ALARM$”, “max”)
```

`:getAlarmMessage(["min"|"max"])`

Returns the alarm occurrence announcement message of a specified sensor value.

Parameters:
- `"min"` or `"max"`: (string, optional) specifies whether the min or max alarm message should be returned. Can be omitted if only one alarm is defined for the value

Returns:
- `msg`: (string) the announcement message of the alarm.

Discussion:
This method can be called on `self` or any value previously instantiated.
If an alarm message does not exist its value is set to `null`.

Usage:
```
msg = self:getAlarmMessage("min")
msg = aSensVal:getAlarmMessage()
msg = aSensVal:getAlarmMessage("max")
```

`:setAlarmMessageClearance(msg [,"min"|"max"])`

Sets the alarm clearance announcement message.

Parameters:
- `msg`: (string) the alarm clearance announcement message
- `"min"` or `"max"`: (string, optional) specifies whether the min or max alarm message should be set. Can be omitted if only one alarm is defined for the value

Returns:
- `n/a`

Discussion:
This method can only set the alarm clearance message of the value the script is attached to or another script value.

Usage:
```
self:setAlarmMessageClearance("Alarm has been cleared", "max")
```

`:getAlarmMessageClearance(["min"|"max"])`

Returns the alarm clearance announcement message of a specified sensor value.

Parameters:
- `"min"` or `"max"`: (string, optional) specifies whether the min or max alarm message should be returned. Can be omitted if only one alarm is defined for the value

Returns:
- `msg`: (string) the alarm clearance message of the alarm.

Discussion:
This method can be called on `self` or any value previously instantiated.
If an alarm message does not its value is set to `null.`

Usage:
```
msg = self:getAlarmMessageClearance("min")
msg = aSensVal:getAlarmMessageClearance()
msg = aSensVal:getAlarmMessageClearance("max")
```

`:setAlarmEnabled(enable [,”min”|”max”])`

Enables or disables the alarm of the actual sensor.

Parameters:
- `enable`: (boolean) set to `true` or `false` to enable or disable the alarm respectively
- `“min”` or `“max”`: (string, optional) specifies whether the min or max alarm message should be enabled or disabled. Can be omitted if only one alarm is defined for the value

Returns:
- `n/a`

Discussion:
This method can only enable or disable the alarm of the value the script is attached to, or another script value.

Usage:
```
self:setAlarmEnabled(true, “max”)
```

`:getAlarmEnabled([min”|”max”])`

Informs if the alarm of a given sensor value is enabled or disabled.

Parameters:
- `“min”` or `“max”`: (string, optional) specifies whether the min or max alarm should be considered. Can be omitted if only one alarm is defined for the value

Returns:
- `enabled`: (boolean) alarm enabled or disabled flag

Discussion:
This method can be called on `self` or any value previously instantiated.
If an alarm does not its value is set to `null`.

Usage:
```
enabled = self:getAlarmEnabled(“min”)
enabled = aSensVal:getAlarmEnabled()
```

`:forceValueAlarmed(enable [,alarmMessage])`

Force a value to go into or exit from an alarm status. The alarm status can be set irrespective from alarm configuration of the actual value.

Parameters:
- `enable`: (boolean) set to `true` or `false` to start or stop the alarm respectively
- `alarmMessage`: (string, optional) an optional alarm message that is to be announced. If not specified, no alarm notifications will be performed

Returns:
- `n/a`

Discussion:
This method is typically used when specific non-standard alarm calculation is required. Manually starting or stopping an alarm requires that the corresponding alarm has been previously set to disabled. Thus forcing an alarm will have no effect if `:setAlarmEnabled(false)` with argument `false` has not been set previously.

When an `alarmMessage` is defined all alarm notifications (voice announcements and notification hud) are issued automatically. Additionally, an entry in the flight log is also generated. If no `alarmMessage` is specified none of the above is performed. Individual notification or log entry must then be performed explicitly via the `imsb.announceMessage()`, `imsb.displayNotification()` and `imsb.writeLogEvent()` functions.
This function can only be called on the value the script is attached to or another script value.

Usage:
```
self:forceValueAlarmed(true, “this value is alarmed”)
```

:isValueAlarmed([”min”|”max”])

Informs whether the referenced value has currently an alarm.

Parameters:
- `“min”` or `“max”`: (string, optional) specifies whether the min or max alarm should be considered. Can be omitted if only one alarm is defined for the value

Returns:
- `alarmed`: (boolean) true if the value has an active alarm

Discussion:
This method can be called on `self` or any value previously instantiated.
If an alarm does not exist its value is set to `null.`

Usage:
```
alarmed = self:isValueAlarmed(“min”)
alarmed = aSensVal:isValueAlarmed()
alarmed = aSensVal:isValueAlarmed(“max”)
```

:setLevelNotificationValue(step, threshold [,”above”|”below”])

Sets the level notification step and threshold of the actual value.

Parameters:
- `step`: (number) the level notification step value
- `threshold`: (number) the level notification threshold
- `“above”` or `“below”`: (string, optional) specifies whether the above level or below level notification should be set. Can be omitted if only one level notification is defined for the value

Returns:
- `n/a`

Discussion:
This method can only set the level notification step and threshold of the value the script is attached to or another script value.

Usage:
```
self:setLevelNotificationValue(20, 100 “above”)
```

:getLevelNotificationValue([”above”|”below”])

Returns the level notification step and threshold values of a specified sensor value.

Parameters:

- `“above”` or `“below”`: (string, optional) specifies whether the above level or below level notification should be retrieved. Can be omitted if only one level notification is defined for the value

Returns:
- `step`: (number) the step value of the level notification
- `threshold`: (number) the threshold value of the level notification

Discussion:
This method can be called on `self` or any value previously instantiated.
If the level notification does not exist or if it is undefined, its value is set to `null`.

Usage:
```
step, threshold = self:getLevelNotificationValue(“above”)
step, threshold = aSensVal:getLevelNotificationValue()
step, threshold = aSensVal:getLevelNotificationValue(“below”)
```

:setLevelNotificationMessage(aboveMsg, belowMsg [,”above”|” below”])

Sets the level notification below threshold and above threshold notification messages of the actual value.

Parameters:
- `aboveMsg`: (string) the above level notification message
- `belowMsg`: (string) the below level notification message
- `“above”` or `“below”`: (string, optional) specifies whether the above level or below level notification should be set. Can be omitted if only one level notification is defined for the value

Returns:
- `n/a`

Discussion:
This method can only set the level notification messages of the value the script is attached to or another script value.

Usage:
```
 self:setLevelNotificationMessage(“Value is above $VALUE ALARM$”,
“Value is below $VALUE ALARM$”, “above”)
```

:getLevelNotificationMessage([above”|”below”])

Returns the level notification below threshold and above threshold notification messages of a specified sensor value.

Parameters:
- `“above”` or `“below”`: (string, optional) specifies whether the above level or below level notification should be retrieved. Can be omitted if only one level notification is defined for the value

Returns:
- `aboveMsg`: (string) the above level notification message
- `belowMsg`: (string) the below level notification message

Discussion:
This method can be called on `self` or any value previously instantiated.

If the level notification message does not exist or if it is undefined, its value is set to `null.`

Usage:
```
aboveMsg, belowMsg = self:getLevelNotificationMessage(“above”)
aboveMsg, belowMsg = aSensVal:getLevelNotificationMessage()
aboveMsg, belowMsg = aSensVal:getLevelNotificationMessage(“below”)
```

:setLevelNotificationEnabled(enable [,”above”|”below”])

Enables or disables the level notification of the actual sensor.

Parameters:
- `enable`: (boolean) set to `true` or `false` to enable or disable the level notification
- `“above”` or `“below”`: (string, optional) specifies whether the above level or below level notification should be set. Can be omitted if only one level notification is defined for the value

Returns:
- `n/a`

Discussion:
This method can only be called on the value the script is attached to or another script value.

Usage:
```
self:setLevelNotificationEnabled(true, “below”)
```

:getLevelNotificationEnabled([”above”|”below”])

Informs if the level notification of a given sensor value is enabled or disabled.

Parameters:
- `“above”` or `“below”`: (string, optional) specifies whether the above level or below level notification should be retrieved. Can be omitted if only one level notification is defined for the value

Returns:
- `enabled`: (boolean) level notification enabled or disabled flag

Discussion:
This method can be called on `self` or any value previously instantiated.
If the level notification does not exist its value is set to `null.`

Usage:
```
enabled = self:getLevelNotificationEnabled(“above”)
enabled = aSensVal:getLevelNotificationEnabled()
enabled = aSensVal:getLevelNotificationEnabled(“below”)
```

:forceValueNotification(type ,notificationMessage [,soundName])

Issues a value notification irrespective from level notification configuration of the actual value.

Parameters:
- `type`: (number) specifies which hud type, i.e. notification icon is displayed in the notification hud and event log
- `notificationMessage`: (string) a notification message that is to be announced or displayed
- `soundName`: (string, optional) the name of the sound to play

Returns:
- `n/a`

Discussion:
This method allows to issue manually a notification message, irrespective of the value notification settings. When this method is called all notifications (voice announcements and notification hud) are issued automatically depending of the user settings. Additionally, an entry in the flight log is also generated.
You would typically use this method as a generic replacement for individual `imsb.announceMessage()`, `imsb.displayNotification()` and `imsb.writeLogEvent()` calls.

The following notification types are available and can be set with the type parameter:
- `1` Information
- `2` Alarm occurrence
- `3` Alarm clearance
- `4` Above level notification
- `5` Below level notification
- `6` Changed notification
- `7` Turn notification
- `8` Reached goal notification

This method can only be called on the value the script is attached to or another script value.

Usage:
```
self:forceValueNotification(1, “Minimum voltage reached. It’s time to land!”)
```

:getDisplayRange()

Returns the lower and upper value of a sensor value’s display range. Display range can be set by the user in the value configuration screen.

Parameters:
- `n/a`

Returns:
- `min`: (number) the minimum value of the display range
- `max`: (number) the maximum value of the display range

Discussion:
This method can be called on `self` or any value previously instantiated.

Usage:
```
min, max = self:getDisplayRange()
min, max = aSensVal:getDisplayRange()
```

:announceValue()

Announces a given sensor value. The standard notification message is used by this method.

Parameters:
- `n/a`

Returns:
- `n/a`

Discussion:
This method can be called on `self` or any value previously instantiated.

Usage:
```
self:announceValue()
aSensValue:announceValue()
```

`:announceValueStat(”max”|”min”|”avg”)`

Announces a flight statistic value of a given sensor value. The standard notification message is used by this method.

Parameters:
- `“max”, “min”` or `“avg”`: (string) specifies whether the min, max or average statistic value should be announced. Can be omitted if only one statistic is defined for the value

Returns:
- `n/a`

Discussion:
This method can be called on `self` or any value previously instantiated.

Usage:
```
self:announceValueStat(“min”)
aSensValue:announceValueStat()
aSensValue:announceValueStat(“avg”)
```

`:getLuaValueParam(paramId)`

Returns the value of a specified value parameter.

Parameters:
- `paramId`: (integer or string) specifies the `id` of the parameter value to be returned

Returns:
- `val`: the value of the requested value parameter. The returned type depends on the parameter definition

Discussion:
If the parameter value does not exist its value is set to `null`.
In order to access value parameter values they must have been defined in a global `luaValueParam` variable in the body of the script. For further information refer to the "Value parameters" section.

Usage:
```
val = self:getValueParam(“Timeout time”)
```

The TransmitterEvent class

The `TransmitterEvent` class allows the instantiation of iMSB trigger and conditional transmitter events, such as GPIO switch events, sensor value transitions and timer triggers. Additionally GPIO switch conditions and Multiplex transmitter channel conditions can be specified and tested.

A trigger (called a transition) or a conditional event must be instantiated via one of the dedicated constructor methods that start with new. For each transition event you also need to provide the reference to a callback function that will be called whenever the trigger conditions are met. For instance the call `TransmitterEvent:newGPIOSwitchCondition(myCallbackFunc, “up”, 0)` registers a GPIO switch transition observer that will call a function names `myCallbackFunc` when a switch on GPIO port 0 gets flipped into the `“up”` position.

If you want to gain access to a conditional event, i.e. the state of a switch for instance you also need to instantiate such conditional event, which will allow you to subsequently test the status of the condition. For instance the call to `TransmitterEvent:newGPIOSwitchCondition(“up”, 0)`. Will register GPIO switch condition on port 0 and return a reference to this condition. A call to `:testCondition`() will then tell you if the corresponding switch is on or off.

`TransmitterEvent:newGPIOSwitchTransition(callbackFct, “up”|” down”|”toggle”, portNum)`

Registers a new GPIO switch transition event. GPIO switch transitions are fired by physical switches that can be associated with one of the eight GPIO ports of the iMSB Smart module. Whenever the event conditions are met a user defined callback function is called.

Parameters:
- `callbackFct`: (function) the reference to a callback function in you code that will be called when the transition conditions are met
- `“up”`, `“down”` or `“toggle”`: (string) the direction of the transition to be tested. `“toggle”` reacts whenever the switch state changes
- `portNum`: (number) the GPIO port number on which you switch available. Port numbers range from 0 to 7.

Returns:
- `instance`: (userdata) the instance of the transmitter event

Discussion:
You should typically use this function if you need to detect a transition on a GPIO switch, as this is much more efficient than regularly polling the value of the switch.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.
If an event function is no more required or is to be temporarily disabled you do such through the `:setIsActive()` method.

Usage:
```
function myCallback()
  print(“myCallback got fired”)
end
event = TransmitterEvent:newGPIOSwitchTransition(myCallback, “up”, 0)
```

M-Link: **`TransmitterEvent:newChannelTransition(callbackFct, “up”|”down”|”toggle”, channelNum)`**
FASSTest: **`TransmitterEvent:newChannelTransition(callbackFct, “enter”|”exit”|”toggle”, channelNum, rangeLow, rangeHigh)`**
HoTT: **`TransmitterEvent:newChannelTransition(callbackFct, “up”|”down”|”toggle”, switchNum, "physical"|"control"|"logical")`**

Registers a new transmitter channel or switch transition event. Channel transitions are fired by modification of a channel value on Multiplex M-Link or Futaba FASSTest transmitters. Switch events are defined in the Graupner HoTT case and respond on actions on physical, control or logical switches. Whenever the event conditions are met a user defined callback function is called.

Depending on the transmitter system used the call to this function is adapted to its specific feature.

Parameters:

- `callbackFct`: (function) the reference to a callback function in you code that will be called when the transition conditions are met

M-Link:

- “`up`”, “`down`” or “`toggle`”: (string) the direction of the transition to be tested. “toggle” reacts whenever the channel state changes
- `channelNum`: (number) the channel number on which the transition is to be assessed. Channels range from 1 to 16.

FASSTest:

- `“enter”`, `“exit”` or `“toggle”`: (string) defines whether the event is fired when the channel value enters or exists the specified range. `“toggle”` reacts in both cases
- `channelNum`: (number) the channel number on which the transition is to be assessed. Channels range from 1 to 18.
- `rangeLow`: (number) the lower bound of the channel value to watch, expressed in %.
- `rangeHigh`: (number) the upper bound of the channel value to watch, expressed in %.

HoTT:

- `“up”`, `“down”` or `“toggle”`: (string) the direction of the switch transition to be tested. `“toggle”` reacts whenever the switch state changes
- `switchlNum`: (number) the switch number that must be monitored. Up to 32 physical switches and 16 control and logical switches can be monitored
- `“physical”`, `“control”` or `“logical”`: (string) specifies if switchNum corresponds to a physical, control or logical switch

Returns:

- `instance`: (userdata) the instance of the transmitter event

Discussion:

You should typically use this function if you need to detect a channel transition, as this is much more efficient than regularly polling its value.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.
If an event function is no more required or is to be temporarily disabled you do such through the `:setIsActive()` method.

Usage:

```
function myCallback()
  print(“myCallback got fired”)
end
```

```
 event = TransmitterEvent:newChannelTransition(myCallback, “down”, 16)
```

HoTT: **`TransmitterEvent:newControlTransition(callbackFct, “up”|”middle”|”down”, “on”|"off"|"both", switchName)`**

This function is only available on Graupner MZ16/32 transmitters.

Registers a new transmitter control transition event. Control transitions are fired by modification of a physical switch position on the MZ16/32 transmitter. Whenever the event conditions are met a user defined callback function is called.

Depending on the transmitter system used the call to this function is adapted to its specific feature.

Parameters:
- `callbackFct`: (function) the reference to a callback function in you code that will be called when the transition conditions are met
- `“up”`, `“middle”` or `“down”`: (string) the position of the switch control transition to be tested.
- `“on”`, `“off”` or `“both”`: (string) the switch transition that triggers the event
- `switchlName`: (string) the switch name that must be monitored.

Returns:
- `instance`: (userdata) the instance of the transmitter event

Discussion:
You should typically use this function if you need to detect a control transition, as this is much more efficient than regularly polling its value.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.
If an event function is no more required or is to be temporarily disabled you do such through the `:setIsActive()` method.

Usage:
```
 function myCallback()
   print(“myCallback got fired”)
 end
 event = TransmitterEvent:newControlTransition(myCallback, “up”, “on”,
“SW1”)
```

`TransmitterEvent:newValueTransition(callbackFct, sensorValue, “above”|”below”|”both”, thresholdValue)`

Registers a new transmitter sensor value transition event. Sensor value transitions are fired when the specified sensor value crosses a set threshold value. Whenever the event conditions are met a user defined callback function is called.

Parameters:
- `callbackFct`: (function) the reference to a callback function in you code that will be called when the transition conditions are met
- `sensorValue`: (userdata) a reference to a `SensorValue` class instance that is obeserved
- `“above”, “below”` or `“both”`: (string) indicates whether the trigger fires when the value is above or below the threshold level. `“both”` triggers in both directions
- `channelNum`: (number) the threshold value firing the event.

Returns:

- `instance`: (userdata) the instance of the transmitter event

Discussion:

You should typically use this function if you need to detect a sensor value transition, as this is much more efficient than regularly polling its value.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.
If an event function is no more required or is to be temporarily disabled you do such through the `:setIsActive()` method.

Usage:

```
function myCallback()
  print(“myCallback got fired”)
end
sensValue = SensorValue:get(“Distance 3D”)
event = TransmitterEvent:newChannelTransition(myCallback, sensValue,
“above”, 50)
```

`TransmitterEvent:newRepeatingTimerTrigger(callbackFct, timeInterval)`

Registers a new repeating timer transition event. Timer transitions are fired at each specified time interval. Whenever the event conditions are met a user defined callback function is called.

Parameters:

- `callbackFct`: (function) the reference to a callback function in you code that will be called when the transition conditions are met
- `timeInterval`: (number) the repeating time interval, expressed in seconds, at which the event will fire

Returns:

- `instance`: (userdata) the instance of the transmitter event

Discussion:

You should typically use this function if you want to execute tasks at repeating time intervals.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.
If an event function is no more required or is to be temporarily disabled you do such through the `:setIsActive()` method.

Usage:

```
function myCallback()
  print(“myCallback got fired”)
end
event = TransmitterEvent:newRepeatingTimerTrigger(myCallback, 10.5)
```

`TransmitterEvent:newUniqueTimerTrigger(callbackFct, timeInterval)`

Registers a new unique timer transition event. Timer transitions are fired once after the elapse of the specified time interval. Whenever the event conditions are met a user defined callback function is called.

Parameters:

- `callbackFct`: (function) the reference to a callback function in you code that will be called when the transition conditions are met
- `timeInterval`: (number) the time interval, expressed in seconds, at which the event will fire

Returns:
- `instance`: (userdata) the instance of the transmitter event

Discussion:
You should typically use this function if you want to execute a task after a certain time intervals. This event does fire only once. You have however to ability to reactivate the time through the `:reset()` method.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.

Usage:
```
function myCallback()
  print(“myCallback got fired”)
end
event = TransmitterEvent:newUniqueTimerTrigger(myCallback, 15)
```

`TransmitterEvent:newGPIOSwitchCondition(“up”|”down”, portNum)`

Registers a new GPIO switch condition event and returns an instance to this event. A GPIO Switch is a physical switch that can be associated with one of the eight GPIO ports of the iMSB Smart module. After instantiation, the value (on/off) of the specified switch can be accesses.

Parameters:
- `“up”` or `“down”`: (string) the direction of the condition to be tested
- `portNum`: (number) the GPIO port number on which you switch available. Port numbers range from 0 to 7.

Returns:
- `instance`: (userdata) the instance of the transmitter event

Discussion:
A call to this function must be performed for each GPIO port you want to observe. Its returned instance will only access that specified port. You have to ability to access the value of the switch through the `:testCondition()` method.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.

Usage:
```
event = TransmitterEvent:newGPIOSwitchCondition(“up”, 0)
```

M-Link: `TransmitterEvent:newChannelCondition(“up”|”down”, channelNum)`
FASSTest: `TransmitterEvent:newChannelCondition(“in”|”out”, channelNum, rangeLow, rangeHigh)`
HoTT: `TransmitterEvent:newChannelCondition(“up”|”down”, switchNum, "physical"|"control"|"logical")`

Registers a new transmitter channel or switch condition event and returns an instance to this event. Channel transitions met by modification of a channel value on Multiplex M-Link or Futaba FASSTest transmitters. Switch events are defined in the Graupner HoTT case and respond on actions on physical, control or logical switches. After instantiation, the value (on/off) of the specified switch can be accesses.

Depending on the transmitter system used the call to this function is adapted to its specific feature.

Parameters:

M-Link:

- `“up”` or `“down”`: (string) the direction of the condition to be tested
- `channelNum`: (number) the channel number to be observed. Channels range from 1 to 16.

FASSTest:

- `“in”` or `“out”`: (string) defines whether the event is true when the channel value lies within or outside the specified range
- `channelNum`: (number) the channel number to be observed. Channels range from 1 to 18.
- `rangeLow`: (number) the lower bound of the channel value to watch, expressed in %.
- `rangeHigh`: (number) the upper bound of the channel value to watch, expressed in %.

HoTT:

- `“up”` or `“down”`: (string) the direction of the condition to be tested
- `switchlNum`: (number) the switch number that must be monitored. Up to 32 physical switches and 16 control and logical switches can be monitored
- `“physical”`, `“control”` or `“logical”`: (string) specifies if switchNum corresponds to a physical, control or logical switch

Returns:

- `instance`: (userdata) the instance of the transmitter event

Discussion:

A call to this function must be performed for each transmitter channel you want to observe. Its returned instance will only access that specified port. You have to ability to access the value of the channel through the `:testCondition()` method.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.

Usage:

```
event = TransmitterEvent:newChannelCondition(“down”, 16)
```

HoTT: **`TransmitterEvent:newControlCondition(callbackFct, “up”|” middle”|”down”, “on”|"off"", switchName)`**

This function is only available on Graupner MZ16/32 transmitters.

Registers a new transmitter control condition event and returns an instance to this event. Control conditions are fired by modification of a physical switch position on the MZ16/32 transmitter. Whenever the event conditions are met a user defined callback function is called.

Depending on the transmitter system used the call to this function is adapted to its specific feature.

Parameters:

- `callbackFct`: (function) the reference to a callback function in you code that will be called when the transition conditions are met
- `“up”`, `“middle”` or `“down”`: (string) the position of the switch control condition to be tested.
- `“on” or “off”`: (string) the switch condition that triggers the event
- `switchlName`: (string) the switch name that must be monitored.

Returns:
- `instance`: (userdata) the instance of the transmitter event

Discussion:
A call to this function must be performed for each transmitter channel you want to observe. Its returned instance will only access that specified port. You have to ability to access the value of the channel through the `:testCondition()` method.
Note that transmitter events must be instantiated in the initialisation function of you script. Instantiating an event in the update function will have no effect.

Usage:
```
 function myCallback()
   print(“myCallback got fired”)
 end
 event = TransmitterEvent:newControlCondition(myCallback, “up”, “on”,
“SW1”)
```

`:testCondition()`

Returns the condition on/off, i.e. true/false of a transmitter event. This method is typically used on conditional events.

Parameters:
- `n/a`

Returns:
- `condition`: (boolean) returns the true or false condition of the event

Discussion:
Note that event conditions are only available once logging has started. Thus this method should only be used in the update function of your script.

Usage:
```
isOn = event:testCondition()
```

`:setIsActive(onOff)`

Allows to activate or deactivate a transmitter event. Once a trigger transmitter event is deactivated its callback method will not be fired anymore.

Parameters:
- `onOff`: (boolean) specifies if the event is active (`true`) or deactivated (`false`)

Returns:
- `n/a`

Usage:
```
event:setIsActive(false)
```

`:getIsActive()`

Informs whether an event is active or deactivated.

Parameters:
- `n/a`

Returns:

- `isActive`: (boolean) informs if the event is activated (`true`) or deactivated (`false`)

Usage:
```
isActive = event:getIsActive()
```

:reset()

Reinitialises an event. This method is particularly useful to reactivate for instance a unique timer trigger.

Parameters:
- n/a

Returns:
- n/a

Discussion:
Except for timer triggers, this method does not have a particular meaning.

Usage:
```
event:reset()
```

The imsb library

The `imsb` library provides a set of functions that can be used within your script to control various aspects of the application. Currently three generic categories are defined:

- Notification and announcement functions
- Application navigation functions
- Miscellaneous functions

i. Notification and announcements

`imsb.announceValue(value)`

Announces a given sensor value. The standard notification message is used by this function.

Parameters:
- `value`: (userdata) a reference to the value to be announced

Returns:
- `n/a`

Discussion:
The announced value must be retrieved prior to use this function or you may use self for announcing the sensor value the script is attached to.

Usage:
```
imsb.announceValue(self)
local aSensValue = SensorValue:get("Receiver voltage")
imsb.announceValue(aSensValue)
```

`imsb.announceValueStat(value [,”max”|”min”|”avg”])`

Announces a flight statistic value of a given sensor value. The standard notification message is used by this function.

Parameters:
- `value`: (userdata) a reference to the value to be announced
- `“max”, “min”` or `“avg”`: (string, optional) specifies whether the min, max or average statistic value should be announced. Can be omitted if only one statistic is defined for the value

Returns:
- `n/a`

Discussion:
The announced value must be retrieved prior to use this function or you may use `self` for announcing the sensor value the script is attached to.

Usage:
```
imsb.announceValueStat(self, “min”)
local aSensValue = SensorValue:get("Receiver voltage")
imsb.announceValueStat(aSensValue, “avg”)
```

`imsb.announceAlarms()`

Announces all currently active alarms.

Parameters:
- `n/a`

Returns:
- `n/a`

Usage:
```
imsb.announceAlarms()
```

`imsb.announceMessage(message)`

Announces a freely defined message.

Parameters:
- `message`: (string) a message string to be announced

Returns:
- `n/a`

Usage:
```
imsb.announceMessage(“iMSB is great”)
```

`imsb.announceView([viewName])`

Announces all values contained in the named display view, or in the currently shown view if this parameter is omitted.

Parameters:
- `viewName`: (string, optional) the name of the view whose values are to be announced. When omitted all values in the currently displayed view will be announced

Returns:
- `n/a`

Usage:
```
imsb.announceView(“Sensors”)
imsb.announceView()
```

`imsb.playSound(soundName)`

Plays on of the available system sounds.

Parameters:
- `soundName`: (string) the name of the sound to play

Returns:
- `n/a`

Discussion:
The name provided with soundName must match with one of the sounds provided by the application.

Usage:
```
imsb.playSound(“Alert”)
```

`imsb.displayNotification(type, notificationText)`

Displays the notification text hud (popup) and displays a user defined text.

Parameters:

- `type`: (number) type specified which hud type, i.e. notification icon is displayed in the notification hud
- `notificationText`: (string) the notification text to be displayed

Returns:

- `n/a`

Discussion:

The following notification types are available and can be set with the `type` parameter:

- `1` Information
- `2` Alarm occurrence
- `3` Alarm clearance
- `4` Above level notification
- `5` Below level notification
- `6` Changed notification
- `7` Turn notification
- `8` Reached goal notification

Usage:

```
imsb.displayNotification(1, “Minimum voltage reached. It’s time to land!”)
```

`imsb.writeLogEvent(type, title[, text])`

Writes a user defined message into the flight log.

Parameters:

- `type`: (number) type specified which kind of log entry is to be generated
- `title`: (string) the log event title (short text)
- `text`: (string, optional) the log event text

Returns:

- `n/a`

Discussion:

The following message types are available and can be set with the `type` parameter:

- `1` Information
- `2` Alarm occurrence
- `3` Alarm clearance
- `4` Above level notification
- `5` Below level notification
- `6` Changed notification
- `7` Turn notification
- `8` Reached goal notification

Usage:

```
imsb.writeLogEvent(2, “Receiver voltage”, “Minimum voltage reached”)
```

ii. Application navigation

`imsb.showView(viewName)`

Shows the display view specified by its name.

Parameters:
- `viewName`: (string) the display view to be shown

Returns:
- `n/a`

Usage:
```
imsb.showView(“Sensors”)
```

`imsb.showNextView()`

Shows the next display view.

Parameters:
- `n/a`

Returns:
- `n/a`

Usage:
```
imsb.showNextView()
```

`imsb.showPreviousView()`

Shows the previous display view.

Parameters:
- `n/a`

Returns:
- `n/a`

Usage:
```
imsb.showPreviousView()
```

`imsb.varioPlayer(onOff)`

Starts or stop the vario player sound.

Parameters:
- `onOff`: (boolean) specifies if vario sound is enabled (`true`) or disabled (`false`)

Returns:
- `n/a`

Usage:
```
imsb.varioPlayer(true)
```

`imsb.voiceAnnouncements(onOff)`

Enables or disables voice announcements.

Parameters:
- `onOff`: (boolean) specifies if voice announcements are enabled (`true`) or disabled (`false`)

Returns:
- `n/a`

Usage:
```
imsb.voiceAnnouncements(true)
```

`imsb.speechRecognition(onOff)`

Enables or disables speech recognition functionality.

Parameters:
- `onOff`: (boolean) specifies if speech recognition is enabled (`true`) or disabled (`false`)

Returns:
- `n/a`

Usage:
```
imsb.speechRecognition(false)
```

iii. Miscellaneous functions

`imsb.elapsedTime()`

Returns the elapsed time in seconds since the beginning of the logging session.

Parameters:
- `n/a`

Returns:
- `time`: (float) actual elapsed time

Discussion:
You should always use this function to determine the time since the start of the logging session, as it correctly takes into account cueing in playback mode, as well as data update rates.

Usage:
```
time = imsb.elapsedTime()
```

`imsb.dataUpdateRate()`

Returns the sensor value update rate, i.e. the number of times per second at which data is being refreshed by the application.

Parameters:
- `n/a`

Returns:
- `update`: (int) data update rate

Discussion:
Available update rates at the time being are 1, 2 and 4 Hz, depending on the value defined in the application settings.

Usage:
```
updRate = imsb.dataUpdateRate()
```

`imsb.dataReaderMode()`

Indicates whether the data reader is performing a real time log or reading from a recording file. In addition when doing a file reading returns the playback mode.

Parameters:
- `n/a`

Returns:
- `rdMode`: (int) reader mode indicating if performing a real time log or reading from a recording file (playback mode)
- `pbMode`: (int) indicates the playback mode when reading from file

Discussion:
The reader mode can return the following values:
- `1` not reading
- `2` data reader performing a real time log
- `3` data reader reading from file recording

The playback mode can take the following values:
- `0` playback mode none
- `1` normal speed reading
- `2` fast forward reading
- `3` slow motion reading
- `4` cueing, i.e. seeking new file position

Usage:
```
rdMode, pbMode = imsb.dataReaderMode()
```

`imsb.stopLogger([stayOnView])`

Stops data logging and recording automatically, without user intervention.

Parameters:
- `stayOnView`: (boolean, optional) indicates whether after the logger has stopped the menu view should be displayed (default behaviour) or the actual displayed view should be kept (`stayOnView = yes`)

Returns:
- `n/a`

Discussion:
This method will stop logging and save the current log recording. The effect is identical as if the user pushes the stop button on the menu view

Usage:
```
imsb.stopLogger()
imsb.stopLogger(true)
```

The device library

The `device` library provides a set of functions related to positional and attitude information of your iOS device. Its grants access to the internal GPS and motion sensors.

The library functions are divided into two categories:

- Location data acquisition to access the device position, altitude and orientation
- Motion data acquisition for accessing attitude and rotation rates

In order to use these services they must first be successfully started before the requested information can be accessed.

GPS and motion sensor data should be used with parsimony in order to save processor resources and battery life.

i. Acquiring location data

`device.startLocationUpdates(doOrientation [, callbackFct])`

Starts acquiring current iOS device location in order to provide continuous position, altitude and orientation data. An optional callback function can be defined to be automatically notified of any location changes.

Parameters:

- `doOrientation`: (boolean) indicates whether device orientation should be acquired
- `callbackFct`: (function, optional) the reference to a callback function in you code that will be called when changes to the device location is available

Returns:

- `success`: (boolean) true if the service could be successfully started

Discussion:

This method should be called once in the script in order to start acquiring location data. Once the location service is started the device `getPosition()`, `getAltitude()` and `getOrientation()` may be called to get the corresponding information. An optional callback function can be defined to be notified of any available location changes.
Note that if device orientation is not required this should be disabled via the `doOrientation` argument in order reduce the amount of processed data.

Usage:

```
function myCallback()
  print(“location updates are available”)
end
success = device.startLocationUpdates(false, myCallback)
```

`device.stopLocationUpdates()`

Stops acquiring device location and disables the location service.

Parameters:

- `n/a`

Returns:

- `n/a`

Discussion:

This function should be called when location updates are not need anymore. Calling `stopLocationUpdates` disables all notification updates in the current lua script.

Usage:
```
device.stopLacationUpdates()
```

`device.getPosition()`

Returns the iOS device actual position coordinates from its internal GPS.

Parameters:
- `n/a`

Returns:
- `{lat=, lon=}`: (table) a table containing latitude and longitude coordinate of actual device position (expressed in decimal degrees)
- `nil` if no GPS coordinates are available

Discussion:
Position data is only available if `device.startLocationUpdates()` has been called initially.
This function uses the iOS device GPS to return its actual position coordinates. In order to use this function your iOS device must be equipped with a GPS.

Usage:
```
postition = device.getPosition()
```

`device.getAltitude()`

Returns the measured absolute altitude of the iOS device.

Parameters:
- `n/a`

Returns:
- `altitude`: (float) actual altitude expressed in meters
- `nil` if no GPS coordinates are available

Discussion:
Altitude data is only available if `device.startLocationUpdates()` has been called initially.
This function uses the iOS device GPS to return its actual altitude. In order to use this function your iOS device must be equipped with a GPS.

Usage:
```
altitude = device.getAltitude()
```

`device.getOrientation()`

Returns the iOS device current orientation measured in degrees relative to north.

Parameters:
- `n/a`

Returns:
- `angle:` (float) actual device orientation in the range 0.0 to 360.0 degrees

Discussion:
Orientation data is only available if `device.startLocationUpdates()` has been called initially.

This function always returns the device orientation relative to north.

Usage:
```
angle = device.getOrientation()
```

ii. Acquiring motion data

`device.startMotionUpdates(updateRate, resetAttitude [, callbackFct])`

Stats acquiring iOS motion information in order to provide continuous attitude and rotation rate data.

Parameters:
- `updateRate`: (integer) indicates at which rate (expressed in Hz) motion data are sent to the script
- `resetAttitude`: (boolean) when set to `true` will level the attitude information to the current device position when the function is called
- `callbackFct`: (function, optional) the reference to a callback function that will be called at each `updateRate` time interval to provide new motion data

Returns:
- `success`: (boolean) true if the service could be successfully started

Discussion:
This method should be called once in the script in order to start acquiring motion data. Once the motion service is started the device `getAttitude()` and `getRotationRate()` may be called to get the corresponding information.
The `updateRate` parameter should be set to the minimum rate required by your script in order save processor resources and battery life. Note that the provided callback function will be called at each update rate time interval. You would typically use it perform application specific processing based on the provided motion data.

Usage:
```
function myCallback()
  print(“motion updates are available”)
end
success = device.startMotionUpdates(5, true, myCallback)
```

`device.stopMotionUpdates()`

Stops acquiring device motion data and disables the service.

Parameters:
- `n/a`

Returns:
- `n/a`

Discussion:
This function should be called when motion updates are not need anymore. Calling `stopMotionUpdates` disables all notification updates in the current lua script.

Usage:
```
device.stopMotionUpdates()
```

`device.getAttitude()`

Returns the attitude of the device. This function provided a measurement of attitude, that is, the orientation of a body relative to a given frame of reference expressed as Euler angles.

Parameters:
- `n/a`

Returns:
- `{roll=, pitch=, yaw=}`: (table) a table containing the device attitude expressed in Euler angles
- `nil` if no valid data is available

Discussion:
Device attitude is only available if `device.startMotionUpdates()` has been called initially.
The returned table contains data specifying the device’s attitude in a given frame of reference. The attitude is measured by the iOS's device gyroscope. In order to use this function your iOS device must be equipped with a gyroscope.

Usage:
```
tableVar = device.getAttitude()
```

`device.getRotationRate()`

Returns the device rotation rate around its x, y and z axes.

Parameters:
- `n/a`

Returns:
`{x=, y=, z=}`: (table) a table containing the device rotation in its x, y and z axis
nil if no valid data is available

Discussion:
Rotation data is only available if `device.startMotionUpdates()` has been called initially.
The returned table contains data specifying the device’s rate of rotation around three axes. Rotation rates are measured by the iOS's device gyroscope. In order to use this function your iOS device must be equipped with a gyroscope.

Usage:
```
tableVar = device.getRotationRate()
```

Example scripts

RPMVarioPlayerCutoff.lua

```
--[[  RPMVarioPlayerCutoff.lua
A very simple script to cutoff the vario player when the engine RPM
value exceeds a user definable threshold.
This sensor value does not provide any value (dummy), but is solely
used to control the vario player.
In addition log events and sound notifications are generated
--]]

local rpmValue
local rpmThreshold
local varioPlayerOn

-- specify user definable parameters for rpm threshold and log
messages
luaValueParam = {
  {id= "rpmThr", name= "RPM cutoff threshold", default= 5000},
  {id= 2, name= "Message disabled", type= "string", default= "Vario
player is disabled"},
  {id= 3, name= "Message enabled", type= "string", default= "Vario
player is enabled"}
}

-- script start initialisation
function initialiseSensorValue(self)
  rpmValue = SensorValue:get(5, 0x500)

  rpmThreshold = self:getLuaValueParam("rpmThr")
  varioPlayerOn = true
  imsb.varioPlayer(varioPlayerOn)
end

-- function called on each value update cycle
function updateSensorValue(self)
  local val = rpmValue:getValue()
  if (val > rpmThreshold and varioPlayerOn == true) then
    varioPlayerOn = false
    imsb.varioPlayer(varioPlayerOn)
    -- issue log message and sound
    imsb.writeLogEvent(1, "Vario player: ", self:getLuaValueParam(2))
    imsb.playSound("Boing")

  elseif (val < rpmThreshold and varioPlayerOn == false) then
    varioPlayerOn = true
    imsb.varioPlayer(varioPlayerOn);
    -- issue log message and sound
    imsb.writeLogEvent(1, "Vario player: ", self:getLuaValueParam(3))
    imsb.playSound("Boing")
  end
end
```

MultiStageAlarm.lua

```
--[[  MultiStageAlarm.lua
This example shows how multi stage alarms, i.e. issuing specific
notifications and messages depending on the alarm criticality, can be
generated.
It specifically shows how to issue various alarms when battery used
capacity exceeds a given threshold and subsequently when this
threshold is exceeded by 10% and 20%.
As a further example a speed record with varying announcements is also
exemplified.

A dedicated string type sensor value is used here that allows
displaying an alarm message in a value text field directly.
Setup:
- configure the script value with value type "String"
- in your display view allocate a Text field in order to display the
alarm message as a value content
--]]

-- specify user definable parameters for capacity threshold and alarm
messages
luaValueParam = {
  {id= "bCapa", name= "Used capacity limit", default= 1500},
  {id= "bCapaAlarm1", name= "Alarm message", type= "string", default=
"Low battery level. Its time to land"},
  {id= "bCapaAlarm2", name= "Alarm message critical", type= "string",
default= "Battery capacity critical.\n It's really time to land
now!"},
  {id= "bCapaAlarm3", name= "Alarm message dead", type= "string",
default= "... sorry can't do anything for you anymore"},
  {id= "spdRec", name= "Speed record above", default= 90},
  }

-- capacity alarm
local capacityValue
local capaAlarmLevel
local capaLimit

-- speed record
local speedValue
local lastSpeedRecord
local lastSpeed
local numSpeedRecords

-- script start initialisation
function initialiseSensorValue(self)
  capacityValue = SensorValue:get("Capacity")

  capaLimit = self:getLuaValueParam("bCapa")
  capaAlarmLevel = 0

  self:setValue("Ok")

  speedValue = SensorValue:get("Speed")

  lastSpeed = speedValue:getValue()
  lastSpeedRecord = self:getLuaValueParam("spdRec")
  numSpeedRecords = 0;
end

-- function called on each value update cycle
function updateSensorValue(self)
  -- capacity checking and alarming
  local val = capacityValue:getValue()
  if (val > capaLimit and capaAlarmLevel < 10) then
    -- trigger first alarm stage
    self:forceValueAlarmed(true, self:getLuaValueParam("bCapaAlarm1"))
    self:setValue(self:getLuaValueParam("bCapaAlarm1"))
    capaAlarmLevel = 10

  elseif (val > (capaLimit + capaLimit * 0.1) and capaAlarmLevel < 20)
then
    -- !! not sure if another alarm can be issued here....!!!
    -- trigger second alarm stage
    self:forceValueAlarmed(true, self:getLuaValueParam("bCapaAlarm2"))
    self:setValue(self:getLuaValueParam("bCapaAlarm2"))
    capaAlarmLevel = 20

  elseif (val > (capaLimit + capaLimit * 0.2) and capaAlarmLevel < 30)
then
    -- trigger second alarm stage
    self:forceValueAlarmed(true, self:getLuaValueParam("bCapaAlarm3"))
    self:setValue(self:getLuaValueParam("bCapaAlarm2"))
    capaAlarmLevel = 30
  end

  -- speed record checking. Only consider speed record announcements
when no capacity alarm is available
  -- do not consider record if speed is rising
  val = speedValue:getValue()
  if (val > lastSpeedRecord and val <= lastSpeed and capaAlarmLevel <
10) then
    lastSpeedRecord = val
    numSpeedRecords = numSpeedRecords + 1

    -- construct different messages depending on the amount of records
achieved
    local msg;
    if (numSpeedRecords < 3) then
      msg = "New speed record: " .. val .. " " .. speedValue.getUnit()
    elseif (numSpeedRecords < 5) then
      msg = "And another speed record: " .. val .. " " ..
speedValue.getUnit()
    else
      msg = "Wow incredible! again a speed record:  <val>" .. val .. "
" .. speedValue.getUnit()
    end

    imsb.playSound("doorbell")
    imsb.announceMessage(msg)
    imsb.writeLogEvent(1, "Speed record", msg)
  end
  lastSpeed = val
end
```

Change history

Document version	Date	Status	Short description of the modification
1.0	03.11.2017	Draft	Initial version
1.1	18.03.2018	Release	Added device library and value parameters
1.2	24.07.2018	Release	Value can be written to other script value, instead of only self Added :getAddress() method
1.3	13.04.2019	Release	Added :getDisplayRange() method
1.4	31.07.2019	Release	Added support for iMSB FASSTest app version Added method to support HoTT and FASSTest transmitter events
1.5	10.02.2021	Release	Added control transition and condition methods to handle physical switch conditions on Graupner MZ16/32 transmitters