The Logic API

The logic API is an extension of the basic Lassie Player XML scripting system which allows XML scripts to communicate with the inner workings of the Lassie Player. In short, the logic API is a set of variables and game methods which can be accessed using features such as the <logic> XML script command. By querying the logic API, you can evaluate game status, access engine configuration settings, or call advanced features of the Lassie Player.

Getting Started: The Basics

Included here is a broad primer in the concept of what the logic API is and how it works. The concepts and keywords identified here in this primer will be used as the basis of the rest of this document.

1. Expressions and Returns

When communicating with the logic API, we give it an **expression** and it comes back with an outcome, or "**return**".

An **expression** is any piece of raw text that we send to the logic API. The logic API assumes that we will only provide it expressions that it knows how to read and interpret. After interpreting an expression, the logic API will **return** the outcome to whatever source called the logic API. A malformed expression (uninterpretable text) will be ignored, or may return an erratic outcome to its source.

2. Variables and Methods

At its simplest level, the logic API is composed of **variables** and **methods**. What are those?

A **variable** is a value that is referenced by name. The value of a variable changes based on game conditions, however it is always accessible by a constant name. For example, requesting the "_currentRoom" variable will always yield the ID of the current game room. While the value of the room ID will change based on game conditions, the information is always available by requesting a variable named "_currentRoom". Within a logic expression, variables are cited as the variable name surrounded by square brackets, like so:

```
[_currentRoom]
```

A **method** is a mechanism that performs an action – usually that action returns a value the way a variable does. However, a method is unique in that it accepts parametrized input – meaning that you can give it information to tell it how to behave. These input values are called **parameters**. For example, let's say that you want to find out if a specific inventory contains a specific item. To do so, you would call the "hasItem" method, and give it parameters for the inventory to look in and the item to look for. The method would then perform that check, and return the outcome. Methods are written within an expression as the method name with a list of parameters, all surrounded by curly braces, like so:

```
{methodName:param1,param2,param3}
```

3. Parsing and Evaluating

When we send an expression to the logic API, we can perform two actions: we can **parse** the expression, or **evaluate** it.

When we **parse** an expression, the logic API interprets what it can of the expression, then returns the resulting text. Let's say that our expression references the current game room variable. When we parse the expression, the returned text will be the ID of the current game room. For example:

```
<logic parse="[_currentRoom]"/>
// return >> "roomId"
```

When we **evaluate** an expression, we do the same process as parsing – except that we treat the operation as a question. In essence, "Is this true?". Evaluating an expression will always return as **true** or **false**. In many cases, variables and methods will specifically return a true or false value. Regardless, when evaluating an expression we first parse the text, then ask: "is this true". In the case of requesting the current room – we can't exactly evaluate a literal room ID. After getting the ID of the current room, an evaluation would ask: "is this true?". Since a room id alone is not really a question, the result would always be negative (except in the specific circumstance when the room id is the literal value "true"!). For example:

```
<logic eval="[_currentRoom]"/>
// return >> false
```

4. Queries

In order to make values like the current game room evaluate logically, we must reference the value as part of a **query**.

A **query** is a question which can be evaluated. In the case of the current game room, the room's id alone is not a question. We must phrase the value as part of a question that can be evaluated as true or false. So, we would ask: "Is the game room called 'castle'"? Now the game room reference is part of a logical question which can be evaluated with an outcome of true or false. Here's a basic example, where the "EQ" phrase means "equal to":

```
<logic eval="[_currentRoom] EQ castle"/>
// return >> true (assuming the current room IS called "castle")
```

As you build confidence with making queries, you'll inevitably want to start formulating **compound queries**. A compound query is a question with multiple clauses. Basically, it asks: "Is this AND this true?", or, "Is this OR this true?". There is no limit to the extent of compound queries, except your own ability to keep all the clauses straight! Have a look at the example below, and notice how the multiple logic clauses are grouped within parenthesis ("()") and separated by the word "AND":

```
<logic eval="([_currentRoom] EQ castle) AND ([_currentActor] EQ
knight)"/>
// return >> true (assuming the current room is "castle", and the
current actor is "knight")
```

You may also group compound queries to build upon already compounded logic. To do so, just keep nesting grouped queries inside one another. For example:

```
<logic eval="(([_currentActor] EQ knight) AND
({hasItem:_current,sword})) OR ([_currentActor] EQ king)"/>
// return >> true (assuming that the actor is the knight and is
carrying a sword, or else if the actor is the king)
```

The above example performs two compound queries. The first compound query asks if the current actor is "knight" and if the current inventory contains a sword. That full compound query is nested in parenthesis ("()") to group it, then the outcome is used in a second compound query which asks if the first was true, or if the current actor is "king". Using grouped compound queries allows you to write extremely specific logic statements.

5. Operators

Queries are phrased using **operator**s; you've seen several examples of operators in the examples above.

An **operator** a symbol that indicates the nature of our question. The logic API reads our operator symbols to determine a result. There are two types of operators: there are **comparison operators**, and **logic operators**.

A **comparison operator** is used to compare two values and draw some conclusion about them. The logic API has six comparison operators. They are:

- "**EQ**" : Is A equal to B?
- "**NE**" : Is A NOT equal to B?
- "**GT**" : Is A greater than B? (numbers only)
- "**GE**" : Is A greater than or equal to B? (numbers only)
- "**LT**" : Is A less than B? (numbers only)

- "**LE**" : Is A less than or equal to B? (numbers only)

A **logic operator** is used to evaluate multiple clauses within a compound query. The logic API has only two logical operators, they are:

- "AND" : Is A and B true?
- "OR" : Is A or B true?

With a foundation understanding of those concepts, you're ready to start working with the logic API! Once you start setting up conditional fields within a game framework, you'll find that you can do exponentially more work with less code. Just be careful, with great power comes great responsibility, so make sure you keep track of your questions so that you don't make your game so convoluted that you can no longer follow its logic!

Communicating with the Logic API

The logic API is accessed using one of three methods:

1. The <logic> command

Primary access to the logic API is provided through the <logic> XML command. This allows basic logic queries to evaluate conditional game scripts. The logic command can be used to tap into the logic API in the form of a simple conditional, or an full logic tree.

A simple conditional is a quick evaluation that will withhold XML scripts if the query fails. A simple conditional is written as:

```
<logic eval="[_currentRoom] EQ barnyard">
   <!-- do custom actions here -->
</logic>
```

In the above example, the logic API is being queried to see if the current room id is called "barnyard". If the test evaluated as true (ie: the current room IS called "barnyard") then all child XML scripts nested within the <logic> node would be processed. This is a fast and easy way to create limited custom game behavior. However, it likely that you'll run into a scenario that requires a full logic tree. A logic tree asks several questions in sequence, continuing through the logic tree until one question proves true. A logic tree may also have a catch-all clause that will be called if all specific questions evaluated as false. A full logic tree is composed as:

```
<logic>
  <if eval="[_currentRoom] EQ barnyard">
     <!-- do custom actions here -->
  </if>
  <elseif eval="[_currentRoom] EQ farm">
     <!-- do custom actions here -->
  </elseif>
  <else>
     <!-- do custom actions here -->
  </else>
</logic>
```

In the above example, the logic tree first evaluates the <if> clause to test if the current room is "barnyard", if so, the first set of custom actions are released, and the tree stops processing. If the first test fails, then the tree moves on to evaluate all <elseif> clauses in

sequence until one clause proves true. There is no limit to the number of <elseif> clauses that you include in a tree. Also, you may completely omit <elseif> clauses if they are not needed. If all <elseif> clauses evaluated as false, then the tree defaults to performing all custom actions provided in the <else> node. Again, the <else> node is optional. For the most part, you may free-form a logic tree however you want it, so long as the order of nodes is <if>, <elseif>, <else>.

---

2. The <scriptParse> command

Secondary access to the logic API is provided through the <scriptParse> XML command. Using the <scriptParse> command, you can parse engine values directly into raw XML scripts, which are then processed using the real-time game values. The method can be used to parse real-time game values into an XML command; which allows the command to be parametrized with current game values.

The <scriptParse> command takes no parameters. It only wraps a text representation of raw XML which is sent to the logic API for processing. The contents of the <scriptParse> command should be placed into a CDATA block, which excludes the contents from formal XML parsing. A CDATA block is an XML-native structure, and is written like so:

```
<scriptParse><![CDATA[ Unformatted XML content goes here!
]]></scriptParse>
```

Place Lassie XML commands to be parsed into the CDATA block. Because they are stored within CDATA, it does not matter if the XML commands are malformed. You can use the logic API to parse any formatting needed into the command. To identify fields that should be parsed and replaced by the logic API, use Lassie's logic field open and close symbols:

```
%* logic API expression here *%
```

In full context, the scriptParse method looks like this:

```
<scriptParse>
    <![CDATA[<layerTween targets="_avatar"
to="%*{math:100+100+100}*%,0" seconds="2" ease="elastic:easeOut"
waitForComplete="1"/>]]>
</scriptParse>
```

In the above example, the logic API is being used to parse a custom X value into a layerTween command. All parsing variables and methods of the logic API are available while

parsing the text within logic field delimiters.

---

3) The ActionScript interface

Advanced access to the logic API is available through the Lassie Player's ActionScript interface. The ActionScript interface allows external Flash media to directly communicate with the Lassie Player through ActionScript. Using the ActionScript interface methods, the logic API can be directly queried for current game conditions. The ActionScript interface will have complete documentation written for it in another document.

More About Using Variables
Variables are the first and most basic operation performed by the Logic API when an expression is parsed. Upon reciving a raw expression, the logic parser will first sort through and define all variables within the expression, and then move on to more advanced operations (such as methods).

Referencing variables within a logic expression is extremely easy: you simply cite a variable name enclosed within square brackets ("[]"), like so:

```
"[variableName]"
```

Variables are available from two different sources. One source is the Lassie Player application, which has a permanent collection of pre-defined variables (see the "Variables" section for a complete list of native engine variables). The second source of variables come from the game cache, which is a collection of values that you (the game developer) can declare using XML script commands. See below for further explanation on both variable sources.

Referencing Engine Variables

The Lassie Player provides a core collection of game variables which provide information on the current game configuration. For a full list of available engine variables, see the "Variables" section of this document. You'll notice that all engine-native variables begin with an underscore, such as "_currentRoom". This is a convention to isolate engine variable names from the names of variables a developer may want want to write into the game cache. When declaring your own variables, avoid using a leading underscore so that you never confuse your variable with those naively provided by Lassie.

Let's look at an example of referencing an engine variable. Assume for a moment that the ID of the currently active game room is "barnyard". In that circumstance, we could reference the engine's "_currentRoom" variable to get that information, like so:

```
<logic parse="[_currentRoom]"/>
// resulting value >> "barnyard"
```

In the form of a logical query, the above expression would be written as:

```
<logic eval="[_currentRoom] EQ barnyard">
   <!-- do this custom action in the above circumstance -->
</logic>
// logic result >> TRUE, so the custom action runs.
```

Referencing Variables from the Game Cache

The second source of variables are fields that you, the developer, have inserted into the game cache. If a variable reference is not found within the engine-native values, then the logic parser looks within the game cache to see if the value has been defined there. **Note**: given that engine-native variables take priority over cached game variables, it is important that variable names never collide. To avoid having your variable names conflict with those of the Lassie Player, avoid using a leading underscore in the name of your variables.

Let's create an example scenario of writing a variable into the game cache and pulling it back out through the logic API. To initially insert a variable into the cache, you'll need to use the <cache> XML script command. That looks like this:

```
<cache field="currentQuest" write="gold"/>
```

The above example creates a variable named "currentQuest" within the game cache, and assigns the word "gold" as its value. Now lets use that value within the cache to make a logical assessment within our game. To do so, we'll set up a simple conditional to test the statue of our "currentQuest" variable, like so:

```
<logic eval="[currentQuest] EQ gold">
    <layer target="waterfall" state="goldQuest"/>
</logic>
```

The above example tests if the variable "currentQuest" equals "gold". If the expression proves to be true, then we're calling an additional script command which sets a layer called "waterfall" into a state that is specific to the current quest.

More About Using Methods
Methods are operations which can be called upon to perform an action. While a variable is sufficient to retrieve a value from within game configuration, a method can **get**, **set**, and/or generally manipulate game configuration in numerous ways. However, unlike variables, all methods are static fixtures within the logic API framework. A developer cannot define their own methods.

Let's look at a basic method example. We'll use logic API's "math" method as an example, which performs a basic math computation. A method call is encased in curly braces ("{}"), and is cited as:

```
{methodName:param1,param2,param3}
```

An actual call the logic API's math method looks like this:

```
{math:2+2}
// return >> 4
```

In the above example, the "math" method was called with a single parameter ("2+2"). The logic API parsed the expression, and returned an outcome of "4". Pretty straight forward, right?

Now let's take this concept a step further and tie a method in with a variable. Given that an expression's variables are parsed before methods, you can place variable into a method to have the method run using the variable's value. For example, lets say that we create a "score" variable within the game cache using the following XML command:

```
<cache field="score" write="0"/>
```

Using the above command, we've given the user a blank score value. From here, we can grow that score throughout the game. To perform a basic math operation on the score variable would look like this:

```
{math:[score]+10}
// score value was 0, so this math operation returns >> "10"
```

In the above example, the "score" game variable is first parsed into the expression, then the "math" method is called with a parameter of "0+10", which results in 10. However, it's important to note that we did NOT change the value of the score field during this operation. The score field's value was parse into the expression, however the score field itself was

never changed. To write to the score field, the easiest solution would be to use the "setCacheVar" method, which calls the "math" method to generate its value parameter... yes, methods can be nested inside one another, so a method can be called while parsing another method. To increment the score through the logic API, the following method arrangement could be called:

```
<logic parse="{setCacheVar:score,{math:[score]+10}}"/>
```

In the above example, the logic API is called upon to parse our logic expression. We use the "setCacheVar" command to set a field called "score" with a value that we generate using a the "math" method, which takes the current value of "score" and adds +10 to it. Simple, right? Yes, some programming experience is helpful while writing parse expressions, however with a little practice, anyone can get the hang of it.

Variable Definitions

The following variables are defined by the Lassie Player and are always available to reference.

_lassieVersion

type: string

Specifies the Lassie Player version.

_currentRoom

type: string

Specifies the Id of the currently active game room.

_currentActor

type: string

Specifies the Id of the currently active game actor.

_roomScrollX

type: number

Specifies the current X-position of the room's scroll offset.

_roomScrollY

type: number

Specifies the current Y-position of the room's scroll offset.

_startPosition

type: boolean (true/false)

Specifies the starting position id referenced while loading the current game room. This

starting position determines the starting position of the avatar within the room. This value is available at the time a room's "enter" script is called, so this property may be used to perform conditional room configuration based on where the avatar is starting.

_calledByPosition

type: boolean (true/false)

Specifies if the active script is being called in response to plotting the avatar's layer at its starting position within a room layout. This property is useful for grid node actions that should only trigger while being used as starting placement for the avatar layer, but not while the node is accessed as part of the normal walkable grid.

_calledByGrid

type: boolean (true/false)

Specifies if the active script is being called in response to a puppet reaching the grid node during a normal move sequence.

_gridFrom

type: string (node id)

For use within walkable grid scripts only. This value tracks the id of the node that a puppet came from to reach the current grid node that it's at.

_gridFromAbove

type: boolean (true/false)

For use within walkable grid scripts only. Specifies if the puppet has arrived at the current grid node with a downward-directional view angle (4, 5, or 6; ie: a front-quarter or front directional view).

Example:

```
<logic eval="[_gridFromAbove]">
    <layerSprite target="_avatar" floatBehind="hill"/>
</logic>
```

_gridFromBelow

type: boolean (true/false)

For use within walkable grid scripts only. Specifies if the puppet has arrived at the current grid node with an upward-directional view angle (8, 1, or 2; ie: a back-quarter or back directional view).

_gridFromLeft

type: boolean (true/false)

For use within walkable grid scripts only. Specifies if the puppet has arrived at the current grid node with a rightward-directional view angle (2, 3, or 4; ie: a right back-quarter, side, or front-quarter view).

_gridFromRight

type: boolean (true/false)

For use within walkable grid scripts only. Specifies if the puppet has arrived at the current grid node with a leftward-directional view angle (6, 7, or 8; ie: a left front-quarter, side, or back-quarter view).

_gridFromView

type: number (1-8 representing a directional view)

For use within walkable grid scripts only. Specifies the actual turn view value that the puppet has arrived at the node with.

_gridTo

type: string (node id)

For use within walkable grid scripts only. This value tracks the id of the node that a puppet will be going to from the node that it is currently at.

Methods Definitions

The following methods are defined by the Lassie Player and are always available to call.

---

echo

```
{echo:message}
```

Params:

- *message*: The message to send to the debugger window. This debugger window will automatically open in response to the message if enabled.

Returns:

- no return.

Outputs the specified message in the debug window. The debug window will need to be enabled using the debug command in order for the message to be visible.

---

hasItem

```
{hasItem:collectionId,itemId}
```

Params:

- *collectionId*: The ID of the inventory collection to access.
- *itemId*: The ID of the inventory item to search the collection for.

Returns:

- "true" if the specified item is currently contained within the inventory. Otherwise, "false" is returned.

Tests if an item currently exists within an inventory collection. Method returns the string "true" if the specified item exists within the specified collection at the time of the method call.

---

hasAddedItem

```
{hasAddedItem:collectionId,itemId}
```

Params:

- *collectionId*: The ID of the inventory collection to access.
- *itemId*: The ID of the inventory item to search the collection's history for.

Returns:

- "true" if the specified item has ever been added to the inventory (regardless if it is currently present). Otherwise, "false" is returned.

Tests if a specific item has ever been added to a specific inventory collection; regardless of whether the item is currently present within the collection. The collection's item history is reviewed, so this method will return the string "true" if the specified item has been placed in the inventory at any point during the game.

---

getItemsList

```
{getItemsList:collectionId}
```

Params:
- *collectionId*: The ID of the inventory collection to access.

Returns:
- A comma-separated list of all item IDs contained within the inventory collection.

Returns a comma-seperated list of all item IDs currently contained within the specified inventory collection. This value may be formatted using on of the comma-separated list formatting methods.

---

countItems

```
{countItems:collectionId}
```

Params:
- *collectionId*: The ID of the inventory collection to access.

Returns:
- The number of inventory items contained within the specified collection.

Specifies the number of items currently contained within the specified inventory collection. Ex: if the queried collection has three items, then "3" is returned.

---

getLayerProp

```
{getLayerProp:roomId,layerId,propName}
```

Params:
- *roomId*: The ID of the room that contains the target layer. "_current" may be used to cite the current room.
- *layerId*: The ID of the target layer within its room layout.
- *propName*: The name of the property to read from the target layer state. You may reference any optional property name that may be set using the <layer> XML command (see XML documentation).

Returns:

- The value of the specified layer property. The type of return (number, string, boolean) depends on the type of the accessed property.

Gets the value of a layer property. You may specify any layer property name listed in the "Optional Parameters" section of the <layer> command's XML script documentation. Note that attempting to access a layer's properties before loading its parent room for the first time may yield unexpected results. This command is generally safest to implement within the room of the layer that it targets.

---

getLayerStateProp

```
{getLayerStateProp:roomId,layerId,stateId,propName}
```

Params:

- *roomId*: The ID of the room that contains the target layer. "_current" may be used to cite the current room.
- *layerId*: The ID of the target layer within its room layout.
- *stateId*: The ID of the target layer state. When referencing the avatar layer, use "main" as the state Id.
- *propName*: The name of the property to read from the target layer state. You may reference any optional property name that may be set using the <layerState> XML command (see XML documentation).

Returns:

- The value of the specified layer state property. The type of return (number, string, boolean) depends on the type of the accessed property.

Gets the value of a layer state property. You may get the value of any layer state property listed in the "Optional Parameters" section of the <layerState> command's XML script documentation. Note that attempting to access a layer's properties before loading its parent room for the first time may yield unexpected results. This command is generally safest to implement within the room of the layer that it targets.

---

getTreeTopicEnabled

```
{getTreeTopicEnabled:roomId,treeId,topicId}
```

Params:

- *roomId*: The ID of the room that contains the target tree. "_current" may be used to cite the current room.
- *treeId*: The ID of the target tree within its room layout.
- *topicId*: The ID of the target tree topic.

Returns:

- True or false depending on the enabled status of the topic line, or false if the specified topic does not exist.

Gets the enabled status of a specific tree topic line. Note that attempting to access a tree's properties before loading its parent room for the first time may yield unexpected results. This command is generally safest to implement within the room of the layer that it targets.

---

math

`{math:expression}`

Params:

- *expression*: A mathematical expression string.

Returns:

- The numeric outcome of the mathematical equation, or "null" if the math parsing failed.

Performs a mathematical computation. The math expression may only include numbers and the four basic math operators: + (add), - (subtract), * (multiply), / (divide). When a linear mathematical operation processes, all multiplication and divisions are performed first, then additions and subtractions. To ground math operation into a different processing order, nest multiple {math} methods into granular equations. Cache variables may be inserted into mathematical expressions, although you must make sure the variable is set to a numeric value.

Examples:

```
<logic parse="{math:1+2*2}"/>
//-- returns as "5" (2 * 2 = 4 + 1 = 5)

<logic parse="{math:{math:1+2}*2}"/>
//-- returns as "6" (1 + 2 = 3 * 2 = 6)

<logic parse="{math:1+[myCacheVar]}"/>
```

---

setCacheVar

`{setCacheVar:fieldName,value}`

Params:

- *fieldname*: The name of the variable field to set within the cache.
- *value*: The value to assign to the field. If the field already existed within the cache, then its value will be replaced by this new value.

Returns:

- no return value.

Writes a variable with the specified field name into the cache with the specified value.

---

renderCacheList

```
{renderCacheList:fieldName,sort,delimiter}
```

Params:

- *fieldname*: The name of the comma-separated list variable to access within the cache.
- *sort*: Specify "1" or "0" (true or false)
- *delimiter*: Optional. This defines the character(s) to separate list elements with. If no custom character is specified, then the list will be separated with commas.

Returns:

- A rendered string of all list items, sorted alphabetically (if enabled) and separated by the specified delimiter character.

Renders a comma-separated list field from the cache. The rendering may be sorted and have commas replaced by a different delimiter character. The source value within the cache is NOT modified. This is useful while managing a list of status keys. While the items may be added to the list in any order, you can use this method to render them into a consistent sorted order.

Example:

```
//-- assume the value of "myList" is "coconut,palm,beach"
<logic parse="{renderCacheList:myList,1,_}"/>
//-- returns as "beach_coconut_palm"
```

---

countCacheList

```
{countCacheList:fieldName}
```

Params:

- *fieldname*: The name of the comma-separated list variable to access within the cache.

Returns:

- The number of elements contained within the specified cache list.

Counts and returns the number of elements within a cached comma-separated list. For example, the cache value "a,b,c" would return the value "3" as the counted number of elements. This field is useful for taking action while tracking a list of objectives, regardless of what the objects are or what order they are completed in.

Example:

```
//-- assume the value of "myList" is "coconut,palm,beach"
<logic parse="{countCacheList:myList}"/>
//-- returns as "3"
```

---

formatCacheList

`{formatCacheList:fieldName,and}`

Params:

- *fieldname*: The name of the comma-separated list variable to access within the cache.
- *and*: The word used for the list's "and" clause. This may be language specific.

Returns:

- The cache list formatted as a human-readable series, where elements are separated by commas and the final item is preceded by the word "and".

Formats a list into a contextual display where all elements are separated by commas and spaces, and the last item in the list is proceeded by the word "and". The word used for "and" is passed to the method as a parameter so that it may be language specific. Formatting a list will produce the following results:

- Single item list (coconut): "coconut"
- Two-item list (coconut,palm): "coconut and palm"
- Three or more item list (coconut,palm,beach): "coconut, palm, and beach"

Example:

```
//-- assume the value of "myList" is "coconut,palm,beach"

<logic parse="{formatCacheList:myList,and}"/>
//-- returns as "coconut, palm, and beach"

<logic parse="{formatCacheList:myList,y}"/>
//-- returns as "coconut, palm, y beach"
```