

# Iriel's Field Guide to Secure Handlers

---

*An introduction and reference to the WoW Secure Handlers and their use.*

*Iriel (Silver Hand) - October 2010 – **4.0 revision 1***

## Contents

Contents.....	1
Introduction.....	3
System Design Goals.....	3
A Quick Overview.....	4
Key Concepts.....	4
The Secure Handler Header Templates .....	5
Base Templates.....	5
SecureHandlerBaseTemplate.....	5
SecureHandlerStateTemplate.....	6
SecureHandlerAttributeTemplate.....	7
Common Action Templates .....	7
SecureHandlerClickTemplate.....	7
SecureHandlerDoubleClickTemplate .....	8
SecureHandlerMouseWheelTemplate .....	8
SecureHandlerMouseUpDownTemplate.....	9
SecureHandlerShowHideTemplate.....	10
SecureHandlerEnterLeaveTemplate.....	11
SecureHandlerDragTemplate.....	12
The Restricted Namespace.....	14
Restricted Tables .....	15
The restrictions.....	15
Library functions.....	15
Access from add-on code .....	16
Restricted Closures .....	17
No function / closure creation.....	17
No direct table creation .....	17

Frame Handles .....	18
Normal Read Methods.....	18
Limited Read Methods .....	19
Handle-Specific Read Methods .....	21
Normal Update Methods .....	21
Limited Update Methods.....	22
Handle-Specific Update Methods.....	23
Handle Control Methods.....	24
Wrapping Script Handlers.....	26
Click Wrappers.....	27
OnClick, OnDoubleClick, PreClick, PostClick.....	27
Drag Wrappers.....	28
OnDragStart, OnRecieveDrag .....	28
Simple Wrappers.....	30
OnShow, OnHide, OnEnter, OnLeave .....	30
Other Specialized Wrappers .....	31
OnMouseWheel.....	31
OnAttributeChanged.....	32
Integration With Other UI Components.....	33
State Drivers.....	33
Hover Manager.....	34
Examples? .....	35
Reference Tables.....	36
Frame Handle Method Reference .....	36
Header Snippet Attribute Reference.....	37
Child Snippet Attribute Reference.....	37
Wrapper Snippet Reference.....	37

## Introduction

The Burning Crusade expansion (aka WoW 2.0) brought about some significant changes to the add-on environment, the most impact almost certainly coming from the restrictions on actions and on modification of frames in combat. In order to still permit add-ons to provide interactions with protected frames the SecureStateHeader was conceived, offering some control over protected frames.

While the SecureStateHeader certainly served its purpose in enabling some functionality while being tightly restricted to prevent abuse, those who have used it (and perhaps even more so, those who tried but failed to use it) are aware that it is at best cryptic, and also severely limited. The Wrath of the Lich King (WoW 3.0) completely replaced the SecureStateHeader with the new secure handlers infrastructure that is both far more intuitive as well as more powerful.

This guide aims to describe the features of the new system, as well as a reference for the capabilities it offers.

## System Design Goals

Before embarking on the description of the system, it's worth laying out some of the design goals that shaped the implementation:

The secure handlers intend to:

- allow add-on code to perform certain otherwise-protected actions on frames during combat.
- define the actions in the most intuitive and familiar manner possible.
- allow any 'decision making' that would be available via the macro conditional system.
- allow all functionality previously available with the SecureStateHeader.
- introduce as little overhead as necessary.

Similarly, they intend to prevent:

- interference/control from insecure code during combat.
- access to information about the state of the world beyond that available through macro conditionals.
- unsafe manipulations of frames during combat.
- invocation of functions beyond those explicitly permitted for use.

## A Quick Overview

In order to manipulate protected frames in combat, you begin by creating a Frame that inherits from one of a number of Secure Handler templates. This frame is known as a 'header' frame.

All secure handlers have some built in behaviors in response to setting of certain frame attributes, in addition each of the handler templates provides support for different triggers (such as a click, or mouse enter/leave, or show/hide, etc).

For each of these triggers you specify an action or actions to perform by writing snippets of normal Lua code, this code has access to most of the normal Lua functions and some of the WoW functions, as well as the ability to control protected frames (such as itself and its children). In addition each header frame has its own variable scope that all of its snippets are executed against which you can use to pass data between them and maintain state simply by reading and writing global variables.

The code for each handler is specified by setting a specially named attribute on the owner frame (or on its children), whose value is a String containing Lua source code (You cannot pass closures directly as they may be referencing unapproved functions or data).

## Key Concepts

There are some key concepts behind what was just described, each of which will be explained in more detail later. These are:

- Secure header instances are Frames that inherit from one of the **Secure Handler Templates**.
- Each header instance frame has a **Restricted Namespace** containing those parts of the Lua and wow API's that are permitted, and a global environment to store and share data which is a **Restricted Table**.
- Secure Handler actions are described by writing Lua code, subject to some small limitations, that internally is used to create **Restricted Closures** – secure code compiled from user provided Lua source that runs in a tightly managed 'sandbox'.
- Each of these blocks of Lua code is referred to in this document as a **Snippet**, and is represented as a Lua string (sometimes as the value of a frame attribute)
- The snippets of code can manipulate themselves and other protected frames using **Frame Handles**, which are objects that behave somewhat like normal frames but only allow permitted operations.
- In addition to the direct header actions, a header can also **Wrap** the script handlers on frames which allows both interaction between a other frames and a handler as well as changes to the behavior of otherwise secure frames.

# The Secure Handler Header Templates

## Base Templates

All secure handler header templates are derived from one of three base templates, each of which provides a special **OnLoad** and two which supply an **OnAttributeChanged** script handler to support the core functionality.

### SecureHandlerBaseTemplate

This is the default base template for all secure headers and it provides some helpful utilities in the form of methods that are added to the frame by the template's **OnLoad** handler (which in turn runs **SecureHandler\_OnLoad(self)**, you're able to call this yourself if you're not using the template's load script). These methods cannot be used in combat.

#### header:Execute(*"snippet"*)

---

The provided string snippet is compiled as Lua code and executed as a restricted closure within the owner's environment. This mechanism is intended primarily to populate or change the contents of the environment, but can be used for other purposes.

The method internally calls the global **SecureHandlerExecute(header, "snippet")** function.

#### header:SetFrameRef(*"label"*, *frame*)

---

This method is given a string label and a UI Frame and creates a 'frame reference' on the header for the specified label which can be retrieved with a call to the frame handle **:GetFrameRef("label")** method later. This mechanism is provided to allow a secure handler to be given control over any frame (though there are restrictions on what can be done with unprotected frames during combat). See the **Frame Handles** section later for more details.

The method internally calls the global **SecureHandlerSetFrameRef(frame, "label", reframe)** function.

#### header:WrapScript(*frame*, *"script"*, *"preBody"* [, *"postBody"*])

---

This method adds a 'wrapper' to the specified frame's handler for a given script such as "OnClick", or "OnShow". The *preBody* and *postBody* parameters specify snippets of Lua code to run before and optionally after the original wrapped script handler is called. A number of script types are supported, each with specific properties, see the section **Wrapping Script Handlers** below for details.

This method internally calls the global **SecureHandlerWrapScript(frame, "script", header, "preBody" [, "postBody"])** function.

**header, “preBody”, “postBody” = header:UnwrapScript(*frame*, “*script*”)**

---

This method removes the outer ‘wrapper’ from the specified frame’s handler for a given script such as “OnClick”, or “OnShow”, and returns the header and scripts that the wrapper was constructed with (which will be nil if the current script handler was not a wrapper). See the section **Wrapping Script Handlers** below for details

This method internally calls the global **SecureHandlerUnwrapScript(*frame*, “*script*”)** function.

### SecureHandlerStateTemplate

This template is an extension of the **SecureHandlerBaseTemplate** that supports all of the methods above in addition to adding an **OnAttributeChanged** script handler for the following:

**header:SetAttribute(“state-*<stateid>*”, *value*)**

---

Whenever an attribute is set whose name begins with “state-“ followed by one or more characters then a snippet of code can be executed to react to the state notification:

<b>ATTRIBUTE:</b> <code>_onstate-<i>&lt;stateid&gt;</i></code>
<b>PARAMETERS:</b> <code>(<i>self</i>, <i>stateid</i>, <i>newstate</i>)</code>
<code><i>self</i></code> the header frame’s handle
<code><i>stateid</i></code> the id of the state attribute that changed
<code><i>newstate</i></code> the new value for the state attribute
<b>RETURNS:</b> none

The corresponding “**\_onstate-*<stateid>***” attribute is queried, and if it is a string then the string is used as a snippet of Lua code which is executed with the parameters (**self**, **stateid**, **newstate**), where **self** is the header frame handle, **stateid** is the stateid component of the attribute name, and **newstate** is the value set. This allows state-dependent capabilities like those from the original SecureStateHeader, and is particularly useful in conjunction with the SecureStateDriver mechanism.

## SecureHandlerAttributeTemplate

This template is an extension of the **SecureHandlerBaseTemplate** that supports all of its methods in addition to an **OnAttributeChanged** script supporting the following:

**header:SetAttribute("name" value)**

---

Whenever an attribute is set whose name does not begin with an underscore a snippet of code can be executed to react to the attribute change notification in the same manner as any normal **OnAttributeChanged** script handler:

<b>ATTRIBUTE:</b> <code>_onattributechanged</code>
<b>PARAMETERS:</b> <code>(self, name, value)</code>
<i>self</i> the header frame's handle
<i>name</i> the name of the attribute that changed
<i>value</i> the new value for the attribute
<b>RETURNS:</b> none

## Common Action Templates

These templates each extend the **SecureHandlerBaseTemplate** but each adds a different set of script handlers, you can combine them as necessary to support interaction at the handler owner level. The templates provide for the most common handlers, but you can use the **Wrap** mechanism to enable others, even on a header.

### SecureHandlerClickTemplate

This template simply provides support for the **OnClick** frame action.

**<OnClick>**

---

A snippet of code is obtained from an attribute and called whenever a registered click is received by the frame.

<b>ATTRIBUTE:</b> <code>_onclick</code>
<b>PARAMETERS:</b> <code>(self, button, down)</code>
<i>self</i> the header frame's handle
<i>button</i> the button that was pressed/released
<i>down</i> flag indicating if the click was press or a release
<b>RETURNS:</b> none

## SecureHandlerDoubleClickTemplate

This template simply provides support for the OnDoubleClick frame action.

### <OnDoubleClick>

---

A snippet of code is obtained from an attribute and called whenever a registered double click is received by the frame.

<b>ATTRIBUTE:</b> <code>_ondoubleclick</code>
<b>PARAMETERS:</b> <code>(self, button)</code> <i>self</i> the header frame's handle <i>button</i> the button that was pressed/released
<b>RETURNS:</b> none

## SecureHandlerMouseWheelTemplate

This template simply provides support for the OnMouseWheel frame action. The default OnLoad handler for the template enables the frame for mouse wheel use.

### <OnMouseWheel>

---

A snippet of code is obtained from an attribute and called the mouse wheel is moved over the frame.

<b>ATTRIBUTE:</b> <code>_onmousewheel</code>
<b>PARAMETERS:</b> <code>(self, delta)</code> <i>self</i> the header frame's handle <i>delta</i> the amount of motion in the mouse wheel
<b>RETURNS:</b> none

## SecureHandlerMouseDownTemplate

This template adds both OnMouseUp and OnMouseDown script handlers.

### <OnMouseUp>

---

Whenever a registered mouse button is pressed down an attribute is queried for a snippet of code that is then executed.

<b>ATTRIBUTE:</b> <code>_onmouseup</code>
<b>PARAMETERS:</b> <code>(self)</code>
<code>self</code> the header frame's handle
<code>button</code> the button that was pressed
<b>RETURNS:</b> none

### <OnMouseDown>

---

Whenever a registered mouse button is released, an attribute is queried for a snippet of code that is then executed.

<b>ATTRIBUTE:</b> <code>_onmousedown</code>
<b>PARAMETERS:</b> <code>(self)</code>
<code>self</code> the header frame's handle
<code>button</code> the button that was released
<b>RETURNS:</b> none

## SecureHandlerShowHideTemplate

This template adds both OnShow and OnHide script handlers.

### <OnShow>

---

Whenever the frame is shown an attribute is queried for a snippet of code, which is executed.

<b>ATTRIBUTE:</b> <code>_onshow</code>
<b>PARAMETERS:</b> <code>(self)</code> <code>self</code> the header frame's handle
<b>RETURNS:</b> none

### <OnHide>

---

Whenever the frame is hidden an attribute is queried for a snippet of code, which is executed.

<b>ATTRIBUTE:</b> <code>_onhide</code>
<b>PARAMETERS:</b> <code>(self)</code> <code>self</code> the header frame's handle
<b>RETURNS:</b> none

## SecureHandlerEnterLeaveTemplate

Provides script handlers for the OnEnter and OnLeave frame actions.

### <OnEnter>

---

When the frame is entered by moving the mouse, then the attribute is queried for a snippet of code, which is then executed.

<b>ATTRIBUTE:</b> <code>_onenter</code>
<b>PARAMETERS:</b> <code>(self)</code> <code>self</code> the header frame's handle
<b>RETURNS:</b> none

### <OnLeave>

---

When the frame is left by moving the mouse, then the attribute is queried for a snippet of code, which is then executed.

<b>ATTRIBUTE:</b> <code>_onleave</code>
<b>PARAMETERS:</b> <code>(self)</code> <code>self</code> the header frame's handle
<b>RETURNS:</b> none

## SecureHandlerDragTemplate

This template provides support for drag and drop activities by adding OnDragStart and OnReceiveDrag script handlers.

### <OnDragStart>

---

When dragging is initiated on the frame, the attribute is queried for a snippet of code, which is then executed with both the drag button and the current cursor contents as parameters. The return value of the handler is used to determine if the cursor contents should be changed.

<b>ATTRIBUTE:</b> <code>_ondragstart</code>
<b>PARAMETERS:</b> <code>(self, button, kind, value, ...)</code>
<i>self</i> the header frame's handle
<i>button</i> the button that initiated the drag
<i>kind</i> the kind of thing on the cursor
<i>value</i> details of the thing on the cursor
<i>...</i> any other returns from <b>GetCursorInfo()</b>
<b>RETURNS:</b> <code>["clear",] kind, value, ...</code>
<i>["clear"]</i> if the first result is the string "clear" then the cursor is cleared before processing the rest of the result
<i>kind</i> the kind of thing to pick up ( <b>nil</b> for none)
<i>value</i> details of the thing to pick up
<i>...</i> any other details required for pick up

## <OnReceiveDrag>

---

When dragging is released on the frame, the attribute is queried for a snippet of code, which is then executed with the current cursor contents as parameters. The return value of the handler is used to determine if the cursor contents should be changed.

<b>ATTRIBUTE:</b> <code>_onreceivedrag</code>
<b>PARAMETERS:</b> <code>(self, button, kind, value, ...)</code>
<i>self</i> the header frame's handle
<i>button</i> always <b>nil</b>
<i>kind</i> the kind of thing on the cursor
<i>value</i> details of the thing on the cursor
<i>...</i> any other returns from <b>GetCursorInfo()</b>
<b>RETURNS:</b> <code>["clear",] kind, value, ...</code>
<i>["clear"]</i> if the first result is the string "clear" then the cursor is cleared before processing the rest of the result
<i>kind</i> the kind of thing to pick up ( <b>nil</b> for none)
<i>value</i> details of the thing to pick up
<i>...</i> any other details required for pick up

## The Restricted Namespace

In order to prevent secure handler code from reacting to unapproved world state (such as unit health), all snippets execute with a limited subset of the normal WoW environment, with one or two additional methods to support macro conditional functionality. This namespace includes these Lua library functions:

- The **string** and **math** scopes
- A **table** scope (see later for details, this is not quite the normal table scope)
- The basic Lua functions **select**, **tonumber**, **tostring**, **type** (the type implementation will return “table” for restricted table objects)
- The global string functions
- The global math functions

A collection of WoW API functions including:

- The keyboard modifier query functions (**IsModifierKeyDown**, etc)
- Basic conditional state query functions (**IsMounted**, **IsSwimming**, **IsIndoors**, etc)
- Form query functions, **IsStealthed** and **GetShapeshiftForm**
- Unit existence functions **UnitExists**, **UnitIsDead**, **UnitIsGhost**, **UnitPlayerOrPetInParty**, **UnitPlayerOrPetInRaid**
- A **print(...)** function for outputting string values for debugging and development
- Key binding query function **GetBindingKey**
- Spell/action/item query functions **HasAction**, **IsHarmfulSpell**, **IsHelpfulSpell**, **IsHarmfulAction**, **IsHelpfulAction**, **GetMultiCastTotemSpells**
- Restricted versions of **UnitHasVehicleUI(unit)** and **GetActionInfo**

And some custom functions to expose some macro conditional capabilities:

- flag = **PlayerCanAttack**(unit)
- flag = **PlayerCanAssist**(unit)
- flag = **PlayerIsChanneling**()
- family, name = **PlayerPetSummary**()
- flag = **PlayerInCombat**()
- status = **PlayerInGroup**() where status is *false*, “raid”, or “party”

You can find the full list in the FrameXML file `RestrictedEnvironment.lua`.

This environment is read only to the executing code, and will take precedence over global values with conflicting names set by the snippet.

## Restricted Tables

Normal Lua tables are somewhat problematic within an environment in which there are both sandboxed and unrestricted code, even with the taint model present in World of Warcraft. Because tables are stored by reference if normal tables were allowed in the sandbox it could be possible for those table references to leak out, and then be modified by insecure code during execution.

To prevent this, and to ensure that certain 'risky' objects cannot be persisted in the state between handler calls, all tables available to the running snippet (including the header's global environment) are instances of a special kind of object – a **restricted table** that acts as a gatekeeper to a hidden internal table.

### The restrictions

As suggested by their name, there are a number of restrictions on what can be done with a restricted table:

- They may only be created and modified by secure code (i.e. Blizzard code and handler code for secure headers)
- Table keys may only be of:
  - The base types **string**, **number**, or **boolean**
  - **Frame handles**, which are references to UI frames and are discussed in detail below.
- Table values may only be one of:
  - The base types **nil**, **string**, **number**, and **boolean**.
  - **Restricted tables** so that normal nested table structures can be built.
  - **Frame handles** that reference UI frames.

Any attempt to store an invalid key or value will be rejected as an error.

### Library functions

Within the environment that snippets execute against restricted tables operate almost identically to normal tables. They support the **#** operator, and the following table functions are all defined:

- `key = next(tbl, prevKey)` – Get the next key in a table
- `iter, tbl, preKey = pairs(tbl)` – Iterator for hash table
- `iter, tbl, preKey = ipairs(tbl)` – Iterator for integer keyed table
- `tbl = newtable(...)` – Create a new restricted table
- `... = unpack(tbl)` – Unpack the array part of a table
- `tbl = wipe(tbl)` – Remove all entries from a table
- `tinsert(tbl[, index], value)` – Insert a value into an array table
- `value = tremove(tbl[, index])` – Remove an entry from an array table

As are the following table scope functions:

- `index = table.maxn(tbl)` – Get the highest numeric index for a table
- `table.insert(tbl[, index], value)` – Insert a value into an array table
- `value = table.remove(tbl[, index])` – Remove an entry from an array table
- `table.sort(tbl[,func])` – Sort the array part of a table
- `string = table.concat(tbl)` – Concatenate the array part of a table into a string
- `tbl = table.wipe(tbl)` – Remove all entries from a table
- `tbl = table.new(...)` – Create a new restricted table

In addition there's a special implementation of the `string.gsub` function that can be passed a restricted table as its replacement argument:

- `result = string.rtgsub (s, pattern, repl, n)` – Replace a pattern within a string globally
- `result = rtgsub (s, pattern, repl, n)` – Global scope version of the above

## Access from add-on code

While add-on code cannot create or modify restricted tables, it is possible to obtain the global environment table for a secure header, and read the data contained inside. The `rtbl = GetManagedEnvironment(headerFrame)` function returns the environment for the specified header, or nil if the header is invalid or has not yet created its environment. Direct access to the table for indexing and the `#` operator work as expected, but if you wish to perform other normal table operations, the special restricted table aware versions used within the sandbox are available in the new **rtable** scope. Those that are useful to add-on code are:

- `key = rtable.next(tbl, prevKey)` – Get the next key in a table
- `iter, tbl, preKey = rtable.pairs(tbl)` – Iterator for hash table
- `iter, tbl, preKey = rtable.ipairs(tbl)` – Iterator for integer keyed table
- `... = rtable.unpack(tbl)` – Unpack the array part of a table
- `index = rtable.maxn(tbl)` – Get the highest numeric index for a table
- `string = rtable.concat(tbl)` – Concatenate the array part of a table into a string
- `type = rtable.type(object)` – An implementation of the type library function that returns “table” if the object is a restricted table
- `result = rtable.rtgsub (s, pattern, repl, n)` – Replace a pattern within a string globally (this is also available via the string scope)

For all of the above, the table parameter may be a normal table or restricted table.

## Restricted Closures

In order to allow user code to perform actions normally reserved only for the Blizzard UI, the restricted execution infrastructure creates closures (compiled Lua functions) out of your Lua source snippets, locked inside a carefully controlled sandbox. There's actually a different sandbox for each unique function signature, and within that sandbox is a cache for the closures corresponding to each valid function body.

The first time a given snippet of code is to be run for a specific signature, it's screened for validity and then compiled into the closure. If there are any compilation errors or problems then an error will be output. Multiple uses of the same snippet of code will then use the cached restricted closure, so a reasonably high degree of performance is maintained.

In order to maintain the boundaries of the sandbox there are a couple of language elements that are prohibited within a snippet:

### No function / closure creation

You cannot create functions and closures of your own inside the snippet. The implementation of this restriction is that the string **function** cannot appear in a function body.

You can still achieve many of the same goals as functions using the **control:Run("snippet", ...)**, **control:RunFor(frameHandle, "snippet", ...)**, and **control:RunSnippet("snippetId", ...)** mechanisms, which allow for arbitrary function bodies to be compiled and executed against a protected frame handle.

### No direct table creation

Since unrestricted tables are not allowed within the snippet, you cannot directly create Lua tables. The implementation of this restriction is that neither the **{** nor **}** characters are allowed to appear in the function body.

To create restricted tables you can use the **table.new(...)** function (also available as **newtable(...)**), which can optionally take a list of values that will be inserted, array-initializer style, into the newly created table.

## Frame Handles

The secure handlers are largely oriented around the manipulation of frames, but these manipulations have some limits on which kinds of frames can be used (non-protected frames are off-limits during combat), and what can be done to them.

Some frame handles are automatically created when executing snippets, for example the **self** parameter is the frame handle to the corresponding frame, and there's a default **owner** global variable created in each header's global environment that is a handle to the header itself. Handles can be obtained for other frames as well through the following mechanisms:

- Via the return values from the handle:**GetParent()**, handle:**GetChildren()**, and handle:**GetChildList(tbl)** methods.
- As attributes set from other snippets with access to frame handles.
- By using the **header:SetFrameRef("label", frame)** method on a secure header frame to create a handle for a frame, then using the handler:**GetFrameRef("label")** method to retrieve the frame handle for that frame via the same label.

In WoW 4.0 the special purpose **control** handle was deprecated and the functionality moved to be available on frame handles where the frame is explicitly protected. The special **control** global is still maintained but it is now just the frame reference to the corresponding owner frame.

Frame handles can be called from both secure and insecure code (though only created via secure code), and provide the following methods:

### Normal Read Methods

These methods have the same names and behavior as standard UI Frame methods, and read data from the referenced frame.

**name = handle:GetName()**

---

Gets the name of the frame.

**type = handle:GetObjectType()**

---

Gets the type of the frame.

**isType = handle:IsObjectType("type")**

---

Tests if the frame implements a type.

**id = handle:GetID()**

---

Gets the ID of the frame.

**shown = handle:IsShown()**

---

Determines if the frame is shown.

***visible = handle:IsVisible()***

---

Determines if the frame is visible.

***width = handle:GetWidth()***

---

Gets the width of the frame.

***height = handle:GetHeight()***

---

Gets the height of the frame.

***left, bottom, width, height = handle:GetRect()***

---

Gets the bounding rectangle of the frame.

***scale = handle:GetScale()***

---

Gets the scale of the frame.

***scale = handle:GetEffectiveScale()***

---

Gets the effective scale of the frame.

***level = handle:GetFrameLevel()***

---

Gets the frame level of the frame.

***strata = handle:GetFrameStrata()***

---

Gets the strata of the frame.

***isProtected, isExplicitlyProtected = handle:IsProtected()***

---

Tests if the frame is protected.

***count = handle:GetNumPoints()***

---

Gets the number of anchor points defined for the frame.

***enabled = handle:IsKeyboardEnabled()***

---

Determines if the frame is enabled for keyboard input.

***enabled = handle:IsMouseEnabled()***

---

Determines if the frame is enabled for mouse input.

## Limited Read Methods

These methods have the same names as standard Frame methods, but have some restrictions on what they may query or return.

***value = handle:GetAttribute(name)***

---

Gets the value of an attribute under the following conditions; The name must be a string and not begin with an underscore,. The returned value can be of type **string**, **number**, **boolean**, or be a **frame handle**. Any other values are mapped to nil.

***parentHandle = handle:GetParent()***

---

Gets the handle to the frame's parent.

***childHandle1, childHandle2, ... = handle:GetChildren()***

---

Gets the handles of the frame's children (unprotected children are ignored in combat).

***point, relhandle, relpoint, xofs, yofs = handle:GetPoint(index)***

---

Gets the parameters of a specific anchor point.

## Handle-Specific Read Methods

These methods are specific to frame handles and are provided for convenience or to access information that might be restricted by the limited read methods.

---

### ***x, y = handle:GetMousePosition()***

Gets the current cursor position relative to the frame's size and location, such that  $x=0, y=0$  is the lower left hand corner, and  $x=1, y=1$  is its upper right hand corner. If the mouse is not over the frame then nil is returned.

---

### ***flag = handle:IsUnderMouse(recursive)***

Determines if the mouse is above the frame, or if recursive is specified, above the frame or any of its visible and protected children (The handle frame is always considered regardless of its visibility).

---

### ***tbl = handle:GetChildList(tbl)***

Appends the handles of the frame's children to the specified array table (unprotected children are ignored in combat), returning the table afterwards. The table is not emptied beforehand.

---

### ***frameRef = handle:GetFrameRef(label)***

Gets a frame reference that was previously created with a call to the header's **SetFrameRef** method, or the global **SecureHandlerSetFrameRef** function.

---

### ***value = handle:GetEffectiveAttribute(name, button, prefix, suffix)***

Gets the value of an attribute with modified search rules under the following conditions; The name must be a string and not begin with an underscore. The prefix and suffix may be nil, or strings, and if present the prefix may not begin with an underscore. The returned value can be of type **string**, **number**, **boolean**, or be a **frame handle**. Any other values are mapped to nil.

## Normal Update Methods

These methods have the same names and behavior as standard UI Frame methods, and update the referenced frame.

---

### ***handle:Show(skipAttr)***

Shows the frame (also sets the "statehidden" attribute to nil unless *skipAttr* is true).

---

### ***handle:Hide(skipAttr)***

Hides the frame (also sets the "statehidden" attribute to true unless *skipAttr* is true).

---

### ***handle:SetID(id)***

Sets the frame's identifier (if id is not numeric then it is replaced with 0).

---

### ***handle:SetWidth(width)***

Sets the frame's width.

---

**handle:SetHeight(*height*)**

Sets the frame's height.

---

**handle:SetScale(*scale*)**

Sets the frame's scale.

---

**handle:SetAlpha(*alpha*)**

Sets the frame's alpha (this is provided largely for convenience, alpha isn't protected (which is why there's no corresponding **handle:GetAlpha()**).

---

**handle:ClearAllPoints()**

Clears all anchor points from the frame.

---

**handle:Raise()**

Raise the frame above its peers.

---

**handle:Lower()**

Lower the frame below its peers.

---

**handle:SetFrameLevel(*level*)**

Sets the frame level of the frame.

---

**handle:SetFrameStrata(*strata*)**

Sets the strata of the frame.

---

**handle:EnableKeyboard(*enabled*)**

Sets whether keyboard input is enabled for the frame.

---

**handle:EnableMouse(*enabled*)**

Sets whether mouse input is enabled for the frame.

---

**handle:Enable()**

Enables the button (only valid for frames that are Button derivatives)

---

**handle:Disable()**

Disables the button (only valid for frames that are Button derivatives)

## Limited Update Methods

These methods have the same names as standard Frame update methods, but have some restrictions on what frames or values they may take as input.

---

**handle:SetPoint(*point*, *relframe*[, *relpoint*][, *xofs*, *yofs*])**

Sets an anchoring point on the frame. The *relframe* parameter must be a frame handle for an explicitly protected frame or nil (for the screen) or one of the special string token values "\$screen" for the screen, "\$cursor" for the cursor (in which case *relpoint* is ignored, but it must have been valid to begin with), or "\$parent" for the frame's parent.

---

**handle:SetAllPoints(*relframe*)**

---

Sets all anchoring points on the frame to those of the relative frame, which must be either a frame handle, nil or “\$screen” for the screen, or “\$parent” for the frame’s parent.

---

**handle:SetAttribute(*name, value*)**

---

Sets an attribute on the frame, the *name* must be a string value that does not begin with an underscore. The *value* must be a **string, number, boolean, nil**, or a frame handle.

---

**handle:SetParent(*parentFrame*)**

---

Sets the frame’s parent to the specified *parentFrame*, which must be a frame handle of an explicitly protected frame or nil for the screen.

## Handle-Specific Update Methods

These methods are specific to frame handles and are provided for convenience or to make changes that are not normally accessed as methods and not otherwise available in the restricted environment.

---

**handle:SetBindingClick(*priority, key, name[, button]*)**

---

Sets an override click binding owned by the frame. The name parameter may be a string name or else a frame handle for a named frame.

---

**handle:SetBinding(*priority, key, action*)**

---

Sets an override binding owned by the frame.

---

**handle:SetBindingSpell(*priority, key, spell*)**

---

Sets an override spell binding owned by the frame.

---

**Handle:SetBindingMacro(*priority, key, macroName*)**

---

Sets an override macro binding owned by the frame.

---

**handle:SetBindingItem(*priority, key, item*)**

---

Sets an override item binding owned by the frame.

---

**handle:ClearBinding(*key*)**

---

Clears an override binding owned by the frame.

---

**handle:ClearBindings()**

---

Clears all override bindings owned by the frame.

---

**handle:RegisterAutoHide(*duration*)**

---

Registers a visible frame to be automatically hidden *duration* seconds after the mouse stops hovering over it (at which point the registration is cleared).

---

**handle:UnregisterAutoHide()**

---

Cancels any pending auto hide registration for this frame.

### **handle:AddToAutoHide(*otherFrame*)**

---

Adds the boundaries of the frame *otherFrame* to the hover region of the current frame. Can only be called immediately after **RegisterAutoHide**.

## **Handle Control Methods**

These methods used to belong to the control handle, but have been unified such that they're available for any frame handle as long as the frame is explicitly protected.

The significant functions provided by the control methods support:

- Executing other snippets, to allow code modularization.
- Firing update snippets attached to children of the header frame.
- Executing non-secure methods on the header to support visual updates.

### **... = handle:Run(*body*, ...)**

---

Invokes a snippet whose body is specified by the *body* parameter against the header with a signature of (self,...), where self is the header. Returns the values returned by the snippet (with any values that are not simple **nil**, **string**, **boolean**, **number** values replaced with nil)

### **... = handle:RunFor(*frameHandle*, *body*, ...)**

---

Invokes a snippet whose body is specified by the *body* parameter against the header with a signature of (self, ...) where self is the frame specified by *frameHandle*, which can be nil. Returns the values returned by the snippet (with any values that are not simple **nil**, **string**, **boolean**, **number** values replaced with nil)

### **... = handle:RunAttribute(*snippetAttr*, ...)**

---

Invokes a snippet fetched from the header attribute specified by *snippetAttr* with a signature of (self, ...) where self is the header. Returns the values returned by the snippet (with any values that are not simple **nil**, **string**, **boolean**, **number** values replaced with nil)

### **handle:CallMethod("methodName", ...)**

---

Calls a method on the header frame (the method cannot be secure code) passing in any parameters provided. There are no return values from this control function, nor will any errors that occur while running the method be passed back to the control handle.

### **handle:ChildUpdate(*scriptid*, *message*)**

---

Iterates over all of the protected children of the header and checks for the attribute “\_childupdate-<*scriptid*>”, if that is nil or *scriptid* is nil then it falls back to “\_childupdate”. If the attribute has a string value then it’s processed as a snippet and invoked with parameters (self, *scriptid*, *message*) where self is the child, and *scriptid* and *message* are the values from the call.

**ATTRIBUTE:** **\_childupdate-<*scriptid*>, \_childupdate**

*These attributes are read from each child, the attribute without a scriptid is only used if there is no attribute defined with the scriptid.*

**PARAMETERS:** **(*self*, *scriptid*, *message*)**

***self*** the child's frame's handle  
***scriptid*** the scriptid provided  
***message*** the message provided

**RETURNS:** none

## Wrapping Script Handlers

One of the most powerful features of the secure handlers is the ability to act in response to script actions occurring on other Frames (or on the Header itself), as well as altering the execution of those handlers in controlled ways. This feature is called **Wrapping**, and it works by replacing the script handler on a frame with a special replacement that runs some code in the restricted environment of a handler, then calls the original handler, and then optionally runs more code in the restricted environment.

Wrappers are created via the header method ***header:WrapScript(frame, "script", "preBody" [, "postBody"])*** or via the global function ***SecureHandlerWrapScript(frame, "script", header, "preBody" [, "postBody"])***.

- ***header*** is the frame that is acting as the secure header, this must be an explicitly protected frame, and identifies the global environment to use.
- ***frame*** is the frame whose handler is to be wrapped.
- ***"script"*** is the name of the handler, such as "OnClick" to wrap, it must be a valid handler for *frame*, but there doesn't need to be a handler set there.
- ***"preBody"*** is a string containing the Lua snippet to run before calling the wrapped handler, it is required.
- ***"postBody"*** is an optional string containing a Lua script which might be run after calling the wrapped handler.

Each supported script handler type (OnClick, etc) has particular parameters that are passed to the "pre" and "post" snippets, as well as specific return values relevant to the handler's purpose. All of the wrappers do have the same three basic phases:

1. Call the "pre" snippet that is defined by the ***"preBody"*** parameter to the wrap, passing in details of the triggering event, and obtaining at least one return value. Every pre snippet has some mechanism for exiting immediately without running the wrapped handler, and every pre snippet can also request that a snippet be run after the wrapped handler, if supplied.
2. Call the wrapped handler normally, passing all of the triggering event details. In some handlers there are mechanisms to change some of those details. If the wrapped handler fails then an error is displayed but execution continues normally as if it had not.
3. If a ***"postBody"*** parameter was supplied and the pre handler has requested that it be executed, then execute the "post" snippet it defines, passing both the details of the action as well as a single 'message' value from the "pre" snippet (the presence of which is what triggers the "post" snippet to be run in the first place)

## Click Wrappers

### OnClick, OnDoubleClick, PreClick, PostClick

All of these click mechanisms have a common specification for their pre and post snippets, as well as the changes that can be made to the execution of the wrapped script.

#### Wrapped Click “pre” Snippet

---

<b>PARAMETERS:</b> ( <i>self, button, down</i> )	
<i>self</i>	the wrapped frame’s handle
<i>button</i>	the button that was pressed or released
<i>down</i>	flag indicating if this was a press or release (if relevant)
<b>RETURNS:</b> <i>newbutton, message</i>	
<i>newbutton</i>	an optional override for the button to use for the rest of the execution. <ul style="list-style-type: none"><li>• If this is <b>nil</b> then the original button is used,</li><li>• if this is <b>false</b> then execution ends immediately (neither the wrapped handler nor the post snippet are executed),</li><li>• otherwise this is converted to a string and replaces button for the call to the wrapped handler and post snippet</li></ul>
<i>message</i>	a message to activate the “post” snippet. If this is not <b>nil</b> and a “post” snippet is specified, then it will be called and passed this message

#### Wrapped Click “post” Snippet

---

In the event that the “pre” snippet provided a non-**nil** *message*, the *newbutton* was not false, and the post snippet is present, it will be executed after the wrapped handler.

<b>PARAMETERS:</b> ( <i>self, message, button, down</i> )	
<i>self</i>	the wrapped frame’s handle
<i>message</i>	the message from the “pre” snippet
<i>button</i>	the button that was pressed or released
<i>down</i>	flag indicating if this was a press or release (if relevant)
<b>RETURNS:</b>	none

## Drag Wrappers

### OnDragStart, OnReceiveDrag

These wrappers support drag and drop functionality for frames. The wrapper “pre” script can bypass normal execution of the wrapped handler and perform its own pick-up/replace actions instead.

#### Wrapped Drag “pre” Snippet

---

**PARAMETERS:** (*self*, *button*, *kind*, *value*, ...)

*self* the wrapped frame’s handle  
*button* the button which was pressed to start dragging (always **nil** for receive drag)  
*kind* the kind of thing on the cursor  
*value* details of the thing on the cursor  
... any other returns from **GetCursorInfo()**

**RETURNS:** [*“clear”*,] *kind*, *value*, ...

*“clear”* if the first result is the string “clear” then the cursor is cleared before processing the rest of the result.  
*kind* the kind of thing to pick up (**nil** for none – see below)  
*value* details of the thing to pick up  
... any other details required for pick up

In the return value for these wrappers, *kind* is treated specially. If *“clear”* is NOT specified then:

- If *kind* is nil, then control passes on to the wrapped handler normally.
- If *kind* is “message” then *value* is stored as a message for the optional “post” snippet, and control passed onto the wrapped handler normally.
- Otherwise the results are processed as for the normal header drag templates, used to alter what is on the cursor, then the wrapper exits without calling the original wrapped function or the post snippet.

## Wrapped Drag “post” Snippet

---

In the event that the “pre” snippet provided a non-**nil** message value (*kind == “message”* and *value != nil*), and the “post” snippet is present, it will be executed after the wrapped handler.

**PARAMETERS:** (*self, message, button*)

*self* the wrapped frame’s handle  
*message* the message from the “pre” snippet  
*button* the button which was pressed to start dragging (always **nil** for receive drag)

**RETURNS:** none

## Simple Wrappers

### OnShow, OnHide, OnEnter, OnLeave

These are a collection of wrappers for simple notification-style handlers. These are called with only the frame handle of the wrapped frame. Each of these wrappers can choose to bypass execution of the wrapped handler. The OnEnter and OnLeave handlers are only triggered by changes due to mouse movement (not frame obscuring changes)

#### Wrapped Simple “pre” Snippet

---

<b>PARAMETERS:</b> ( <i>self</i> )
<i>self</i> the wrapped frame’s handle
<b>RETURNS:</b> <i>allow, message</i>
<i>allow</i> an optional flag to control whether to proceed with handler execution. If the value is the boolean <b>false</b> ( <b>nil</b> does not count) then the wrapped handler is not executed, nor does the “post” snippet, if present.
<i>message</i> a message to activate the “post” snippet. If this is not <b>nil</b> and a “post” snippet is specified, then it will be called and passed this message

#### Wrapped Simple “post” Snippet

---

In the event that the “pre” snippet provided a non-**nil** *message*, and *allow* was not false, and the “post” snippet is present, it will be executed after the wrapped handler.

<b>PARAMETERS:</b> ( <i>self, message</i> )
<i>self</i> the wrapped frame’s handle
<i>message</i> the message from the “pre” snippet
<b>RETURNS:</b> none

## Other Specialized Wrappers

### OnMouseWheel

This wrapper allows interaction with the mouse wheel, which may optionally suppress execution of the original wrapped handler.

#### Wrapped Mouse Wheel “pre” Snippet

---

**PARAMETERS:** (*self*, *offset*)

*self* the wrapped frame’s handle  
*offset* the scroll offset received from the mouse wheel

**RETURNS:** *allow*, *message*

*allow* an optional flag to control whether to proceed with handler execution. If the value is the boolean **false** (**nil** does not count) then the wrapped handler is not executed, nor does the “post” snippet, if present.  
*message* a message to activate the “post” snippet. If this is not **nil** and a “post” snippet is specified, then it will be called and passed this message

#### Wrapped Mouse Wheel “post” Snippet

---

In the event that the “pre” snippet provided a non-**nil** *message*, and *allow* was not false, and the “post” snippet is present, it will be executed after the wrapped handler.

**PARAMETERS:** (*self*, *message*, *offset*)

*self* the wrapped frame’s handle  
*message* the message from the “pre” snippet  
*offset* the scroll offset received from the mouse wheel

**RETURNS:** none

## OnAttributeChanged

A wrapper that provides the ability to detect and react to changes in the value of any attribute whose name does not begin with an underscore. The “pre” script can also optionally suppress execution of the wrapped handler.

### Wrapped Attribute “pre” Snippet

---

<b>PARAMETERS:</b> ( <i>self, name, value</i> )	
<i>self</i>	the wrapped frame’s handle
<i>name</i>	the name of the attribute that changed
<i>value</i>	the new value for the attribute
<b>RETURNS:</b> <i>allow, message</i>	
<i>allow</i>	an optional flag to control whether to proceed with handler execution. If the value is the boolean <b>false</b> ( <b>nil</b> does not count) then the wrapped handler is not executed, nor does the “post” snippet, if present.
<i>message</i>	a message to activate the “post” snippet. If this is not <b>nil</b> and a “post” snippet is specified, then it will be called and passed this message

### Wrapped Attribute “post” Snippet

---

In the event that the “pre” snippet provided a non-**nil** *message*, and *allow* was not false, and the “post” snippet is present, it will be executed after the wrapped handler.

<b>PARAMETERS:</b> ( <i>self, message, name, value</i> )	
<i>self</i>	the wrapped frame’s handle
<i>message</i>	the message from the “pre” snippet
<i>name</i>	the name of the attribute that changed
<i>value</i>	the new value for the attribute
<b>RETURNS:</b> none	

## Integration With Other UI Components

### State Drivers

The WoW UI provides a mechanism to notify frames of certain occurrences in the player's environment, specified using the macro conditional format. This mechanism can be used to drive execution of code in the secure headers by utilizing any of the attribute-sensitive mechanisms:

- The **SecureHandlerStateTemplate** if the header only needs awareness of the state attributes.
- The **SecureHandlerAttributeTemplate** if the header wishes to be aware of other attributes in addition to the state attributes.
- The **OnAttributeChanged** wrapper to make frames other than just the header sensitive to these notifications.

To manage these notifications the Blizzard UI code provides:

#### **RegisterStateDriver(*frame*, "*stateid*", "*values*")**

---

This function is called with a frame and a string state identifier, as well as a value description string written in the same format as the conditionals for a macro. The conditional value string is re-evaluated periodically by the state driver, and whenever there's a change in the result of the evaluation, the "state-*stateid*>" attribute on the frame is set to the new value (Note that the *stateid* "visibility" is reserved for internal use to show and hide the frame)

e.g.:

```
RegisterStateDriver(myHeaderFrame, "targgtype", "[noexists] none [help] help [harm] harm");
```

#### **UnregisterStateDriver(*frame*, "*stateid*")**

---

Removes any state notification associated with the specified frame and state identifier.

There's also a mechanism provided if the only item of interest is whether the unit associated with a frame (by way of its "**unit**" attribute) exists or not, which is accessed by the following Blizzard UI functions:

#### **RegisterUnitWatch(*frame*, *asState*)**

---

Sets up the frame to monitor for unit existence, if you pass a true value as *asState* then whenever the unit existence changes the attribute "state-unitexists" is changed between the values true and false.

#### **UnregisterUnitWatch(*frame*)**

---

Removes the unit watch registration for the specified frame.

## Hover Manager

Having popup and fly-out menus hide themselves a short time after the mouse leaves them is a fairly standard and useful UI feature, but protected frames cannot be manipulated in combat by normal AddOn code, and secure handler code cannot access time delays or the current time. In order to address this problem the Blizzard UI code provides an API mechanism for a frame that has just been shown to request that it be automatically be hidden once the mouse leaves it.

These functions can be used by normal AddOn code out of combat, and are also available to secure handler code via a set of restricted frame methods with the same names.

**IMPORTANT: These are “one-shot” registrations, a frame must request auto hiding when it is shown, and the registration expires if the frame is hidden (automatically or otherwise), or when it is moved.**

### **RegisterAutoHide(*frame*, *duration*)**

---

This function is called with a frame and a duration in seconds, and if the frame is presently visible it will be registered to monitoring and automatic hiding by the driver. Once registered, once the mouse has entered the frame (or other frames added to it), and then left for at least duration seconds, the frame will be automatically hidden. If the mouse re-enters the frame(s) then the timer is reset.

### **UnregisterAutoHide(*frame*)**

---

Removes any active auto hide registration for the frame.

### **AddToAutoHide(*frame*, *otherFrame*)**

---

Adds another frame’s boundaries to the area considered for hover inclusion for a frame that has just been registered for auto hiding. This can only be called for a *frame* that has been registered for auto hide, and must be called immediately following the initial registration (before the next OnUpdate round).

## Examples?

There are now a number of AddOns in the wild which use the new SecureHandlers mechanisms.

## Reference Tables

### Frame Handle Method Reference

<i>"name"</i> = <i>handle</i> :GetName()	<i>enabled</i> = <i>handle</i> :IsKeyboardEnabled()
<i>"type"</i> = <i>handle</i> :GetObjectType()	<i>enabled</i> = <i>handle</i> :IsMouseEnabled()
<i>isType</i> = <i>handle</i> :IsObjectType( <i>"type"</i> )	<i>scale</i> = <i>handle</i> :GetScale()
<i>id</i> = <i>handle</i> :GetID()	<i>scale</i> = <i>handle</i> :GetEffectiveScale()
<i>shown</i> = <i>handle</i> :IsShown()	<i>level</i> = <i>handle</i> :GetFrameLevel()
<i>visible</i> = <i>handle</i> :IsVisible()	<i>strata</i> = <i>handle</i> :GetFrameStrata()
<i>width</i> = <i>handle</i> :GetWidth()	<i>value</i> = <i>handle</i> :GetAttribute( <i>"name"</i> )
<i>height</i> = <i>handle</i> :GetHeight()	<i>parentHandle</i> = <i>handle</i> :GetParent()
<i>left, bottom, width, height</i> = <i>handle</i> :GetRect()	<i>flag</i> = <i>handle</i> :IsUnderMouse( <i>recursive</i> )
<i>x, y</i> = <i>handle</i> :GetMousePosition()	<i>tbl</i> = <i>handle</i> :GetChildList( <i>tbl</i> )
<i>count</i> = <i>handle</i> :GetNumPoints()	<i>frameRef</i> = <i>handle</i> :GetFrameRef( <i>"label"</i> )
<i>isProtected, isExplicitlyProtected</i> = <i>handle</i> :IsProtected()	
<i>childHandle1, childHandle2, ...</i> = <i>handle</i> :GetChildren()	
<i>point, relhandle, relpoint, xofs, yofs</i> = <i>handle</i> :GetPoint( <i>index</i> )	
<i>value</i> = <i>handle</i> :GetEffectiveAttribute( <i>"name"</i> , <i>"button"</i> , <i>"prefix"</i> , <i>"suffix"</i> )	

<i>handle</i> :Show( <i>skipAttr</i> )	<i>handle</i> :ClearAllPoints()
<i>handle</i> :Hide( <i>skipAttr</i> )	<i>handle</i> :SetAllPoints( <i>relframe</i> )
<i>handle</i> :SetID( <i>id</i> )	<i>handle</i> :Raise()
<i>handle</i> :SetWidth( <i>width</i> )	<i>handle</i> :Lower()
<i>handle</i> :SetHeight( <i>height</i> )	<i>handle</i> :Enable()
<i>handle</i> :SetScale( <i>scale</i> )	<i>handle</i> :Disable()
<i>handle</i> :SetAlpha( <i>alpha</i> )	<i>handle</i> :EnableKeyboard( <i>enabled</i> )
<i>handle</i> :SetFrameLevel( <i>level</i> )	<i>handle</i> :EnableMouse( <i>enabled</i> )
<i>handle</i> :SetFrameStrata( <i>strata</i> )	<i>handle</i> :SetParent( <i>parentFrame</i> )
<i>handle</i> :RegisterAutoHide( <i>duration</i> )	<i>handle</i> :SetAttribute( <i>"name"</i> , <i>value</i> )
<i>handle</i> :UnregisterAutoHide()	<i>handle</i> :SetBinding( <i>priority</i> , <i>"key"</i> , <i>"action"</i> )
<i>handle</i> :AddToAutoHide( <i>otherframe</i> )	<i>handle</i> :SetBindingSpell( <i>priority</i> , <i>"key"</i> , <i>"spell"</i> )
<i>handle</i> :ClearBinding( <i>key</i> )	<i>handle</i> :SetBindingItem( <i>priority</i> , <i>"key"</i> , <i>item</i> )
<i>handle</i> :ClearBindings()	
<i>handle</i> :SetPoint( <i>"point"</i> , <i>relframe</i> [, <i>"relpoint"</i> ] [, <i>xofs</i> , <i>yofs</i> ])	
<i>handle</i> :SetBindingClick( <i>priority</i> , <i>"key"</i> , <i>"name"</i> [, <i>"button"</i> ])	
<i>handle</i> :SetBindingMacro( <i>priority</i> , <i>"key"</i> , <i>"macroName"</i> )	

<i>... = handle</i> :Run( <i>"body"</i> , ...)	<i>handle</i> :CallMethod( <i>"name"</i> , ...)
<i>... = handle</i> :RunFor( <i>frameHandle</i> , <i>"body"</i> , ...)	<i>handle</i> :ChildUpdate( <i>"scriptId"</i> , <i>message</i> )
<i>... = handle</i> :RunAttribute( <i>"snippetAttr"</i> , ...)	

## Header Snippet Attribute Reference

Attribute	Parameters	Returns
<code>_onstate-&lt;stateid&gt;</code>	self, stateid, newstate	<i>none</i>
<code>_onattributechanged</code>	self, name, value	<i>none</i>
<code>_onclick</code>	self, button, down	<i>none</i>
<code>_ondoubleclick</code>	self, button	<i>none</i>
<code>_onshow</code>	self	<i>none</i>
<code>_onhide</code>	self	<i>none</i>
<code>_onenter</code>	self	<i>none</i>
<code>_onleave</code>	self	<i>none</i>
<code>_ondragstart</code>	self, button, kind, value, ...	["clear",] kind, value, ...
<code>_onreceivedrag</code>	self, button, kind, value, ...	["clear",] kind, value, ...
<code>_onmousewheel</code>	self, offset	<i>none</i>

## Child Snippet Attribute Reference

Attribute	Parameters	Returns
<code>_childupdate</code>	self, scriptid, message	<i>none</i>
<code>_childupdate-&lt;scriptid&gt;</code>	self, scriptid, message	<i>none</i>

## Wrapper Snippet Reference

Scripts	Snippet	Parameters	Returns
<code>&lt;OnClick&gt;</code> <code>&lt;PreClick&gt;</code> <code>&lt;PostClick&gt;</code>	pre	self, button, down	newbutton, message
	post	self, message, button, down	<i>none</i>
<code>&lt;OnDoubleClick&gt;</code>	pre	self, button	newbutton, message
	post	self, message, button	<i>none</i>
<code>&lt;OnDragStart&gt;</code> <code>&lt;OnReceiveDrag&gt;</code>	pre	self, button, kind, value, ...	["clear",] kind, value, ...
	post	self, message, button	<i>none</i>
<code>&lt;OnShow&gt;</code> <code>&lt;OnHide&gt;</code> <code>&lt;OnEnter&gt;</code> <code>&lt;OnLeave&gt;</code>	pre	self	allow, message
	post	self, message	<i>none</i>
<code>&lt;OnMouseWheel&gt;</code>	pre	self, offset	allow, message
	post	self, message, offset	<i>none</i>
<code>&lt;OnAttributeChanged&gt;</code>	pre	self, name, value	allow, message
	post	self, message, name, value	<i>none</i>

—  
\_childupdate · 25  
\_childupdate-<scriptid> · 25  
\_onattributechanged · 7  
\_onclick · 7  
\_ondblclick · 8  
\_ondragstart · 12  
\_onenter · 11  
\_onhide · 10  
\_onleave · 11  
\_onmousedown · 9  
\_onmouseup · 9  
\_onmousewheel · 8  
\_onreceivedrag · 13  
\_onshow · 10  
\_onstate-<stateid> · 6

## “

“post” snippet · 26  
“pre” snippet · 26

## \$

\$cursor · 22  
\$parent · 22, 23  
\$screen · 22, 23

## A

AddToAutoHide  
*function* · 34  
*method* · 24

## C

CallMethod · 24  
ChildUpdate · 25  
ClearAllPoints · 22  
ClearBinding · 23  
ClearBindings · 23  
Control Handle · 24

## D

Disable · 22

## E

Enable · 22  
EnableKeyboard · 22  
EnableMouse · 22  
Execute · 5

## F

Frame Handles · 18

## G

GetActionInfo · 14  
GetAlpha  
*lack of* · 22  
GetAttribute · 19  
GetChildList · 21  
GetChildren · 20  
GetEffectiveAttribute · 21  
GetEffectiveScale · 19  
GetFrameLevel · 19  
GetFrameRef · 21  
GetFrameStrata · 19  
GetHeight · 19  
GetID · 18  
GetManagedEnvironment · 16  
GetMousePosition · 21  
GetName · 18  
GetNumPoints · 19  
GetObjectType · 18  
GetParent · 19  
GetPoint · 20  
GetRect · 19  
GetScale · 19  
GetWidth · 19

## H

Hide · 21  
Hover Manager · 34

## I

ipairs · 15  
IsKeyboardEnabled · 19  
IsMouseEnabled · 19  
IsObjectType · 18  
IsProtected · 19  
IsShown · 18  
IsUnderMouse · 21  
IsVisible · 19

## L

Lower · 22

## N

newtable · 15, 17  
next · 15

## O

- OnAttributeChanged
  - SecureHandlerAttributeTemplate* · 7
  - SecureHandlerStateTemplate* · 6
  - wrapper* · 32
- OnClick
  - header* · 7
  - wrapper* · 27
- OnDoubleClick
  - header* · 8
  - wrapper* · 27
- OnDragStart
  - header* · 12
  - wrapper* · 28
- OnEnter
  - header* · 11
  - wrapper* · 30
- OnHide
  - header* · 10
  - wrapper* · 30
- OnLeave
  - header* · 11
  - wrapper* · 30
- OnLoad · 5
- OnMouseDown
  - header* · 9
- OnMouseUp
  - header* · 9
- OnMouseWheel
  - header* · 8
  - wrapper* · 31
- OnReceiveDrag
  - header* · 13
  - wrapper* · 28
- OnShow
  - header* · 10
  - wrapper* · 30

## P

- pairs* · 15
- PlayerCanAssist · 14
- PlayerCanAttack · 14
- PlayerInCombat · 14
- PlayerInGroup · 14
- PlayerIsChanneling · 14
- PlayerPetSummary · 14
- PostClick
  - wrapper* · 27
- PreClick
  - wrapper* · 27
- print · 14

## R

- Raise · 22
- RegisterAutoHide
  - function* · 34
  - method* · 23
- RegisterStateDriver · 33
- RegisterUnitWatch · 33
- Restricted Closures · 17
- Restricted Namespace · 14
- Restricted Tables · 15
- rtable · 16
- rtable.concat · 16
- rtable.ipairs · 16
- rtable.maxn · 16
- rtable.next · 16
- rtable.pairs · 16
- rtable.rtgsub · 16
- rtable.type · 16
- rtable.unpack · 16
- rtgsub · 16
- Run · 24
- RunFor · 24
- RunSnippet · 24

## S

- SecureHandlerAttributeTemplate* · 7
- SecureHandlerBaseTemplate* · 5
- SecureHandlerClickTemplate* · 7
- SecureHandlerDoubleClickTemplate* · 8
- SecureHandlerDragTemplate* · 12
- SecureHandlerEnterLeaveTemplate* · 11
- SecureHandlerMouseUpDownTemplate* · 9
- SecureHandlerMouseWheelTemplate* · 8
- SecureHandlerShowHideTemplate* · 10
- SecureHandlerStateTemplate* · 6
- SecureHandlerWrapScript* · 26
- SetAllPoints · 23
- SetAlpha · 22
- SetAttribute
  - frame handle method* · 23
  - header template* · 7
- SetBinding · 23
- SetBindingClick · 23
- SetBindingItem · 23
- SetBindingMacro · 23
- SetBindingSpell · 23
- SetFrameLevel · 22
- SetFrameRef · 5
- SetFrameStrata · 22
- SetHeight · 22
- SetID · 21
- SetParent · 23
- SetPoint · 22
- SetScale · 22
- SetWidth · 21
- Show · 21
- State Drivers · 33
- state-<stateid> · 6

statehidden · 21  
string.rtgsub · 16

## T

table.concat · 16  
table.insert · 16  
table.maxn · 16  
table.new · 16, 17  
table.remove · 16  
table.sort · 16  
table.wipe · 16  
tinsert · 15  
tremove · 15

## U

UnitHasVehicleUI · 14  
unpack · 15  
UnregisterAutoHide  
    *function* · 34  
    *method* · 23  
UnregisterStateDriver · 33  
UnregisterUnitWatch · 33  
UnwrapScript · 6

## W

wipe · 15  
Wrapping Script Handlers · 26  
WrapScript · 5