Handling Application Events

Table of Contents

Observer Callback Models

Firmware v7.5 introduces a fundamental change in the observer callback model changing from a queued or deferred model to a more intuitive and expected recursive callback model.

Queued Callback Model

In firmware releases prior to v7.5, callbacks operated in a queued manner.

Nested Functions Example

```
function init()
    m.top.observeField("f1", "c1")
    m.top.observeField("f2", "c2")
    m.top.f1 = v1
    print "init(): "; m.top.f2
end function
function c1()
    m.top.f2 = v2 ' immediate callback of c2() right here
    print "c1(): "; m.top.f2
end function      ' deferred callback of c2() on return
function c2()
    print "c2(): "; m.top.f2
end function
```

In the nested functions example above, the queued callback model would have the ordering of the print statements as:

Output

```
c1(): v2
c2(): v2
init(): v2
```

Recursive Callback Model

With the recursive callback model, the expected output is now more intuitive and the order of the print statements would be:

Output

```
c2(): v2
c1(): v2
init(): v2
```

With firmware version 7.5 and greater, the `rsg_version` entry in your channel's manifest will default to 1.1. To check and test different SceneGraph versions without refactoring your channel, see the guide on Debugging.

> Please note that support for the "rsg_version=1.0" manifest flag is deprecated as of Roku OS 8. This deprecation means that the 1.0 features continue to work in Roku OS 8, but will no longer be supported (and thus should not be expected to work) starting with the next major firmware release. All channels will have to adopt the current observer callback model in successive firmware updates.

Event Handling

There are three types of events that can occur in a SceneGraph application that you can use to control the behavior of an application component.

- User remote control key presses
- Component and node field value changes, such as state changes of a previously-launched component, and node actions
- Functional Fields

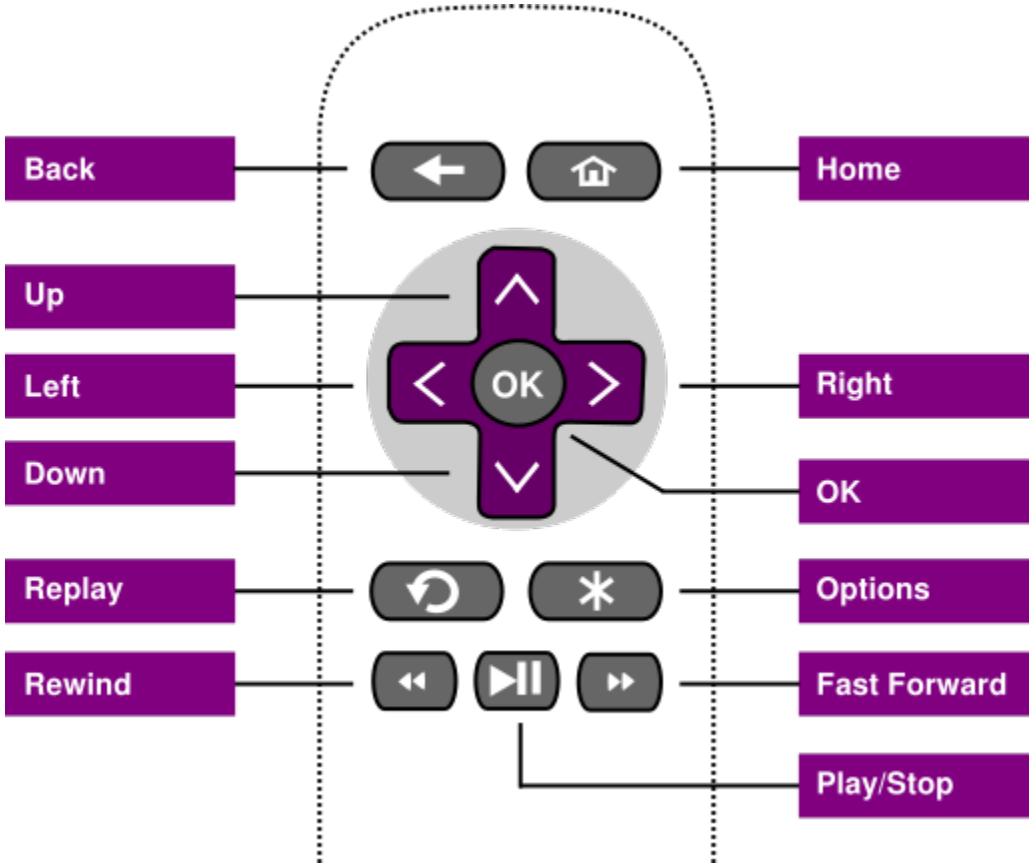
Handling Remote Control Key Presses

You can handle remote control key events by writing an **`onKeyEvent()`** function in the **`<script>`** element of the component you want to handle the key press event. When the component, or its children, have remote control focus, the **`onKeyEvent()`** function is called whenever an unhandled key event bubbles up the focus chain to the component.

Using the onKeyEvent() Function

The **`onKeyEvent()`** function takes two parameters, `key` and `press`. The `press` parameter is a Boolean value that is true if the key was pressed, and false if the key was released. The `key` parameter of the **`onKeyEvent()`** function contains a string, which is case-sensitive, that identifies the

key that was pressed. The `key` strings supported by the **`onKeyEvent()`** function, and the corresponding remote key, are as follows:



String	Key	Appearance/Icon
back	**Back**	left-pointing arrow at top of remote
up	**Up**	up-pointing caret of remote directional pad
down	**Down**	down-pointing caret of remote directional pad
left	**Left**	left-pointing caret of remote directional pad
right	**Right**	right-pointing caret of remote directional pad
OK	**OK**	key usually labeled **OK** near or in the center of remote directional pad
replay	**Replay**	key usually labeled with a circular-pointing arrow
play	**Play/Stop**	key usually labeled with a right-pointing triangle and two bars
rewind	**Rewind**	key usually labeled with two left-pointing triangles
fastforward	**Fast Forward**	key usually labeled with two right-pointing triangles
options	**Options**	key labeled with an asterisk

The **onKeyEvent()** function must return true if the component handled the event, or false if it did not handle the event. Returning false allows the event to continue bubbling up the focus chain (see **Remote Control Events**) so that ancestors of the component can handle the event.

Starting from firmware 8, the behavior of the Roku system overlay is such that the system overlay now slides in whenever the * button is pressed, the Video node is in focus, and the app does not have its OnKeyEvent() handler fired. When the Video node is not in focus, the system overlay does not slide in and the OnKeyEvent() handler is fired.

> There are one or more keys on any Roku remote control which are not handled by the **onKeyEvent()** function (or any Roku application event handler), such as the **Home** key. Presses of these keys are handled by the global Roku firmware event handler in a default manner that cannot be modified by application code. Also note that several node classes handle certain remote control key events automatically, so **onKeyEvent()** is not required to handle those events, and should not be used for those events in those nodes. As an example of node classes that automatically handle certain remote control key events, grid node classes such as **PosterGrid** automatically handle **Up** , **Down**, **Right**, and **Left** key presses when the poster grid has focus. Typically, you should use the **ifSGNodeField** **observeField()** m ethod to handle changes in the subject node fields caused by automatic key event handling of the node.

Example

The following `onKeyEvent()` example handles supported remote control key presses other than the **Back** key by displaying a warning message until the **OK** key is pressed.

onKeyEvent() Event Handling Example

```
function onKeyEvent(key as String, press as Boolean) as Boolean
  handled = false
  if press then
    if (key = "back") then
      handled = false
    else
      if (m.warninglabel.visible = false)
        m.warninglabel.visible="true"
      else
        if (key = "OK") then
          m.warninglabel.visible="false"
        end if
      end if
      handled = true
    end if
  end if
  return handled
end function
```

Handling Node Field Value Changes

There are two **ifSGNodeField** methods that allow you to create (and remove) observers that continuously monitor any field value of any **roSGNode** object, including the interface fields you have created for custom SceneGraph components:

- **observeField()**
- **unobserveField()**

Using the ifSGNodeField observeField() and unobserveField() Methods

observeField() is an overloaded method with two versions, useful for different purposes. The first allows you to trigger a specified callback function in response to any change in the value of the observed node field. For example, to set up an observer of a **Timer** node `fire` field that calls a `handleexampletimerfire()` event handler function that you write:

```
exampletimer.ObserveField("fire", "handleexampletimerfire")
```

Once this observer is set up, the component will continuously monitor the `exampletimer` node object `fire` field for the remaining existence of the component or node, or until **unobserveField()** is called (perhaps as part of the event handler function itself):

```
exampletimer.unobserveField("fire")
```

The event handler function you write must be included in the component **<script>** element, and manipulate objects within the scope of the component.

> By default observers are only called when the value of a field changes. Code that assigns a field value to the same value as currently assigned to the field will not trigger observers. If you want observers to be called any time a field value is set, regardless of the value, an **<interface>** field must be defined with the `alwaysNotify` attribute set to true.

Here is an example of a field observer and the associated event handler function:

Field Observer XML BrightScript Example

```
<script type = "text/brightscript" >

  <![CDATA[

  sub init()
    m.top.setFocus(true)
    m.bottomlabel = m.top.findNode("bottomLabel")
    m.texttimer = m.top.findNode("textTimer")
    m.defaulttext = "All The Best Videos!"
    m.alternatetext = "All The Time!!!"
    m.textchange = false
    m.texttimer.ObserveField("fire", "changetext")
  end sub

  sub changetext()
    if (m.textchange = false) then
      m.bottomlabel.text = m.alternatetext
      m.textchange = true
    else
      m.bottomlabel.text = m.defaulttext
      m.textchange = false
    end if
  end sub

  ]]>

</script>
```

Optional roSGNodeEvent Callback Function Argument
Note that field observer callback functions can specify an roSGNodeEvent argument. For example, the changetext() callback function signature in the example above could have been written as sub changetext(event as roSGNodeEvent). In this case, the callback function can call the roSGNodeEvent functions to extract information about the node that triggered the callback, specific field that triggered the callback, etc.

The second version of **`observeField()`** lets you specify a message port to notify when the observed field changes:

```
m.texttimer.ObserveField("fire", m.port)
```

This second case is used when you want a field change to trigger an event in a BrightScript message loop. This is useful when using BrightScript components (such as roChannelStore) which can only be instantiated in the main BrightScript thread or a **Task** node thread. In this case, when the observed field changes, an roSGNodeEvent is sent to the port passed to the **`observeField()`** call. The event `getField()` and `getData()` functions can be called to determine the specific node field that changed, and the new value of the field, respectively.

Event Cascades

Setting the value of a field triggers any field observer functions that may be associated with the field as the result of an observeField() call. These observer functions may set other component fields, triggering calls to their observer functions. Such a situation triggers what is called an **event cascade**. Simply put, this is the chain of field setting and observer function calls that result from setting a single field of a node.

For example, if the width field of a Rectangle that contains a Label is set, an observer of that width field might set the width field of the Label. The width field of the Label might have an observer function that sets the Label's wrap field. The chain of field settings and observer callback functions that result from setting a field (in this case, the Rectangle's width field) is an event cascade.

With the release of firmware v7.5, nested observer callbacks are no longer deferred. Observer callbacks now happen recursively. See the **Queued Callback Model** section above for details.

Handling Component <interface> Field Value Changes

Any **<field>** element defined in a component **<interface>** element can have an observer attached by setting the value of the optional `onChange` attribute. Set the `onChange` attribute to the callback function name that will handle the component field value change.

Note that the **<field>** element also includes a related optional attribute, `alwaysNotify`. You can set this attribute to true to have the callback function triggered every time the component field value is set, even if the value itself does not change. To have the callback function triggered only when the component field value changes, set the `alwaysNotify` attribute to false.

Functional Fields

Firmware v7.5 introduces the concept of a functional field. In addition to the field observer model, functions can now be called procedurally.

To use a functional field, define the function within an interface:

Functional Field Declaration

```
<interface>
 <function name="addSomeValue" />
</interface>
```

Next, define what the function does in BrightScript:

Function Definition

```
function addSomeValue(params as Object) as Object
 passedDataLabel = m.top.findNode("passedDataLabel")
    passedDataLabel.text = params.passedString
    result = {
        resultString : "Returned Value " + stri(params.passedInt + 4)
    }

 return result
end function
```

callFunc() is a synchronized interface on roSGNode. It will always execute in the component's owning ScriptEngine and thread (by rendezvous if necessary), and it will always use the m and m.top of the owning component. Any context from the caller can be passed to the function via an associative array, which is fully converted/cloned on call. The result is also fully converted/cloned on return as well.

To call the function, use the `callFunc` field. Parameters can only be passed via one object which can be an associative array argument or an array with arbitrary convertible contents. A return value, if any, can be an object that is similarly arbitrary.

callFunc with parameters

```
params = {passedString:"", passedInt:12}

result = node.callFunc("addSomeValue", params)
```

Your function must determine how to interpret the parameters.