ifSGNodeField

Available since firmware version 7.0

The **ifSGNodeField** interface allows querying, getting, setting, and performing other similar manipulation operations on Scene Graph node fields. This interface also allows you to set and unset event observers on a subject node field.

Implemented By

- roSGNode

Supported Methods

- hasField(fieldName as String) as Boolean
- getField(fieldName as String) as Object
- getFields() as Object
- addField(fieldName as String, type as String, alwayNotify as Boolean) as Boolean
- addFields(fields as Object) as Boolean
- getFieldType(fieldName as String) as String
- getFieldTypes() as Object
- setField(fieldName as String, value as Object) as Boolean
- setFields(fields as Object) as Boolean
- removeField(fieldName as String) as Boolean
- removeFields(fieldNames as Object) as Boolean
- queueFields(queueNode as Boolean) as Boolean
- observeField(fieldName as String, functionName as String) as Boolean
- observeField(fieldName as String, port as Object) as Boolean
- unobserveField(fieldName as String) as Boolean
- observeFieldScoped(fieldName as String, functionName as String) as Boolean
- observeFieldScoped(fieldName as String, port as Object) as Boolean
- unobserveFieldScoped(fieldName as String) as Boolean
- threadInfo()
- signalBeacon(beacon As String) As Integer

Description of Methods

hasField(fieldName as String) as Boolean

Returns true if the subject node has a field whose name exactly matches `fieldName`, or whose fully lowercase analog is identical to that of `fieldName`. Returns false otherwise.

getField(fieldName as String) as Object

Returns the appropriately-typed value from the subject node field identified by `fieldName`.

> You can also use the *`node.field`* syntax to get the same result as `getField()`. That is, `rectpos = rect.getField("translation")` is equivalent to `rectpos = rect.translation`. You can also use the syntax *`node`*`[`*`fieldName`*`]`.

getFields() as Object

Returns an associative array with all of the node field names as associative array element keys, and corresponding field values as corresponding associative array element values.

addField(fieldName as String, type as String, alwayNotify as Boolean) as Boolean

Adds the field name specified by `fieldName`, with the type specified by `type`, to the subject node. The added field is initialized to the default value for the type. The third argument, `alwaysNotify`, specifies whether observers of the field are triggered only when the field value changes (`alwaysNotify` set to `false`), or whenever the field value is set, whether to the same or a new value (`alwaysNotify` set to `true`). Returns true if the field is successfully added, false otherwise.

addFields(fields as Object) as Boolean

Adds the field(s) and corresponding field value(s) defined as key-value pair(s) in the associative array `fields` to the subject node. The types of the added fields are determined by the values which correspond to the allowable types for an <interface> field. Returns true if the fields are successfully added, false otherwise.

getFieldType(fieldName as String) as String

Returns a string of the field type specified by `fieldName` for the subject node.

getFieldTypes() as Object

Returns an associative array for the subject node containing all the field names as the array keys, and corresponding field types as the array values.

setField(fieldName as String, value as Object) as Boolean

Sets the value of the subject node field identified by `fieldName` to `value`. This will fail and stop script execution if the value is not of the appropriate type. Returns true if the field is successfully set, false otherwise.

> You can also use the *`node.field`* syntax to get the same result as `setField()`. That is, `rect.setField("opacity", 0.5)` is equivalent to `rect.opacity = 0.5`.

setFields(fields as Object) as Boolean

Sets the value of subject node fields in the `fields` associative array indicated by the corresponding key. Returns true if the fields are successfully set, false otherwise.

removeField(fieldName as String) as Boolean

Removes the field specified by `fieldName` from the subject node. Returns true if the field is successfully removed, false otherwise. Fields defined as part of the built-in node class cannot be removed. Fields defined in Content Meta-Data and the related Scene Graph node class meta-data bindings can be removed, but will be dynamically re-added at any time they are explicitly accessed.

removeFields(fieldNames as Object) as Boolean

Removes the fields specified by an array of `fieldNames` from the subject node. Returns true if the fields are successfully removed, false otherwise. Fields defined as part of the built-in node class cannot be removed. Fields defined in Content Meta-Data and the related Scene Graph node class meta-data bindings can be removed, but will be dynamically re-added at any time they are explicitly accessed.

queueFields(queueNode as Boolean) as Boolean

If `queueNode` is set to true, subsequent operations on the node fields will queue on the node itself rather than on the Scene node render thread. This prevents the operations from being executed immediately. Subsequently setting `queueNode` to false will then cause all of the operations to be transferred to the Scene node render thread queue to be immediately executed in a single render cycle. This can be helpful when multiple fields of a node that affect the appearance of the user interface need to be changed at one time from another thread. It should not be used on a node that is not owned by the render thread, as the render thread will not be able to execute the operations when they are released to it. You can use it when a node owned by a Task node thread is transferred to the render thread, by setting it to a child or a node field of a node already owned by the render thread, where the queue is then released.

observeField(fieldName as String, functionName as String) as Boolean

Calls the function specified by `functionName` when the subject node field specified by `fieldName` changes. The function called must be in the scope of the current component. Optionally, this form can pass an roSGNodeEvent message to the callback function, by specifying the message object as an argument to the callback function:

```
sub callback_function(message as Object)
  ...
end sub
```

From this message in the callback function, you can get the node ID, the field name, and the field value at the time it was set, using the same roSGNodeEvent methods described in the overloaded form observeField(fieldName as String, port as Object). The roSGNodeEvent message also includes a pointer to the node that can be accessed using `getRoSGNode()`, to associate nodes without an ID in the callback function. Additional information can be accessed in the callback function by storing the information in a custom field of the node.

observeField(fieldName as String, port as Object) as Boolean

Available since firmware version 7.1

This overloaded form sends an roSGNodeEvent message to the roMessagePort identified by `port` when the subject node field identified by `fieldName` changes value. Running `GetNode()` on the message retrieves the ID of the node that changed, running `GetField()` on the message gets the name of the field that changed, and running `GetData()` on the message gets the new field value at the time of the change. This allows other threads to react to field changes, and avoids missing a value when the field changes twice before the message handler is able to receive the roSGNodeEvent messages.

unobserveField(fieldName as String) as Boolean

Removes the previously established connections between the subject node field identified by `fieldName` and any callback functions or message ports.

observeFieldScoped(fieldName as String, functionName as String) as Boolean

Available since firmware version 7.5

Similar to observeField(fieldName as String, functionName as String), this method sets up a connection between the observed node's field and the current component from which this call is made. While the connection exists, any change in the called/observed node's field specified by `fieldName` results in calling the function specified by `functionName` in the observing component.

The callback will be on the thread that owns the observed node. This is usually the render thread except in some narrowly defined scenarios. See SceneGraph Threads for further details.

The connection state is implicitly stored in the calling/observing component. This means that multiple such connections made from different components to the same observed node have different lifetimes bounded by that of the calling/observing component. See unobserveFieldScoped() for further details.

observeFieldScoped(fieldName as String, port as Object) as Boolean

Available since firmware version 7.5

Similar to observeField(fieldName as String, port as Object), this method sets up a connection between the observed node's field and the current component from which this call is made. While the connection exists, any change in the called/observed node's field specified by `fieldName` result s in a message being sent to the roMessagePort specified by `port`.

The message will be received on the thread that owns the port. This is either a task thread or the main BrightScript thread. See SceneGraph Threads for further details.

The connection state is implicitly stored in the calling/observing component, just as for the other form of `observeFieldScoped()`. This means that multiple such connections made from different components to the same observed node have different lifetimes bounded by that of the calling/observing component. See unobserveFieldScoped() for further details.

unobserveFieldScoped(fieldName as String) as Boolean

Available since firmware version 7.5

Much like unobserveField(fieldName as String) undoes both forms of `observeField()`, this undoes both forms of `observeFieldScoped()`. It removes the connection between the observing component and the observed node's field. This method looks for and removes the implicit connection state stored in the observing object, so that the calling component will no longer receive notification of changes in the specified node's field. Connections in other observing objects or even in the observed object are not affected.

threadInfo()

Available since firmware version 9

threadInfo() is a roSGNode function that returns an Associative Array (AA) to Brightscript with information that can be analyzed to help minimize Rendezvous spread. This function can be called on any node from any thread and it returns the following information about that node in an AA similar to:

```
{   node: { type: "XXComponent",
 id: "XXID",
 address: 0x123XXX,
 willRendezvousFromCurrentThread: "yes",
 owningThread: { type: "Render", name: "newMainScene", id:"123456" }
},
 currentThread: { type: "Task",   name: "conviva",     id: "234567" },
 renderThread: { type: "Render", name: "newMainScene", id: "123456" }
}
```

roSGNode.threadInfo() has been developed as a runtime introspective informational tool for the developer to help them understand the following:

- What is calling a thread function is being called from,

- On which component's behalf (what m.top) the current function is executing,
- The thread ownership of the node in question, and
- Whether or not access to the node from the current thread would cause a rendezvous

threadInfo() returns an associative array with all of this information.

> Do not call the threadInfo() function from within function main() or any function called by function main().

signalBeacon(beacon As String) As Integer

Available since firmware version 9.1

Signals start and/or stop points for measuring channel launch and Electronic Program Guide (EPG) launch times.

To pass certification, a channel must finish launching within the time specified in the certification performance requirements. The Roku firmware automatically fires an **AppLaunchInitiate** event to mark when the user presses the OK button to launch a channel from the Roku home screen. The channel, however, must fire the corresponding **AppLaunchComplete** to mark when the channel home page is fully rendered (or when video playback starts after handling a deep link, and the channel can respond to commands sent via the remote control).

If your application contains an EPG, the application must also fire both an **EPGLaunchInitiate** beacon when the user initiates a keypress to display the EPG and the corresponding **EPGLaunchComplete** beacon when the EPG is fully rendered and navigable.

To fire the **AppLaunchComplete**, **EPGLaunchIntitiate**, and **EPGLaunchComplete** beacons within your application, call the **signalBeacon()** funct ion on any node as demonstrated in the following examples:

```
myScene.signalBeacon(“AppLaunchComplete”)
myEPGComponent.signalBeacon(“EPGLaunchInitiate”)
m.top.signalBeacon(“EPGLaunchComplete”)
```

> Only the first sequence of EPG launch beacons is recorded. If a user launches the EPG more than once while the channel is running, a warning message is output to the debug console. This warning message, which acknowledges the receipt of the beacon while notifying that subsequent ones will not be recorded, may be ignored.
>
> Only EPG launch sequences that start within 5 seconds of the **AppLaunchComplete** event being fired qualify as a valid measurements for certification. EPG launch sequences fired after the 5-second window are still recorded so that channel performance can be compared against requirements.

The following table summarizes when to fire the AppLaunchComplete and **EPGLauchInitiate**/**EPGLauchComplete** beacons and when their timestamps are recorded:

Launch Event	Placement	Timestamping
AppLaunchComplete	When the channel home page is fully rendered, or when video playback starts and the channel can responds to commands sent via the remote control (if your channel is launched to direct playback). The system automatically fires an AppLaunchInitiated beacon to marks when your app is initially launched.	The first render pass completes after the stop beacon has been signaled.
EPGLaunchInitiate	Where your app initiates the display of the channel guide.	The last keypress before the start beacon was signaled. If there was no prior keypress, the start beacon signal time.

EPGLaunchComplete	Where the channel guide is fully rendered and operational.	The first render pass completes after the stop beacon has been signaled.

When you fire a launch event, the system will return an integer indicating the result of its signaling:

Return Code		Description
0	Success	The event was successfully signaled.
1	Not Ready	The event cannot be fired until after the AppLaunchComplete beacon has been completed.
2	Invalid	An invalid string was passed into the signalBeacon function.
3	Already Signaled	An event that can only be fired once (AppLaunchComplete) was signaled again.
4	Wrong Order	The completion event was fired before the corresponding initiate event (for example, EPGLaunchComplete was signaled before EPGLaunchInitiate).

In addition to the channel launch and EPG launch times, the Roku firmware automatically measures five other certification performance metrics: app compile time, video start time, live start time, channel change time, and channel exit time. You can use the BrightScript console (port 8085) to view a report detailing your channel's performance. See Measuring channel performance for more information.