Extending Lasso 7 Guide

blueworld

Trademarks

Third Party Links

Copyright

Additional copies of this documentation may be purchased at the Blue World store at http://store.blueworld.com/.

First Edition: December 9, 2003

Blue World Communications, Inc.
10900 NE 8th Street, Suite 900
Bellevue, Washington 98004 U.S.A.

Telephone: (425) 646-0288
Fax: (425) 454-4383
Email: blueworld@blueworld.com
Web Site: http://www.blueworld.com

Contents

1

Chapter 1
Introduction

This chapter provides on overview of the Lasso 7 documentation, the section outline, and documentation conventions for this book.

- *Lasso 7 Documentation* describes the documentation included with Lasso 7 products.
- *Extending Lasso 7 Guide* describes the sections in this book.

Lasso 7 Documentation

The documentation for Lasso 7 products is divided into several different manuals and also includes several online resources. The following manuals and resources are available.

- **Lasso Professional 7 Setup Guide** is the main manual for Lasso Professional 7. It includes documentation of the architecture of Lasso Professional 7, installation instructions, the administration interface, and Lasso security. After the release notes, this is the first guide you should read.
- **Lasso Studio 6 User Guide** is the main documentation for Lasso Studio 6. It includes documentation both of Lasso Studio for Adobe GoLive and Lasso Studio for Dreamweaver.
- **Lasso 7 Language Guide** includes documentation of LDML (Lasso Dynamic Markup Language), the language used to access data sources, specify programming logic, and much more.
- **LDML 7 Reference** provides detailed documentation of each tag in LDML 7. This is the definitive reference to the language of Lasso 7. This reference is provided as a LassoApp and Lasso MySQL database within

Lasso Professional 7 and also as an online resource from the Blue World Web site.

- **Lasso 7 Tutorial** includes detailed, step-by-step documentation on how to build a specific Lasso-driven solution.
- **Extending Lasso 7 Guide** is a collection of documentation and sample projects which provide instructions on how to extend Lasso.

Comments, suggestions, or corrections regarding the documentation may be sent to the following email address.

documentation@blueworld.com

Extending Lasso 7 Guide

This is the guide you are reading now. This guide contains information about extending Lasso and LDML. It is organized into the following sections.

- *Chapter 2: LassoApps* contains important information about administering, developing and building LassoApps.
- *Chapter 3: Custom Tags* documents how to build custom substitution tags, container tags, and asynchronous tags entirely in LDML.
- *Chapter 4: Custom Types* documents how to build custom data types in LDML including sub-classing existing types, overriding built-in math, string, and comparison symbols and more.
- *Chapter 5: Advanced Programming Topics* documents how to use pipes and semamphores for inter-thread communication, the tag data type, compound expressions, and the TCP/IP tags which facilitate low-level communication with remote servers.
- *Chapter 6: Lasso C/C++ API* documents how to create new tags, data types, and data sources in C or C++.
- *Chapter 7: Lasso Connector Protocol* documents the protocol which is used for communication between Lasso Web server connectors and Lasso Service.
- *Chapter 8: Lasso Java API* documents how to create new tags and data sources in Java using the new LJAPI 6 programming interface.

The appendix contains the *Extending Lasso Copyright Notice* which covers all of the source code and sample projects that are included with Lasso Professional 7.

Source Code

The Extending Lasso 7 Guide includes example source code for the concepts that are discussed in each chapter. The source code files are included within the 5-ExtendingLasso folder inside the Documentation folder.

All of the source code is provided solely for example purposes. Blue World does not provide support for the included source code or for any derivative LassoApps, tags, data sources, or connectors created using the source code.

Note: Different source files, make files, and project files are installed on Mac OS X and Windows 2000 so that each project can built on the appropriate platform. However, unless otherwise noted a version of each example is provided on both platforms.

What follows are descriptions of the contents of the 5-Extending Lasso folder and its sub-folders:

LassoApps

Includes source code for each of the compiled LassoApps that ship with Lasso Professional 7. Each sub-folder represents one LassoApp.

- **Admin** – The source code for Lasso Administration or Admin.LassoApp. This source is a good reference for creating custom administration interfaces.
- **DatabaseBrowser** – The source code for DatabaseBrowser.LassoApp. This source code is a good example of datasource independent design.
- **GroupAdmin** – The source code for GroupAdmin.LassoApp. This source code provides a good example of using the [Admin_...] tags to add users to groups dynamically.
- **LDMLReference** – The source for the LDML 7 Reference or LDMLReference.LassoApp. This source code is a good example of using named inlines, using sessions to keep state, and using custom tags for data customization.
- **RPC** - The source for the RPC.LassoApp file which processes incoming remote procedure calls. This source is a good example of how to process incoming remote procedure call server requests.
- **Startup** – The source for the Startup.LassoApp file found in LassoStartup. This source code provides many examples of custom tags including the LDML 3 compatibility [List_...] tags, custom data types including [Client_Address] and [Repetition], and dynamic background processes including the event scheduler and email sender.

LCAPI

Includes source code for custom tags and data source connectors. The following projects are provided.

Data Source Connectors

- **Lasso Connector for MySQL** – The complete source code for the built-in data source connector for external MySQL data sources. This is a good starting point for creating a custom data source connector.
- **Sample Data Source** – An example that provides demonstrations of many aspects of creating custom data source connectors with the LCAPI interface.

Custom Tags

- **Math Tags** – The source for a set of extended math tags.
- **File Tags** – The source for a set of file manipulation tags.
- **Tester** – An example that provides demonstrations of many aspects of creating custom tags with the LCAPI interface.

LJAPI

Includes source code for custom tags, types, and data source connectors. Documentation in the JavaDoc format is provided in the HTML folder. The following projects are provided.

Data Source Connectors

- **JDBC Data Source Connector** – Source code for an example JDBC data source connector is provided. This is a good starting point for creating a custom data source connector.

Custom Tags

- **Zip Count Substitution Tag** – The source for a [Zip_Count] tag that counts the number of files within an archive of the zip format.
- **XML Tags**– The source for the built-in [XML_Extract] and [XML_Transform] tags based on the Xalan and Xerces libraries.

Custom Types

- **PDF Type** – The source for the built-in PDF data types based on the iText libraries.
- **Zip Custom Type** – The source for a data type that can store or extract data within archives of the zip format.

LCP

Includes source code for Lasso Web server connectors. The following projects are provided.

Note: The source code for the Web server connectors is only installed on the platform on which that code can be compiled and used. Mac OS X inclues the connectors for Apache and WebSTAR V and Windows includes the connector for IIS.

- **Apache for Mac OS X** – The complete source code for the Lasso Web server connector for Apache 1.x.
- **IIS for Windows 2000** – The complete source code for the Lasso Web server connector for IIS.
- **WebSTAR V for Mac OS X** – The complete source for the Lasso Web server connector for WebSTAR V.

2

Chapter 2
LassoApps

This chapter discusses how to develop, build, and administer LassoApps.

- *Overview* describes LassoApps and their benefits for distributing Lasso-based solutions.
- *Administration* explains how to enable LassoApp serving and how LassoApps are cached.
- *Preparing Solutions* documents how to prepare a Lasso-based solution for conversion into a LassoApp.
- *Building LassoApps* explains how to use LassoApp Builder in Lasso Administration or the [LassoApp_Create] tag to build a LassoApp.
- *Tips and Techniques* provides helpful information about how to create professional quality LassoApps.

Overview

LassoApps allow entire Lasso-based solutions, including format files and image files, to be packaged into a single archive file with a .LassoApp extension. A compiled LassoApp can be easily distributed and executed on any machine running Lasso Professional 7.

LassoApps offer the following benefits:

- **Performance** – LassoApps are loaded into RAM and cached for efficient serving. All format files within the LassoApp are pre-parsed and served without additional disk accesses. LassoApp solutions generally provide better performance than their non-LassoApp counterparts.
- **Size** – LassoApps are stored efficiently as a single file. The overhead associated with multiple format and image files is reduced. Redundant data

within format files is optimized so only a single copy of duplicate strings is stored.

- **Security** – The code within a LassoApp is stored securely in a pre-parsed form. It is not possible to extract format files and code from a LassoApp.
- **Portability** – A LassoApp is an ideal way to distribute a solution by copying and installing a single file with all internal paths intact. LassoApps are fully cross platform. They can be created on either Mac OS X or Windows 2000 and then deployed on either platform without modifications.

LassoApps are stored in a custom binary file format with a .LassoApp extension. LassoApps can be created programmatically using the [LassoApp_Create] tag or through the LassoApp Builder located in Lasso Administration.

LassoApps can be used for any of the following purposes:

- **Packaged Solutions** – LassoApps enable developers to create packaged solutions that can be easily installed by end-users and served by any copy of Lasso Professional 7. LassoApps are placed in the Web serving folder and referenced like a Lasso-based format file.
- **Client Solutions** – LassoApps enable developers to deliver solutions to clients in a convenient, secure package. This is ideal so clients can evaluate the functionality of a solution without requiring access to the source code. LassoApps are placed in the Web serving folder and referenced like a Lasso-based format file.
- **Startup Libraries** – LassoApps can be installed into the LassoStartup folder. The default page of the LassoApp will be executed as a library when Lasso Service starts up and can define custom tags or perform initialization code.
- **Secure Includes** – LassoApps can be included into other format files using the [Include] or [Library] tag. LassoApps can be used to define custom tags or to provide HTML code in a secure manner.

See the sections that follow for information about enabling LassoApps within Lasso Administration, preparing an existing solution for compilation as a LassoApp, and detailed instructions about building LassoApps.

Table 1: LassoApp Tags

Tag	Description
[LassoApp_Create]	Creates a LassoApp. Requires three parameters: the -Root of the LassoApp, the -Entry page or default page, and the -Result path where to write the completed LassoApp.
[LassoApp_Dump]	Removes a LassoApp from the cache. Removes a specific LassoApp if a name is specified or all LassoApps if no name is specified.
[LassoApp_Link]	Defines a link to a file within a LassoApp. This tag must be used to mark all links in HTML anchor, form, and image tags and format file references in [Include] and [Library] tags.
-ResponseLassoApp	Returns a specific page from a LassoApp.

Default LassoApps

Lasso Professional 7 relies on LassoApps for all of its administration interfaces, online documentation, and server start-up code. Lasso Professional 7 ships with the following LassoApps.

- **Admin.LassoApp** – The Lasso Administration interface pre-installed in the Lasso folder within the Web server root. This LassoApp is used to configure Lasso Security, to establish the global preferences of Lasso Service, to browse existing databases, to monitor the email and event queues, and to create new databases and LassoApps.
- **DatabaseBrowser.LassoApp** – The Database Browser allows site visitors to browse through any databases that they have permission to access. This LassoApp is an optional install. It can be found in the Admin folder within the Lasso Professional 7 folder and must be copied into the Web server root in order to be used.
- **GroupAdmin.LassoApp** – The Group Administration interface pre-installed in the Lasso folder within the Web server root. This LassoApp allows group administrators to create users and assign them to groups and for users to change their passwords.
- **LDMLReference.LassoApp** – The LDML Reference is the definitive source for information about each tag in Lasso Dynamic Markup Language. This LassoApp is pre-installed in the Lasso folder within the Web server root.
- **RPC.LassoApp** – This LassoApp responds to incoming remote procedure calls using the XML-RPC format.

- **Startup.LassoApp** – This LassoApp defines custom tags and performs initialization for Lasso Security, the email sender, and the event queue. This LassoApp is installed in the LassoStartup folder and must be present for Lasso Service to start.

The code for each of these LassoApps can be found within the *Documentation Folder > 3-ExtendingLassoGuide > LassoApps* folder. This code is provided as-is without any warranty or support.

Warning: Do not compile LassoApps with the same name as the Blue World supplied LassoApps (e.g. Startup.LassoApp or Admin.LassoApp). Blue World cannot provide any support for customized versions of these LassoApps or for Lasso Professional 7 installations which make use of customized versions of these LassoApps.

Administration

This section discusses how to enable or disable LassoApp support and administer the LassoApp cache using LDML tags and within Lasso Administration.

Enabling LassoApp Support

Lasso Administration includes a global setting to enable or disable LassoApp support. This setting can be found in the *Setup > Global Settings > LassoApps* section of Lasso Administration.

When LassoApp support is disabled only the LassoApps which ship with Lasso Professional 7 can be served (including Admin.LassoApp, GroupAdmin.LassoApp, LDMLReference.LassoApp, and Startup.LassoApp in the LassoStartup folder.

Please see *Chapter 7: Setup* in the Lasso Professional 7 Setup Guide for more information about enabling or disabling LassoApp support.

LassoApp Cache

LassoApps are cached in RAM for efficient serving. Each LassoApp only needs to be read from disk once and from then on is served from high-speed memory. LassoApps are read from disk automatically the first time they are called so there is no need to pre-load them (unless the fastest performance is required on the first load).

Since LassoApps are only read from disk the first time they are called it is necessary to ask Lasso to dump any LassoApps that need to be re-read from

disk. For example, this is necessary if a new version of a LassoApp is copied into the Web serving folder.

LassoApps can be removed from the cache using the *Cache* page in the *Setup > Global Settings > LassoApps* section of Lasso Administration. See *Chapter 7: Setup* of the Lasso Professional 7 Setup Guide for more information. LassoApps can also be removed from the cache programatically using the following steps.

To remove a LassoApp from the cache:

Use the [LassoApp_Dump] tag with the name of the LassoApp. The following example shows how to remove a LassoApp named MySolution.LassoApp from the cache. The LassoApp will be read from disk the next time the LassoApp is called.

```
[LassoApp_Dump: 'MySolution.LassoApp']
```

To remove all LassoApps from the cache:

Use the [LassoApp_Dump] tag without any parameters The following example shows how to remove all LassoApps from the cache. Each LassoApp will be read from disk the next time it is called.

```
[LassoApp_Dump]
```

To preload a LassoApp into the cache:

LassoApps can be preloaded into the cache by calling them from a Web browser or by using the [Include_URL] tag. The following example shows how to preload a LassoApp named MySolution.LassoApp using [Include_URL].

```
[Include_URL: 'http://www.example.com/Lasso/MySolution.LassoApp']
```

If a LassoApp will be used frequently on the server it can be preloaded using the [Event_Schedule] tag in a format file in LassoStartup. The following code would preload a LassoApp named MySolution.LassoApp five minutes after Lasso Service is started. The delay is specified so the other initialization steps have a chance to complete before the LassoApp is loaded.

```
[Event_Schedule: -URL='http://www.example.com/Lasso/MySolution.LassoApp',
  -Delay=5]
```

Serving LassoApps

LassoApps can be served the same way as Lasso format files. They can be served from the Web server root, included in other format files, or placed in the LassoStartup folder and executed at startup. This section includes information about how to use LassoApps in each of these situations.

Web Serving Folder

LassoApps which are placed in the Web serving folder are served like any Lasso-based format files. When they are referenced by name in HTML anchor tags, HTML form actions, or as the target of a -Response... tag, the entry page for the LassoApp is always the page that is served.

Since LassoApps are cached, only one copy of each named LassoApp can be served from a single copy of Lasso Professional 7. If a second LassoApp with the same name is called the cached copy of the first LassoApp will be served in its place. It is important to ensure that multiple copies of the same LassoApp are identical or unexpected results can occur.

The links in the entry page must be marked with the [LassoApp_Link] tag in order to reference other files contained within the LassoApp. See the section on *Preparing Solutions* for more details.

The [LassoApp_Link] tag modifies internal links to be of the form LassoAppName.FileNumber.LassoApp. For example, the link to the entry page of a LassoApp named MySolution.LassoApp would be formated as follows in the source of the LassoApp.

```
<a href="[LassoApp_Link: 'default.lasso']"> Entry Page </a>
```

After the LassoApp is compiled, this link will be changed to the following code. The number referenced in the link is determined when the LassoApp is compiled. This number should not be relied on since it may change if the LassoApp is recompiled.

```
<a href="MySolution.0.LassoApp"> Entry Page </a>
```

The conversion of links marked [LassoApp_Link] is handled automatically. No further action beyond marking internal links with the [LassoApp_Link] tag is required. The site visitor will be able to visit any pages which can be reached from the entry page within the LassoApp and will be able to view any linked images within the LassoApp.

To reference pages in a LassoApp from outside the LassoApp:

Individual pages within a LassoApp can be referenced using the -ResponseLassoApp tag as a parameter to the LassoApp name. For example, the entry page (e.g. default.lasso) of the MySolution.LassoApp LassoApp could be referenced explicitly using the following link.

```
<a href="MySolution.LassoApp?-ResponseLassoApp=default.lasso"> Entry Page </a>
```

The path specified for the -ResponseLassoApp tag should be relative to the folder which was compiled into the LassoApp. The -ResponseLassoApp tag should not be used as part of a database action or to specify the response file for a database action. It should only be used to return a specific format file or image file from within a LassoApp.

Note: By using this technique, even files and images within a LassoApp which cannot be reached from the entry page can be viewed if the visitor knows the path to the file they want to view within the LassoApp.

Database Action Responses

The entry page of a LassoApp can be used as the response to a database action by specifying the path to the LassoApp as the parameter for any of the -Response... command tags. The following form returns the entry file of MySolution.LassoApp as the response to a -FindAll action.

```
<form action="Action.Lasso" method="POST">
  <input type="hidden" name="-FindAll" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-Response" value="MySolution.LassoApp">
  <input type="submit" name="-FindAll" value="Find All People">
</form>
```

Note: The -ResponseLassoApp tag cannot be used in conjunction with a database action to return a particular page from within a LassoApp. Only the entry page of a LassoApp can be returned as the result of a database action.

Lasso Startup Folder

The entry page of a LassoApp can be executed when Lasso Service starts up by placing the LassoApp file within the LassoStartup folder inside the Lasso Professional 7 application folder. The entry file can include as many other files within the LassoApp as it needs in order to perform the desired actions. For example, the Startup.LassoApp LassoApp located in the LassoStartup folder executes code which defines a number of custom tags (e.g. [Email_Send], [Include_URL]) in Lasso Professional 7. Because Startup.LassoApp is located in the LassoStartup folder, these custom tags are automatically available upon startup.

Preparing Solutions

Any Lasso-based solution can be compiled into a LassoApp following these preparation instructions. These steps require changes to be made to each format file which needs to link to another file within the LassoApp and requires files that need to remain user customizable to be stored and referenced outside the LassoApp.

The following steps need to be performed to prepare a solution for compilation as a LassoApp.

- The entire solution must be contained in a single folder including all format files and image files which will be compiled into the LassoApp. The folder should only contain text and GIF or JPEG image files.
- The solution must have a single entry point. One file will be loaded when the LassoApp is called, this file must reference other files within the LassoApp either through HTML links, HTML form actions, redirects or [Include] tags.
- All links to files or images within the LassoApp must be marked with the [LassoApp_Link] tag. This tag changes relative paths to a LassoApp specific format.

Preparing Links

The biggest change required to make most solutions ready to be compiled as a LassoApp is to mark all of the links which reference other files within the solution with the [LassoApp_Link] tag. All HTML anchor ... , image <img>, and form <form> ... </form> tags which reference other files within the LassoApp need to be marked as well as [Include] and [Library] tags. The [LassoApp_Link] tag is processed when the solution is compiled into a LassoApp.

Named anchors, links to targets within the same file, mailto links to email addresses, and links to Web sites on other servers do not need to be marked with the [LassoApp_Link] tag.

The [LassoApp_Link] tag can be safely used in any Lasso solution whether it is compiled into a LassoApp or not. When used in a non-compiled solution the [LassoApp_Link] simply returns the specified link value unchanged.

Note: The [LassoApp_Link] tag cannot be used within custom tags or custom data types. Since a custom tag could be called from a different LassoApp than the one in which it is defined (e.g. if a custom tag is defined in the LassoStartup folder, there is no way for Lasso to determine to which LassoApp the [LassoApp_Link] tag should refer. See the end of this section for tips on working with custom tags within LassoApps.

To prepare links to other files within the LassoApp:

- Anchor tags which reference other files within the LassoApp need to be marked with the [LassoApp_Link] tag. The [LassoApp_Link] tag will accept any relative path which is legal within an HTML anchor tag including those which contain ../ to reference files higher in the folder structure.

The following example shows an HTML anchor tag that references a file named default.lasso contained in a folder named People.

```
<a href="People/default.lasso"> People Page </a>
```

After being marked with the [LassoApp_Link] tag this anchor tag appears as follows.

```
<a href="[LassoApp_Link: 'People/default.lasso']"> People Page </a>
```

Note: Do not mark named anchors, links to targets within the same file, mailto links to email addresses, or links to Web sites on other servers with the [LassoApp_Link] tag.

- Image tags should be marked with the [LassoApp_Link] tag if the referenced image is contained within the compiled LassoApp. The following example shows an HTML image tag that references a file named boat.gif contained in a folder named Images.

  ```
  <img src="Images/boat.gif" height="288" width="288">
  ```

  After being marked with the [LassoApp_Link] tag this anchor tag appears as follows.

  ```
  <img src="[LassoApp_Link: 'Images/boat.gif']" height="288" width="288">
  ```

- The action parameter for HTML <form> tags should be marked with the [LassoApp_Link] tag if it reference a format file explicitly. The following example shows an HTML <form> tag that references a file named result.lasso which is contained in the same folder as the current page.

  ```
  <form action="result.lasso" method="POST">
    ...
  </form>
  ```

  After being marked with the [LassoApp_Link] tag this HTML <form> tag appears as follows.

  ```
  <form action="[LassoApp_Link: 'result.lasso']" method="POST">
    ...
  </form>
  ```

- If an HTML <form> tag references Action.Lasso as its action then the value parameter for the appropriate <input> tag for the -Response command tag should be marked with the [LassoApp_Link] tag. The following example shows an HTML <form> tag that references Action.Lasso. The response for the form is specified as response.lasso in a hidden input for the -Response command tag.

  ```
  <form action="Action.Lasso" method="POST">
    <input type="hidden" name="-Response" value="response.lasso">
    ...
  </form>
  ```

After being marked with the [LassoApp_Link] tag the hidden input appears as follows.

```
<form action="Action.Lasso" method="POST">
  <input type="hidden" name="-Response"
    value="[LassoApp_Link: 'response.lasso']">
  ...
</form>
```

- The file parameter for an [Include] or [Library] tag needs to be marked using the [LassoApp_Link] tag. The following examples show an [Include] tag for a file named include.lasso and a [Library] tag for a file library.lasso.

```
[Include: 'include.lasso']

[Library: 'library.lasso']
```

After being marked with the [LassoApp_Link] tag the tags appear as follows.

```
[Include: (LassoApp_Link: 'include.lasso')]

[Library: (LassoApp_Link: 'library.lasso')]
```

- The response parameter for a [Link_...] tag needs to be marked using the [LassoApp_Link] tag. For example, the [Link_DetailURL] tag accepts a -Response parameter which specifies the format file that should be returned when the link is selected. The following example shows a [Link_DetailURL] tag used within an HTML anchor <a> tag.

```
<a href="[Link_DetailURL: -Response='response.lasso', -Table='People']"> ... </a>
```

After being marked with the [LassoApp_Link] tag, the [Link_DetailURL] tag appears as follows.

```
<a href="[Link_DetailURL: -Response=(LassoApp_Link: 'response.lasso'),
  -Table='People']"> ... </a>
```

Notice that only the name of the response page is marked with the [LassoApp_Link] tag, not the entire href attribute of the anchor tag.

To reference files within a LassoApp from a custom tag:

The [LassoApp_Link] tag cannot be used within custom tags and custom data types. The following techniques can be used to reference files within a LassoApp from custom tags or custom data types.

- References to files can be stored in variables and referenced by variable name within a custom tag. In the following example a reference to a file include.lasso is stored in a variable named IncludeFile. This variable is then referenced within a custom tag.

```
[Variable: 'IncludeFile' = (LassoApp_Link: 'include.lasso')]
…
[Define_Tag: 'myInclude']
  [Return: (Include: $IncludeFile)]
[/Define_Tag]
```

- References to LassoApp files can be passed into custom tags as parameters. In the following example a reference to a file include.lasso is passed as a parameter to a custom tag.

```
[Define_Tag: 'myInclude', -Required='IncludeFile']
  [Return: (Include: #IncludeFile)]
[/Define_Tag]
…
[myInclude: (LassoApp_Link: 'include.lasso')]
```

Building LassoApps

LassoApps can be built programmatically using the [LassoApp_Create] tag or can be built using LassoApp Builder provided in the *Build > LassoApp Builder* section of Lasso Administration.

Lasso Administration

In order to build a LassoApp using LassoApp Builder, the folder containing the files which will be compiled into the LassoApp must be placed in the Admin/BuildLassoApps folder within the Lasso Professional 7 application folder.

The name of the folder to be converted to a LassoApp is selected from a pop-up menu and the path to the entry file within the folder is specified. Any errors which occur are reported within the interface. If successful, the completed LassoApp is placed within the Admin/BuildLassoApps folder. The completed LassoApp has the same name as the folder that was selected with .LassoApp appended.

See *Chapter 8: Build* of the Lasso Professional 7 Setup Guide for complete documentation of LassoApp Builder.

To create a LassoApp using LassoApp Builder:

1 Place all of the files which will be compiled into the LassoApp into a single folder. The folder should only contain Lasso format files and image files. All of the format files should have been prepared following the instructions in the *Preparing Solutions* section of this chapter.

For example, place the format files within a folder named MySolution. This folder contains the entry file default.lasso, a folder of included sub-files, and a folder of images.

Note: All of the files within the source folder will be compiled into the LassoApp even if some of the files are never referenced. In order to create the smallest LassoApps possible, any files which are not needed should be removed from the source folder prior to compiling a LassoApp

2 Place the folder MySolution into the Admin/BuildLassoApps folder within the Lasso 7 application folder.

3 Load Lasso Administration in a Web browser and go to the *Build > LassoApp Builder* section.

```
http://www.example.com/Lasso/Admin.LassoApp
```

4 Choose MySolution from the pop-up menu and ensure that the entry file is default.lasso. Select the *Create LassoApp* button.

Note: If the name of the source folder is not present in the pop-up menu select the Refresh button.

5 If any errors are reported, correct them within the format files of the solution and then return to Lasso Administration to build the LassoApp again. The LassoApp Builder must complete without any errors in order for a LassoApp file to be created.

6 The completed LassoApp will be in the Admin/BuildLassoApps folder named MySolution.LassoApp. This file should be copied into the Web serving folder and can then be loaded through a Web browser. If this solution were placed at the root of the Web serving folder it could be loaded through the following URL.

```
http://www.example.com/MySolution.LassoApp
```

[LassoApp_Create] Tag

In order to build a LassoApp using the [LassoApp_Create] tag the files which will be compiled into a LassoApp need to be placed in a single folder on the same machine as Lasso Service. The source folder and destination file path for the LassoApp will both be specified using fully qualified, platform-specific paths on the same machine as Lasso Service.

The parameters for the [LassoApp_Create] tag are detailed in *Table 2: [LassoApp_Create] Tag Parameters*. An example of using the tag to create a LassoApp follows. The [LassoApp_Create] tag will return 0 if it is successful creating a LassoApp or an error message otherwise. The tag will replace an existing LassoApp file if the -Result parameter specifies a file that already exists.

Table 2: [LassoApp_Create] Tag Parameters

Parameter	Description
-Root	The folder which contains the files that will be compiled into the LassoApp. Should be specified using a fully qualified, platform-specific path.
-Entry	The default format file within the LassoApp which will be loaded when the LassoApp is called. Should be specified relative to the root folder.
-Result	The destination file name for the created LassoApp. Should be specified using a fully qualified, platform-specific path and must end in the file suffix .LassoApp.

To create a LassoApp using the [LassoApp_Create] tag:

1 Place all of the files which will be compiled into the LassoApp into a single folder. The folder should only contain Lasso format files and image files. All of the format files should have been prepared following the instructions in the *Preparing Solutions* section of this chapter.

This folder contains the entry file default.lasso, a folder of included sub-files, and a folder of images. Determine the platform-specific, fully qualified path to this folder.

For example, if the folder is located at the root of the Web serving folder on a Windows 2000 machine then the root path would be as follows.

```
C:\\InetPub\wwwroot\MySolution\
```

If the folder is located at the root of the Web serving folder on a Mac OS X machine then the root path would be as follows.

```
/Library/WebServer/Documents/MySolution/
```

3 Create a format file which contains the following [LassoApp_Create] tag. This tag will build a LassoApp named MySolution.LassoApp stored at the same location as the root folder defined above. The entry file for the LassoApp will be default.lasso immediately inside the MySolution folder.

The [LassoApp_Create] tag for Windows 2000 would be as follows.

```
[LassoApp_Create: -Root='C:\\InetPub\wwwroot\MySolution\',
  -Entry='default.lasso',
  -Result='C:\\InetPub\wwwroot\MySolution.LassoApp\']
```

The [LassoApp_Create] tag for Mac OS X would be as follows.

```
[LassoApp_Create: -Root='/Library/WebServer/Documents/MySolution/',
  -Entry='default.lasso',
  -Result="/Library/WebServer/Documents/MySolution.LassoApp']
```

5 If any errors are reported, correct them within the format files of the solution and then reload the format file to build the LassoApp again.

6 The completed LassoApp should have been created within the Web serving root and can be loaded through the following URL.

http://www.example.com/MySolution.LassoApp

Tips and Techniques

This section presents a number of tips and techniques which can make creating professional quality LassoApps easier.

Naming Conventions

LassoApps should be named with the identifier of the company that created the LassoApp followed by the name of the solution. For example, if Blue World shipped a phone book LassoApp it could be named BW_PhoneBook.LassoApp. This ensures that the LassoApp name will not conflict with LassoApps created by other companies.

Warning: Do not compile LassoApps with the same name as the Blue World supplied LassoApps (e.g. Startup.LassoApp or Admin.LassoApp). Blue World cannot provide any warranty or support for customized versions of these LassoApps or for Lasso Professional 7 installations which make use of customized versions of these LassoApps.

Run-Time Errors

Errors which occur when a LassoApp is executing are reported the same way they are for any Lasso format files. It is important to thoroughly test a LassoApp to ensure that all errors are caught and properly reported to the site visitor. The [Protect] ... [/Protect], [Handle] ... [/Handle] and [Fail] tags can be used to trap for errors and handle them so that the errors are not reported to the site visitor.

Auto-Building Databases

If a LassoApp requires a database table to store solution-specific data it can be created automatically by the LassoApp using the [Database_Create...] tags. Using this technique ensures that a LassoApp can be shipped as a single file and cuts down on the installation required by the end-user.

- LassoApps can safely create tables in the Site database within any installation of Lasso Professional 7. This database is the appropriate place to store both preferences and solution-specific data.
- Tables created in the Site database should follow a naming convention which includes the name of the LassoApp in each table name. For example, a LassoApp named MySolution.LassoApp could create tables named MySolution_Preferences and MySolution_Data. Using a clear naming convention ensures that the global administrator knows why individual tables were created and ensures that different LassoApps do not create tables with the same name.
- If necessary, the LassoApp may need to ask for additional permissions in order to create new tables or to gain access to the tables that have been created. See the section on *Lasso Security* below for more information.
- Always check to make sure that a table does not exist before creating a new table. A LassoApp should never overwrite data in the Site table without explicitly ensuring that the administrator wants to do so.

Lasso Security

LassoApps are executed with the permissions of the current site visitor the same as any Lasso format files. If a LassoApp needs to have access to databases, tables, or tags that can be secured in Lasso Administration then it should check that the appropriate permissions are present before executing.

Tags

If a LassoApp requires access to tags which can be secured in Lasso Administration such as the [Admin_...] tags, [Database_Create...] tags, [File_...] tags, [Email_Send] or [Event_Schedule] tags, it should first check to be sure those tags are allowed by the current user before executing. The following code will check to be sure the [Email_Send] tag is available and display an error message if it is not.

```
[If: (Lasso_TagExists: 'Email_Send') == False]
  <br>Error: The tag Email_Send is required in order for this LassoApp to execute.
  Please enable it within Lasso Administration before proceeding.
[/If]
```

LassoApps can be created even if the tags they require are not present when they are built and compiled. However, syntax errors will be reported when the LassoApp is served or executed.

Databases and Tables

If a LassoApp requires access to certain databases or tables it should first check to be sure they are available to the current user before executing. The following code will check to be sure the People table of the Contacts database is available.

```
[Inline: -Database='People', -Table='Contacts', -Show]
  [If: (Error_CurretError) != (Error_NoError) || (Field_Name: -Count) == 0]
    <br>Error: The People table of the Contacts database is required
      in order for this LassoApp to execute. Please enable it within Lasso
      Administration before proceeding.
  [/If]
[/Inline]
```

Groups and Users

The [Admin_...] tags can be used to create new users and assign them to a group. These tags are essential if Lasso Security is going to be used to handle multiple user accounts for a LassoApp. Since there is no tag to create a group and assign it permissions, the documentation for a LassoApp solution will need to walk a Lasso global administrator through creating a group with the proper name, assigning permissions, and creating a group administrator.

Lasso Startup

If code needs to be executed when Lasso Service starts up, then a LassoApp can be placed within the LassoStartup folder within the Lasso Professional 7 application folder. Usually, a solution that requires startup code would consist of two LassoApps, one that installs in LassoStartup and a second that defines the user interface for the solution.

3

Chapter 3
Custom Tags

This chapter introduces custom tags and shows how each can be created using LDML tags.

- *Overview* introduces the concepts behind custom tags.
- *Custom Tags* describes how to create custom tags including information about processing parameters and using local variables.
- *Container Tags* describes how to create custom container tags.
- *Web Services, Remote Procedure Calls, and SOAP* describes how to create tags that function as remote procedure calls through XML-RPC or SOAP and how to call those tags from another server.
- *Asynchronous Tags* describes how to create custom asynchronous process tags and background processes.
- *Overloading Tags* describes how to use criteria to determine which tag will execute and how to redefine built-in LDML tags.
- *Libraries* describes how to package sets of custom tags for distribution.

Overview

Lasso Professional 7 allows Web developers to extend Lasso Dynamic Markup Language (LDML) by creating custom tags programmed using LDML tags.

LDML custom tags have the following features:

- Custom tags operate just like built-in LDML substitution tags. They can be used in nested expressions, return data of any data type, and allow the use of encoding keywords.
- Custom process, substitution, or container tags can be created.

- They can be created in any Lasso format file and used instantly.
- They are written in LDML. No programming experience or knowledge of a programming language other than LDML is required.
- They can be collected into libraries of tags which can be loaded into any format file using the [Library] tag.
- Custom tags can be used as the target for remote procedure calls enabling communication between Web servers.
- Existing tags can be redefined.
- Tags can be defined with criteria for when they will run. This allows the same tag name to be used with different parameters and makes it easy to redefine tags for custom purposes.
- They can be defined in a format file or library within the LassoStartup folder, making them available to all pages processed by Lasso.
- Asynchronous tags allow operations to be performed in a separate thread so the current format file is served as fast as possible to the site visitor.

Custom data types can also be created in LDML. See *Chapter 4: Custom Types* for more information.

Possible Uses

Custom tags can be used in any of the following ways:

- To define a new LDML tag that can be called like any built-in LDML tags.
- To reuse a portion of LDML code several times in the same format file.
- To create a macro which allows the same HTML code to be reused several times without being retyped.
- To structure the logic of complex calculations using local variables and tag parameters.
- To redefine and customize existing LDML tags.
- To defer processing of some code until after the visitor has already received the format file.
- To allow remote Web servers to make remote procedure calls to Lasso through XML-RPC.

Naming Conventions

Custom tags and custom types should be named using a combination of letters, numbers, and the underscore character, but should never start with the underscore character. In order to prevent confusion between custom tags created by different developers, all custom tags should start with an

identifier for the author of the custom tag, followed by an underscore, then the name of the custom tag.

For example, if Blue World was providing a custom tag which wrapped code with HTML bold tags it might be named [BW_Bold]. All of the tags in this guide will be defined with an Ex_ prefix meaning Example.

RPC Note: Tags which will be used for XML-RPC are typically named with a group named followed by a method, e.g. group.method.

Parameter References

All values are passed to and from custom tags by reference. This improves the speed and efficiency of custom tags by reducing the number of times that date needs to be copied. Parameeter references make tags that perform operations on their parameters possible, but require careful programming in order to avoid unintended side-effects.

Lasso is an object-oriented system and every value in a given format file can be thought of as an object. Variables are simply references to objects and it is possible to have multiple references to the same object.

For example, the [Iterate] ... [/Iterate] tag accepts two parameters. The first is an array of values. The second is a variable that will be set as a reference to each element in the array in turn. The values are not copied out of the array, but the variable points to each value in turn. If the variable modifies the value then that new value is automatically modified in the array as well. This code modifies each element in an array to be uppercase.

```
[Var: 'myArray' = (Array: 'one','two','three')]
[Iterate: $myArray, (Var: 'myItem')]
  [Var: 'myItem' = (String_Uppercase: $myItem)]
[/Iterate]
[Output: $myArray]
```

➜ Array: (ONE), (TWO), (THREE)

Custom tags work similarly. The following rules defined how values are passed to and from custom tags.

- All values passed into a custom tag are passed by reference. References are stored in local variables with the same name as the parameter and in a [Params] array.
- Any modifications of the values in the automatically created local variables or the [Params] array will result in the original values outside the custom tag being modified.
- It is recommended to use a set of uniquely named local variables within the custom tag so as not to interfere with the parameters passed by refer-

ence. The values of parameters can be copied into the local variables making their modifications safe.

- Local variables are created new for each custom tag call. References to local variables do not persist from tag call to tag call.
- All values are returned from a custom tag by reference. Normally this will be a reference to a local variable. Since a new set of local variables are created each time a tag is called the return value is safe.
- The return value can also be a reference to one of the input parameters or to a page or global variable. In this case any further modifications to the return value after the custom tag has returned will be reflected in the original value.

These rules are illustrated in the many examples that follow.

Custom Tags

Custom tags can be created in LDML using the [Define_Tag] ... [/Define_Tag] tags. The following table details the tags that are used to create custom tags. These tags are used to process the parameters of the custom tag and to return values from the custom tag.

Custom substitution and process tags can be created in any format file and will be available immediately. Custom container tags can only be created in the LassoStartup folder. See the section on *Libraries* for information about how to create libraries of tags, load tags in LassoStartup, and create tags which can be used by any format file.

It is not possible to create custom command tags using LDML. Command tags are implemented in data source modules. See the documentation on LCAPI later in this book for more information.

See *Chapter 4: Custom Types* for information about creating custom data types and member tags.

Table 1: Tags For Creating Custom Tags

Tag	Description
[Define_Tag]	Defines a new substitution tag or a new member tag if used within a type definition. Requires a single parameter, the name of the tag to be defined. Other parameters are defined in Table 2: [Define_Tag] Parameters.
[Local]	Sets or retrieves the value of a local variable within a custom tag definition.
[Local_Defined]	Checks to see if a local variable has been defined within a custom tag definition.
[Locals]	Returns a map of all the local variables which have been defined within a custom tag definition.
[Params]	Returns an array of all the parameters which were passed to the custom tag.
[Params_Up]	Returns an array of all the parameters which were passed to the custom tag which called the current custom tag.
[Return]	Returns a value from a custom tag. No further processing is performed.
[Run_Children]	Process the contents of a custom tag created with the -Container option.
[Tag_Name]	Returns the name of the current tag.

The parameters for the [Define_Tag] ... [/Define_Tag] tags are detailed in teh following table. The type of tag created, required parameters, return data type, and more are all specified in the opening [Define_Tag] tag.

Table 2: [Define_Tag] Parameters

Tag	Description
'Tag Name'	The name of the tag to be defined. Required.
-Async	Specifies that the tag should be run asynchronously. Asynchronous tags cannot return a value. Optional.
-Container	Specifies that the tag is a container tag. [Run_Children] can be used if this parameter is specified. Optional. See also -Looping for looping container tags.
-Criteria	Specifies the criteria under which the tag will run. If the criteria is not met then the next tag in the calling chain will be used instead. Optional.
-Description	A brief description of the tag. Can include calling instructions, author of the tag, etc. Optional.
-Looping	Specifies that the tag is a looping container tag. [Run_ Children] can be used if this parameter is specified. Optional. See also -Container for non-looping container tags.
-Optional	Names an optional parameter of the tag. Optional.
-Priority	Requires the value 'High', 'Low', or 'Replace'. Specifies whether the tag should replace an existing tag with the same name or be placed before or after existing tags in the calling chain. Optional.
-Privileged	Specifies that the custom should run with the privileges of the current user rather than with the privileges of the user who ultimately calls the custom tag.
-Required	Names a required parameter of the tag. If the parameter is not specified then an error will result. Optional.
-ReturnType	Specifies the type of the return value of the tag. If a value of different type is returned then an error is generated.
-RPC	Specifies that the tag should be made available to remote Web servers as a remote procedure call. The tag can then be accessed through RPC.LassoApp.
-SOAP	Specifies that the tag should be made available to remote Web servers as a SOAP operation.The tag can then be accessed through RPC.LassoApp. The -Type and -ReturnType tags must be used to specify parameter and return types.
-Type	Specifies the type for the preceding -Required or -Optional parameter. If the tag is called with a parameter that is not of the proper type then an error is generated.

See the section on *Libraries* for information about how to create libraries of tags, load tags in LassoStartup, and create tags which can be used by any format file.

It is not possible to create custom command tags using LDML. See *Chapter 4: Custom Types* for information about creating custom data types and member tags.

Substitution Tags

A new substitution tag is defined using the [Define_Tag] ... [/Define_Tag] container tag within an enclosed [Return] tag that defines the value of the tag. The opening [Define_Tag] tag requires the name of the new substitution tag to be defined. All of the LDML code between the two tags is stored and is executed each time the tag is called.

In the following example, a tag [Ex_EmailAddress] is defined which returns an example email address for John Doe, johndoe@example.com.

```
[Define_Tag: 'Ex_EmailAddress']
  [Return: 'johndoe@example.com']
[/Define_Tag]
```

This tag can be called like any LDML substitution tag within the format file where the tag is defined. The following code calls this tag twice, once to provide the address for the HTML anchor tag and a second time to provide the text of the anchor.

```
<a href="[Ex_EmailAddress]"> [Ex_EmailAddress] </a>
```

```
➜ <a href="mailto:johndoe@example.com"> johndoe@example.com</a>
```

Process Tags

A new process tag is defined using the [Define_Tag] ... [/Define_Tag] container tags. The opening [Define_Tag] tag requires the name of the new process tag to be defined. All of the LDML code between the two tags is stored and is executed each time the tag is called. Since process tags do not return a value, the body of the tag should not contain a [Return] tag.

In the following example, a tag [Ex_SendEmail] is defined which sends an email to an example email address for John Doe, johndoe@example.com. The tag is defined within a LassoScript.

```
<?LassoScript
  Define_Tag: 'Ex_SendEmail';
    Email_Send: -Host='mail.example.com',
      -To='johndoe@example.com',
      -From='lasso@example.com',
```

```
        -Subject='Sample Email',
        -Body='This email was sent from a custom tag.';
    /Define_Tag;
?>
```

This tag can be called like any LDML process tag within the format file where the tag is defined. The following code calls this [Ex_SendEmail] so an email will be sent to johndoe@example.com each time the page with this code is served by Lasso.

```
[Ex_SendEmail]
```

Privileged Tags

Custom tags normally run with the permissions of the user that calls the custom tag. Using the -Privileged keyword a custom tag will instead run with the permissions of the user who defined the custom tag.

This allows the execution of privileged actions to be written into custom tags. The privileged action can be performed without opening up general permission for performing similar actions to the end-users.

For example, a custom tag which is defined in LassoStartup that has the -Privileged keyword will always execute as the global administrator of the machine. Priviliged custom tags can then be used to modify internal security settings or perform other actions that require global administrator permission.

Returning Values

In order for a custom tag to return a value it needs to use the [Return] tag. The parameter for the [Return] tag will be returned as the value of the custom tag and no further processing will be performed. A value of any type can be returned using the [Return] tag including simple decimal or integer numbers, strings, complex maps and arrays, or even custom types.

Custom tags can also return values by setting variables. See the section on *Page Variables* that follows for additional details.

The following custom tag returns a string that informs the site visitor of what day it is. If the current day is January 1st then Happy New Year! is returned. Note that if the conditional returns True then the [Return: 'Happy New Year!'] tag is executed and the tag is exited without executing the second [Return] tag that follows.

```
[Define_Tag: 'Ex_Greeting']
  [If: (Date_GetDay) == 1 && (Date_GetMonth) == 1]
    [Return: 'Happy New Year!']
  [/If]
  [Return: 'The date is ' + (Server_Date: -Long) + '.']
[/Define_Tag]
```

When executed on any day other than the 1st of January this tag returns the current date.

```
[Ex_Greeting]
```

➜ The date is August 27, 2001.

Encoding

Encoding is handled automatically by Lasso when values are returned from a custom tag. Encoding follows the same rules as for built-in substitution tags. These rules are summarized below.

- If no encoding keyword is specified and the custom tag returns a string value then the tag follows the same rules as built-in substitution tags. The string value will be HTML encoded if it is output to the format file or will have no encoding applied if the tag is used as a sub-tag or in an expression.

  The following custom tag [Ex_String] would have HTML encoding applied.

  ```
  [Ex_String]
  ```

  ➜ Bold Text

  However, if the same tag is used as a sub-tag, no encoding is applied.

  ```
  [Variable: 'myString'=(Ex_String)]
  [Variable: 'myString', -EncodeNone]
  ```

  ➜ Bold Text

  Note: If the tag is used within [Encode_Set] ... [/Encode_Set] tags then the default encoding which is set in the opening [Encode_Set] tag will be used instead of -EncodeHTML when the tag's value is output directly to a format file.

- If no encoding keyword is specified and the custom tag returns any data type other than string then no encoding is applied and the specified data type is returned.

  The following custom tag [Ex_Array] has no encoding applied since it returns an array.

  ```
  [Ex_Array]
  ```

  ➜ (Array: (Bold Text))

Note: Even if the tag is used within [Encode_Set] ... [/Encode_Set] tag, no encoding will by applied by default unless an explicit encoding keyword is specified.

- If an explicit encoding keyword other than -EncodeNone is specified then the return value from the tag is converted to a string and the specified encoding is applied. Use of an explicit encoding keyword guarantees that the value from the tag will be of data type string.

  The following custom tag [Ex_Array] has explicit HTML encoding applied.

```
[Ex_Array: -EncodeHTML]
➜ (Array: (&lt;b&gt;Bold Text&lt;/b&gt;))
```

Note: The encoding keyword -EncodeNone instructs Lasso that no encoding is desired for a custom tag. For custom tags which return any data type other than string, -EncodeNone is equivalent to not specifying an encoding keyword.

Parameters

Custom tags can accept any mix of named or unnamed parameters. These parameters can be named using the -Required and -Optional parameters in the opening [Define_Tag] tag. Each parameter is automatically defined as a local variable within the tag. If a required parameter is omitted from a tag call then an error is generated. If an optional parameter is omitted then the local variable corresponding to that parameter will not be defined.

- **Named Parameters** – The -Required and -Optional parameters for a tag can be listed in any order. Each -Required parameter must have a matching keyword/value parameter in the parameters for the tag. Keywords must be preceded by a hyphen.

  The following example defines a tag [Ex_Note] which accepts two parameters. -Message is required and is the message to be displayed. -Font is an optional parameter that changes the font of the displayed message if it is specified, otherwise Arial is used.

```
[Define_Tag: 'Ex_Note', -Required='Message', -Optional='Font']
  [If: (Local_Defined: 'Font') == False]
    [Local: 'Font' = 'Arial']
  [/If]
  [Return: '<font face="' + #Font + '"> ' + #Message + ' </font>']
[/Define_Tag]
```

  The parameters can be used in any order when the tag is called, but the -Message parameter must be present.

```
[Ex_Note: -Font='Helvetica', -Message='Hello World', -EncodeNone]
```

➜ Hello World

[Ex_Note: -Message="Hello World', -EncodeNone]

➜ Hello World

Note: Extra named parameters passed into a custom tag will also create local variables automatically even if the -Required and -Optional parameters are not used.

- **Unnamed Parameters** – The -Required parameters for a tag should be listed in the order they will be specified in the tag followed by any optional parameters that may be specified. Each unnamed parameter of the tag will be assigned in order to the -Required or -Optional parameter in the corresponding position.

  The tag [Ex_Note] defined above accepts two parameters. The first parameter is required and is assigned the name Message. The second parameter is optional and is assigned the name Font if specified.

  When the tag is called at least one parameter must be specified. If a second parameter is specified it is used as the font for the message, otherwise the default font is used..

  [Ex_Note: 'Hello World', 'Helvetica', -EncodeNone]

  ➜ Hello World

  [Ex_Note: 'Hello World', -EncodeNone]

  ➜ Hello World

- **Combination Parameters** – A combination of named an unnamed parameters can be used. First, all keyword/value parameters are assigned to the -Required or -Optional parameters specified in the opening [Define_Tag] tag. Then, any remaining parameters are assigned in order to any -Required or -Optional parameters that have not yet been assigned values.

  For example, the tag [Ex_Note] defined above is called with one unnamed parameter and one keyword/value -Font parameter. First, the -Font parameter is assigned to the -Optional font parameter. Then, the unnamed parameter is assigned to the -Required message parameter.

  [Ex_Note: 'Hello World', -Font='Helvetica', -EncodeNone]

  ➜ Hello World

- **Parameters Types** – The type of each parameter can be specified by including a -Type parameter immediately after the -Required or -Optional parameter. When the tag is called if the specified parameter is not of the proper type then an error will be generated.

The [Ex_Note] tag can be redefined to require that the -Message parameter be a string.

```
[Define_Tag: 'Ex_Note', -Required='Message', -Type='String', -Optional='Font']
  [If: (Local_Defined: 'Font') == False]
    [Local: 'Font' = 'Arial']
  [/If]
  [Return: '<font face="' + #Font + '"> ' + #Message + ' </font>']
[/Define_Tag]
```

Now if the tag is called with a decimal value for the -Message parameter an error will be generated..

```
[Ex_Note: -Message=99, -EncodeNone]
```

➜ *Syntax Error*

Any tag defined with -Required and -Optional parameters can always be called with a combination of named and unnamed parameters. Documentation for custom tags should always specify how a tag should be called.

Parameters Array

If greater control is required over the parameters which are passed into a tag then the [Params] array can be inspected directly. This array contains one element for each parameter that is passed into a custom tag.

- **Simple Parameters** – Simple parameters are included as single elements within the array. Each parameter has the same data type as the literal or variable which was passed to the tag.
- **Name/Value Parameters** – Name/Value parameters are included as elements of the data type pair within the array. Each part of the pair has the same data type as the literal or variable which was passed to the tag.
- **Keyword Parameters** – Keyword parameters are included as string parameters. They should be distinguished by requiring that all keyword names start with a leading hyphen.
- **Keyword/Value Parameters** – Keyword/Value parameters are included as a pair with a string as the first element and the value as the second element. They should be distinguished by requiring that all keyword names start with a leading hyphen.
- **Encoding Keywords** – Encoding keywords are handled automatically by Lasso. They are not passed to custom tags within the [Params] array. Custom tags do not need to do anything special to take advantage of encoding nor is there any way to disable automatic encoding of returned string values.

The [Params_Up] tag is a special purpose tag that allows inspection of the [Params] array from the custom tag which called the current tag. This tag can

only be used if the current tag was called from within a custom tag and can be used to create tags that change their values based on the parameters to the calling tags.

To inspect the parameters of a custom tag:

The [Params] array provides access to all the parameters of the current tag. The following example shows a custom tag [Ex_Echo] that outputs information about all the parameters that were passed to the tag by looping through the [Params] array.

```
[Define_Tag: 'Ex_Echo']
  [Local: 'Output' = '']
  [Loop: (Params)->Size]
    [Local: 'Temp' = (Params)->(Get: (Loop_Count))]
    [If: #Temp->Type == 'pair']
      [#Output += '<br>Pair: ']
      [#Output += '<br> ' + #Temp->First->Type + ': ' + (#Temp->First)]
      [#Output += '<br> ' + #Temp->First->Type + ': ' + (#Temp->Second)]
    [Else]
      [#Output += '<br>' + #Temp->Type + ': ' + (#Temp)]
    [/If]
  [/Loop]
  [If: (#Output == '')]
    [#Output = '<br>No Parameters']
  [/If]
  [Return: #Output]
[/Define_Tag]
```

When this tag is called with different parameters the following output is created. Note that keywords are simply strings that start with a hyphen and that the -EncodeNone encoding keyword is not represented in the output.

```
[Ex_Echo: 'One', 'Two='Three', -Four, -Five='Six', -Seven=8, -Nine=1.0, -EncodeNone]

➜ String: One
  Pair:
    String: Two
    String: Three
  String: -Four
  Pair:
    String: -Five
    String: Six
  Pair:
    String: -Seven
    Integer: 8
  Pair:
    String -Nine
    Decimal: 1.0
```

To get the value of a keyword/value parameter:

The following custom tag uses the [Params->Find] tag to retrieve several named keyword/value parameters from the [Params] array. The tag [Ex_Greeting] accepts two parameters: -First which should have the first name of a person as its value and -Last which should have the last name of its person as its value. It returns a greeting to that person.

```
<?LassoScript
  Define_Tag: 'Ex_Greeting';
    Local: 'First' = Params->(Find: '-First')->(Get: 1);
    Local: 'Last' = Params->(Find: '-Last')->(Get: 1);
    Return: 'Dear ' + #First + ' ' + #Last;
  /Define_Tag;
?>
```

When the tag is called it parses the two defined parameters and ignores all others.

```
[Ex_Greeting: -First='John', -Last='Doe'] ➜ Dear John Doe
```

```
[Ex_Greeting: -First='John', -Last='Doe', -Title='Mr.'] ➜ Dear John Doe
```

To get the value of all unnamed parameters:

The [Params] array provides access to all the parameters of the current tag. The following example shows a custom tag [Ex_Concatenate] that concatenates the value of all simple, unnamed parameters together and ignores all name/value and keyword/value parameters.

```
[Define_Tag: 'Ex_Concatenate']
  [Local: 'Output' = '']
  [Loop: (Params)->Size]
    [Local: 'Temp' = (Params)->(Get: (Loop_Count))]
    [If: #Temp->Type != 'pair']
      [#Output += #Temp]
    [/If]
  [/Loop]
  [Return: #Output]
[/Define_Tag]
```

When this tag is called with different parameters the following output is created. Note that any named parameters are ignored and that the -EncodeNone encoding keyword is not represented in the output.

```
[Ex_Echo: 'One', 'Two='Three', -Four, -Five='Six', -Seven=8, -Nine=1.0, -EncodeNone]
```

➜ One-Four

To get the parameters from the calling tag:

The [Params_Up] tag provides access to the parameters of the calling tag. The following tag [Ex_UnnamedParams] returns an array of all unnamed param-

eters from the calling tag. This tag could be used to filter the [Params] array so only unnamed parameters remained.

```
[Define_Tag: 'Ex_UnnamedParams']
  [Local: 'Output' = (Array)]
  [Loop: (Params_Up)->Size]
    [Local: 'Temp' = (Params_Up)->(Get: (Loop_Count))]
    [If: #Temp->Type != 'pair']
      [#Output->(Insert: #Temp)]
    [/If]
  [/Loop]
  [Return: #Output]
[/Define_Tag]
```

The [Ex_UnnamedParams] tag can now be used to rewrite the [Ex_Concatenate] custom tag by looping through the [Ex_UnnamedParams] array rather than through the [Params] array.

```
[Define_Tag: 'Ex_Concatenate']
  [Local: 'Output' = '']
  [Local: 'Unnamed_Params' = (Ex_UnnamedParams)]
  [Loop: (#Unnamed_Params)->Size]
    [#Output +=  (#Unnamed_Params)->(Get: (Loop_Count))]
  [/Loop]
  [Return: #Output]
[/Define_Tag]
```

When this tag is called with different parameters the following output is created. Note that any named parameters are ignored and that the -EncodeNone encoding keyword is not represented in the output.

```
[Ex_Echo: 'One', 'Two='Three', -Four, -Five='Six', -Seven=8, -Nine=1.0, -EncodeNone]
```

➜ One-Four

Page Variables

Custom tags can set and retrieve the values of variables which are defined in the current format file. This provides a method of passing additional parameters to custom tags by setting pre-defined variables and a method of passing multiple values out of a custom tag.

Any use of page variables should be considered carefully. Local variables, which are defined in the following section, are usually sufficient for storing data required while executing a tag. If data needs to be stored between executions of a tag then it might be more efficient to create a custom data type. See the following section on *Custom Types* for more information.

If a custom tag must store values in page variables it should precede all variable names with the full name of the custom tag followed by an under-

score. For example, the custom tag [Ex_Concatenate] would create variables named Ex_Concatenate_Value, Ex_Concatenate_Output, etc.

Local Variables

Each custom tag can create and manipulate its own set of local variables. These variables are separate from the page variables and are deleted when the custom tag returns. Using local variables ensures that the custom tag does not alter any variables which other custom tags or the page developer is relying on having a certain value.

For example, many developers will use the variable Temp to store temporary values. If a page developer is using the variable Temp and then calls a custom tag which also sets the variable Temp, then the value of the variable will be different than expected.

The solution is for the custom tag author to use a local variable named Temp. The local variable does not interfere with the page variable of the same name and is automatically deleted when the custom tag returns. In the following example, a custom tag returns the sum of its parameters, storing the calculated value in Temp.

```
<?LassoScript
  Define_Tag: 'Ex_Sum';
    Local: 'Temp'=0;
    Loop: (Params)->Size;
      Local: 'Temp'=(Local: 'Temp') + (Params)->(Get: Loop_Count);
    /Loop;
    Return: #Temp;
  /Define_Tag;
?>
```

The final reference to the local variable temp is as #Temp. The # symbol works like the $ symbol for page variables, allowing the variable value to be returned using shorthand syntax.

When this tag is called, it does not interfere with the page variable named Temp.

```
[Variable: 'Temp' = 'Important value:']
[Variable: 'Sum' = (Ex_Sum: 1, 2, 3, 4, 5)]
[Output: '<br>' + $Temp + ' ' + $Sum + '.', -EncodeNone]
```

➜
Important value: 15.

Parameter and Return Types

The -Type and -ReturnType parameters can be used to check that the parameters which are being passed to the tag are of the proper type before the tag

is executed and that the return value of the tag is the proper type when the tag completes.

The -Type parameter is placed immediately after each -Required or -Optional parameter. The corresponding parameter must be of the specified type when the tag is executed or a syntax error is generated. Using these tags reduces the amount of double checking of types that is required within the body of the tag.

The -ReturnType parameter specifies the type that the returned value of the tag must be. If the tag attempts to return a value of a different type then an error is generated. Using this tag is useful as a double check for a tag that is always expected to return a certain data type. It makes enforcement of the return type explicit rather than relying on the custom tag author to ensure that the return type is always proper.

```
[Define_Tag: 'Ex_Bold', -Required='theString', -Type='String', -ReturnType='String']
  [Return: '<b>' + #theString + '</b>']
[/Define_Tag]
```

If the [Ex_Bold] tag is called with a number then a syntax error will be returned. The following example first shows a successful call to the tag, then an unsuccessful call.

```
[Ex_Bold: 'Bold Text'] ➜ <b>Bold Text</b>

[Ex_Bold: 123.456] ➜ Syntax Error
```

Criteria

The -Criteria parameter allows custom tags to check certain conditions before any code in the tag is executed. Usually this is used to confirm that the appropriate parameters have been passed to the custom tag. If the criteria fails then a syntax error will be generated.

The -Criteria parameter requires a conditional expression. If the evaluated expression returns False then the tag execution is halted and an error is returned.

The code within the -Criteria are executed as if they were specified within the body of the [Define_Tag] ... [/Define_Tag]. Locals can be used to reference -Required or -Optional parameters and the [Params] array can be inspected. -Criteria can also inspect page variables.

To use criteria to check the parameters of a custom tag:

Specify the -Criteria parameter in the opening [Define_Tag] tag. If the condition in the criteria fails then the tag will not be executed. The following code checks to be sure that the tag's required parameter is a string.

```
[Define_Tag: 'Ex_Bold', -Required='theString', -Criteria=(#theString->Type == 'string')]
  [Return: '<b>' + #theString + '</b>']
[/Define_Tag]
```

If the [Ex_Bold] tag is called with a number then a syntax error will be returned. The following example first shows a successful call to the tag, then an unsuccessful call.

```
[Ex_Bold: 'Bold Text'] ➜ <b>Bold Text</b>

[Ex_Bold: 123.456] ➜ Syntax Error
```

Error Control

Custom tags should use the -Required, -Optional, -Type, -ReturnType, and -Criteria parameters to ensure that the parameters of the tag are of the proper type and that the return value is of the proper type. These tags ensure that Lasso developers are alerted of errors when the page is first executed, rather than encountering obscure runtime errors later.

Errors can be returned from custom tags using the [Error_SetErrorMessage] and [Error_SetErrorCode] tags. A custom tag which is explicitly returning an error code should always return [Error_NoError] if no error occured or an explicit error message otherwise.

Container Tags

A container tag can be created by specifying either the -Container or -Looping keyword within the opening [Define_Tag] tag. When the tag is used both an opening and a closing tag must be specified or an error will occur. The return value of the tag replaces the entire container tag. The contents of the container tag can be accessed using the [Run_Children] tag.

If the -Looping keyword is used the [Loop_Count] will be automatically changed when the custom tag is called. If the -Container keyword is used then the [Loop_Count] will not be modified by the container tag. This distinction allows both looping and simple container tags to be created.

Note: The output of a container tag is not encoded. This allows HTML to be output from container tags without requiring an -EncodeNone tag.

To create a simple container tag:

The following example creates a simple container tag [Ex_Font] ... [/Ex_Font] that wraps its parameters with an HTML <font> tag. The tag takes three optional parameters -Face, -Size, and -Color which correspond to the parameters of the HTML <font> tag.

```
[Define_Tag: 'Ex_Font', -Container,
    -Optional='Face', -Optional='Size', -Optional='Color']
  [If: !(Local_Defined: 'Face')][Local: 'Face' = 'Verdana']
  [If: !(Local_Defined: 'Size')][Local: 'Size'= 1]
  [If: !(Local_Defined: 'Color')][Local: 'Color' = 'black']

  [Return: '<font face="' + #face + '" size="' + #size + '" color="' + #color + '"> ' +
    (Run_Children) + ' </font>']
[/Define_Tag]
```

A call to this tag appears like this. The -Face and -Color of the output are specified, but the -Size is left to the default of 1.

```
[Ex_Font: -Face='Helvetica', -Color='red'] My Message [/Ex_Font]

➜ <font face="Helvetica" size="1" color="red"> My Message </font>
```

To use the contents of the container tag multiple times:

The following example creates a tag [Ex_Link] that creates a pair of HTML anchor tags with the contents of the container used as both the URL to be followed and the text of the link. This could be used to automatically create hyperlinks out of URLs contained in text. The tag does not require any parameters.

```
[Define_Tag: 'Ex_Link', -Container]
  [Return: '<a href="' + (Run_Children) + '"> ' + (Run_Children) + ' </a>']
[/Define_Tag]
```

A call to this tag appears like this. The specified URL is included in the results twice.

```
[Ex_Link] http://www.blueworld.com [/Ex_Link]

➜ <a href="http://www.blueworld.com"> http://www.blueworld.com </a>
```

To create a looping container tag:

The following example creates a tag that loops ten times repeating its contents. The -Looping keyword is used in the [Define_Tag] tag to indicate that this is a looping tag rather than a simple container.

```
[Define_Tag: 'Ex_Loop10', -Looping]
  [Local: #Output = '']
  [Loop: 10]
    [#Output += Run_Children]
  [/Loop]
  [Return: #Output]
[/Define_Tag]
```

A call to this tag appears like this. The specified contents of the tag is repeated ten times with the [Loop_Count] updated each time..

```
[Ex_Loop10] <br>This is loop [Loop_Count]. [/Ex_Loop10]
```

```
➜ <br>This is loop 1.
  <br>This is loop 2.
  …
  <br>This is loop 10.
```

If the -Container keyword rather than the -Looping keyword had been used the tag still would have repeated its contents ten times, but the [Loop_Count] would have returned the same value for each repetition.

Web Services, Remote Procedure Calls, and SOAP

Lasso supports remote procedure calls through the XML-RPC and Simple Object Access Protocol (SOAP) standards. Both types of remote procedure calls allow one server on the Internet to call a function that is located on another server. The parameters of the function call and the results of the function call are transmitted between the servers using XML.

Custom tags can be automatically made available to remote servers by specifying the -RPC or -SOAP parameter when the tag is created. Any tag which is specified as a remote procedure call will be accessible through RPC.LassoApp which is located in the LassoStartup folder. The LassoApp handles all of the translation of parameters and the return value to and from XML.

SOAP tags additionally require that each required and optional parameter be assigned a type using the -Type parameter and that the return type of the tag be specified using the -ReturnType parameter. The parameter and return types are used to automatically translate incoming SOAP requests into appropriate Lasso data types and to properly describe the return value.

When called, remote procedure call tags will be executed using the permissions of the Anonymous user. If the tags require additional permissions a username and password must be written into an [Inline] ... [/Inline] container within the tag or the tag must accept a username and password as parameters.

Tags are called within the context of a page load of the RPC.LassoApp. Tags can access global variables, but will not be able to access any page variables from the page where they were defined. RPC and SOAP tags function essentially as asynchronous tags described elsewhere in this chapter.

Remote procedure calls are well suited to a number of different applications. See *Chapter 29: XML* in the Lasso 7 Language Guide for more information. Some possible applications of remote procedure calls include:

- Serving news stories to remote servers. For example, creating a system where other Web sites can show the latest news stories automatically.
- Performing administrative tasks on remote servers. Tags can be created which perform periodic administrative tasks and then those tasks can be triggered using XML-RPC or SOAP calls.
- Integrating with remote systems that communicate via XML-RPC or SOAP. Both Windows 2000 and Mac OS X have systems for sending XML-RPC or SOAP calls and processing the results.

To create a remote procedure call tag:

Use the -RPC parameter in the opening [Define_Tag] tag. In the following example a method Example.Fortune is created which returns a random message each time it is called. Since the tag will not have access to page variables the array of messages is created inside the tag.

```
[Define_Tag: 'Example.Fortune', -RPC]
  [Local: 'Messages' = (Array: 'You will go on a long boat trip.',
    'You will meet a long lost friend',
    'You will strike it rich in the stock market')]
  [Local: 'Index' = (Math_Random: -Min=1, -Max=(#Messages->Size + 1))]
  [Return: #Messages->(Get: #Index)]
[/Define_Tag]
```

The tag can be called from a remote Lasso 7 server using the [XML-RPC] tags. A call to the Example.Fortune remote procedure on the server at http://www.example.com/ would look like as follows.

```
[Variable: 'Result' = XML_RPC->(Call: -Method='Example.Fortune',
  -URI='http://www.example.com/RPC.LassoApp')]
[Variable: 'Result']
```

The result will be one of the messages from the Messages array.

```
➜ You will meet a long lost friend.
```

To create a remote proecedure call tag with complex data types:

The previous example demonstrated how a remote procedure call tag could be created and called using a simple tag which accepted no parameters and returned a string result. Remote procedure calls can be used with any number of parameters including any of Lasso's built-in data types such as array, map, boolean, integer, decimal, etc.

In the following example a method Example.TopStories is created that returns an array of formatted URLs for the top stories from a Web site. An optional -Count parameter allows the number of top stories to be returned to be specified. The top stories are found by finding all records in the Stories table of the News database and sorting the results first by Priority then by DateTime.

```
[Define_Tag: 'Example.TopStories', -Optional='Count']
  [Local: 'Results' = (Array)]
  [If: !(Local_Defined: 'Coun't)]
    [Local: 'Count' = 10]
  [/If]
  [Inline: -Findall,
      -Database='News',
      -Table='Stories',
      -SortField='Priority', -SortOrder='Descending',
      -SortField='DateTime', -SortOrder='Descending',
      -MaxRecords=#Count]

  [Records]
    [#Results->(Insert: '<a href=\"' + (Field: 'URL') + '\">' + (Field: 'Headline') + '</a>')]
  [/Records]

  [Return: #Results]
[/Define_Tag]
```

The tag can be called from a remote Lasso 7 server using the [XML-RPC] tags. A call to the Example.TopStories remote procedure on the server at http://www.example.com/ which requests the top 3 stories would look like as follows.

```
[Variable: 'Result' = (XML_RPC: (Array: -Count=3))->(Call:
  -URI='http://www.example.com',
  -Method='Example.TopStories')]
[Variable: 'Result']
```

The result will be an array of the top three stories from the database each formatted as a URL linking to the page which contains the story.

```
➜ (Array: (<a href="http://www.example.com/story.lasso?id=106">Annual Results</a>),
    (<a href="http://www.example.com/story.lasso?id=105">Shareholder Meeting</a>),
    (<a href="http://www.example.com/story.lasso?id=104">Company Picnic!</a>))
```

To create a SOAP tag:

Use the -SOAP parameter in the opening [Define_Tag] tag. In the following example a method Example.Repeat is created which returns baseString repeated multiplier number of times. Both -Required parameters are followed by -Type parameters and the -ReturnType for the tag is specified.

```
[Define_Tag: 'Example.Repeat', -SOAP,
    -Required='baseString', -Type='string',
    -Required='multiplier', -Type='integer,
    -ReturnType='string']
  [Return: (#baseString * #multiplier)]
[/Define_Tag]
```

The tag can be called from a remote server server that supports SOAP.

Asynchronous Tags

Asynchronous tags are LDML process tags that are executed in a separate thread from the main portion of the page. Lasso does not have to wait for completion of an asynchronous tag before processing and serving the remainder of the format file in which the tag is called.

Since asynchronous tags usually finish executing after a page has been served to the site visitor they cannot return values or modify the page variables for the format file from which they were called.

Asynchronous tags are usually used in one of the following situations:

- To perform database actions which are a side effect of loading a format file, but the results of which are not required for serving the file to the current site visitor.
- To create a background process that periodically checks for certain conditions and performs a database action or sends an email if that condition is met.

Asynchronous tags can be created using the [Define_Tag] ... [/Define_Tag] tags. Newly defined tags will be available below the point where they are defined in a format file. They can be used as many times as needed.

Note: There is no control over when an asynchronous tag will be executed. Depending on how busy the server is the tag may be executed immediately or may be delayed until after the current page is served to the client. The order of execution of asynchronous tags should never be assumed.

Defining Tags

A new asynchronous tag is defined using the [Define_Tag] ... [/Define_Tag] container tags. The opening [Define_Tag] tag requires the name of the new substitution tag to be defined and the second parameter should be -Async which specifies that the tag should be called asynchronously. All of the LDML code between the two tags is stored and is executed each time the tag is called.

In the following example, a tag [Ex_SendEmail] is defined which sends an email to an example email address for John Doe, johndoe@example.com. The tag is defined within a LassoScript and the second parameter is set to True to ensure that the tag will be called asynchronously.

```
<?LassoScript
  Define_Tag: 'Ex_SendEmail', -Async;
    Email_Send: -Host='mail.example.com',
      -To='johndoe@example.com',
      -From='lasso@example.com',
```

```
        -Subject='Sample Email',
        -Body='This email was sent from a custom tag.';
    /Define_Tag;
?>
```

This tag can be called like any LDML process tag within the format file where the tag is defined. The following code calls this [Ex_SendEmail] so an email will be sent to johndoe@example.com each time the page with this code is loaded in a Web browser.

```
[Ex_SendEmail]
```

The code immediately following this tag is executed immediately without waiting for the tag to complete. The email will be queued for sending shortly after the page is finished executing and is served to the client.

Page Variables

None of the page variables which are defined when an asynchronous tag is called are available within the asynchronous tag. The only variables which are available to a custom asynchrnous tag are server-wide global variables. Any values which are going to be used by an asynchronous tag should be set using the [Global] tag.

Calling Custom Tags

Only custom tags which are defined in the LassoStartup folder can be called by an asynchronous tag. Tags which are defined in the same format file as the asynchronous tag definition or call cannot be called by an asynchronous tag.

Custom tags can be defined within the body of an asynchronous tag if needed. These custom tags will be deleted as soon as the asynchronous tag finishes executing.

Background Processes

Asynchronous tags can be used to create background processes that continue to run independent of the visitors to a Lasso-powered Web site. An asynchronous tag will continue executing until the end of the tag body or a [Return] tag is reached. By putting an asynchronous tag into an infinite loop it will continue to run until Lasso Service is quit.

Warning: There is no way to stop an asynchronous tag from executing once it is started. Care should be taken to ensure that any background processes which are implemented with asynchronous tags are well behaved.

The [Sleep] tag can be used to pause execution of an asynchronous tag for a number of milliseconds. The asynchronous tag consumes virtually no resources while it is paused.

Most background processes are started by a format file within the LassoStartup folder. This ensures that the background process runs from when Lasso Service starts up until it is quit.

To create a background process:

Place the following code in a format file within the LassoStartup folder. This code will be executed the next time Lasso Service is started.

Two global variables are created. Since these variables are created in the LassoStartup folder they can be read and set from any page which is executed by Lasso. The first global variable Ex_Background_Pause can be used to pause the background task if it is set to True. The second global variable Ex_Background_Kill can be used to kill the background task if it is set to True.

```
[Global: 'Ex_Background_Pause' = False]
[Global: 'Ex_Background_Kill' = False]
```

These variables are not required to create a background task, but are useful for debugging and to kill a runaway task. By setting the appropriate variable to True in any format file the task can be paused or killed.

The task itself is defined in a [Define_Tag] ... [/Define_Tag]. Notice that the naming convention has the name of the tag which defines the task Ex_Background as the first part of the name of the variables associated with the task. The task contains a while loop that checks the Ex_Background_Kill variable and a conditional that checks the Ex_Background_Pause variable. After each execution, the tag pauses for 15 seconds (15000 milliseconds).

```
[Define_Tag: 'Ex_Background', True]
  [While: (Variable: 'Ex_BackGround_Kill) != True]
    [If: (Variable: 'Ex_Background_Pause') != True)]
      ... Perform Task ...
    [/If]
    [Sleep: 15000]
  [/While]
[/Define_Tag]
```

The task is started by calling the [Ex_Background] tag immediately after it is defined. The task starts executing and does not stop until Lasso Service is quit or the variable Ex_Background_Kill is set to True.

```
[Ex_Background]
```

It is important not to call the [Ex_Background] tag more than once or else multiple instances of the background task will be created.

Background tasks can be made more robust by:

- Adding a variable which is set when the background task is executed so it cannot be executed again.
- Adding variables which control how long the background task sleeps.
- Outputting to the console window with [Log: -Window] ... [/Log] or to a log file in order to track the progress of a background task.

Overloading Tags

Lasso provides the ability to create several versions of a tag each with a criteria that dictates when it should be called. Tag overloading makes several advanced techniques possible.

- **Data Types** – Different tags can be created which operate only when their parameters are of a certain data type. The logic of each tag can be made simpler by removing laborious [Select] ... [Case] ... [/Select] statements.
- **Redefine Existing Tags** – Existing tags can be redefined with a specified criteria. The new version of the tag will be called only when the criteria is met, but the old version of the tag is still available. The source code for the original tag is not needed and even built-in tags can be redefined.
- **Debug Tags** – Tags can be created which output debugging information when a page variable is set appropriately. A page can be debugged and then all status messages can be suppressed by resetting the variable.

When a given tag is called, Lasso will check each tag with the same name in turn until the criteria of one of the tags is met. A tag with no criteria will always execute. All built-in tags will always execute when called.

The -Priority and -Criteria parameters of the [Define_Tag] tag will be discussed followed by examples of how to use those parameters to create systems of overloaded custom tags.

Important: In Lasso Professional 7 many built-in tags which comprise the core of the language can not be overloaded. See the LDML Reference for a complete list of tags that cannot be overloaded.

Priority

The placement of each custom tag in the list of tags in the calling chain can be specified using the -Priority parameter of the [Define_Tag] tag. The following three priorities are available.

- **Replace** – The tag will replace any tags of the same name. Only the newly defined tag will be called when a tag of the given name is called. This allows existing tags to be completely redefined. Aliases and synonyms of the replaced tag will not be redefined.
- **High** – The tag will be placed at the front of the calling chain. The criteria of this tag will be checked first to see if it can be called. If another tag is defined with high priority after this tag then that tag will actually be checked first.
- **Low** – The tag will be placed at the end of the calling chain. The criteria of this tag will be checked only after all other tags have been checked. If another tag is defined with low priority after this tag then that tag will actually be checked last. If the tag is placed after a built-in tag or a custom tag with no criteria then the tag will never be called.

Note: By default, tags have no priority. They must have unique names and will be the sole tag in the calling chain.

To replace a built-in tag:

A built-in tag can be replaced by creating a new tag that has a -Priority of Replace. This technique can be used to redefine a custom tag or to redefine a built-in tag.

Note: Blue World does not support systems which have built-in tags replaced. It is always advisable to create new tag names rather than redefining existing tags.

For example, the [Form_Param] tag could be redefined so it only retrieved parameters that were sent using the Post method in an HTML form. This is done by inspecting the [Client_PostParams] tag and returning those items from the array that match the parameter to the tag.

```
<?LassoScript
  Define_Tag: 'Form_Param', -Priority='Replace', -Require='name';
    Local: 'id_array' = (Client_PostParams)->(Find: #name);
    Local: 'output' = '';
    Iterate: #id_array, (Local: 'id_item');
      #output += (Client_PostParams)->(Get: #id_item)->Second + '\r';
    /Iterate;
    #output->(RemoveTrailing: '\r');
    Return: #output;
  /Define_Tag;
?>
```

This tag can now be used anywhere on a page to get access to the parameters that were passed through a form using the Post method. Since the tag uses the [Client_PostParams] tag it can even be used within nested [Inline] tags.

If this tag is defined on a page then it will replace the [Form_Param] tag only until the end of the page. If this tag is defined in the LassoStartup folder then it will replace the [Form_Param] tag for all users of the site. The [Action_Param] tag is not modified by redefining the [Form_Param] tag even though they are aliases.

Criteria

If a tag has a -Criteria parameter defined then it will only be called when the specified criteria are met. If the criteria are not met then the next tag in the calling chain will be consulted or an error will be generated.

The -Criteria parameter should be a conditional expression that returns True of False. It is called within the environment of the tag being defined and has access to local variables created by the -Required and -Optional parameters and to the [Params] array. The -Criteria parameter can also reference page variables.

Required parameters specified by the -Required tag are checked prior to the -Criteria parameter. If a tag is missing a -Required parameter then a syntax error is returned and no further checking of the tags in the calling chain occurs. -Optional parameters should be used with appropriate an appropriate -Criteria expression to require parameters only on certain tags within a calling chain.

To execute a tag when it is called with a parameter of a given type:

Create the tag with a -Required parameter and a -Criteria expression that checks the type of the local defined by the -Required parameter. The following tag prints a formatted message only when it is called with a string parameter.

```
[Define_Tag: 'Ex_Print',
    -Priority='High',
    -Required='myParam',
    -Criteria=(#myParam->Type == 'string')]
  [Return: '(String: \'' + #myParam + '\')]
[/Define_Tag]
```

When this tag is called with a string parameter the formatted output is generated, otherwise a syntax error is generated.

```
[Ex_Print: 'Text'] ➜ (String: 'Text')

[Ex_Print: 123.456] ➜ Syntax Error
```

Now, an additional tag can be added with the same name that executes when it is called with a parameter of a different data type. The following version of [Ex_Print] will be called when the parameter is of type decimal.

The -Priority of this tag is set to High ensuring that it is called before the other version of [Ex_Print] in the calling chain.

```
[Define_Tag: 'Ex_Print',
    -Priority='High',
    -Required='myParam',
    -Criteria=(#myParam->Type == 'decimal')]
  [Return: '(Decimal: ' + #myParam + ')]
[/Define_Tag]
```

When this tag is called with a decimal parameter the formatted output is generated. When it is called with a string parameter the prior version of the tag is used and its formatted output is generated. If the tag is called with a parameter of a different data type then a syntax error is generated.

```
[Ex_Print: 'Text'] ➜ (String: 'Text')

[Ex_Print: 123.456] ➜ (Decimal: 123.456)

[Ex_Print: 123] ➜ Syntax Error
```

Additional tags can be created for each of the built-in data types: arrays, dates, maps, pairs, integers, boolean values, etc.

Rather than returning a syntax error when an unknown data type is specified as a parameter to the tag, a version of the tag can be created that accepts parameters of any type. The following version of [Ex_Print] is used for unknown data types. The -Priority is set to Low ensuring that this version of the tag is checked after all other versions of [Ex_Print] in the calling chain.

```
[Define_Tag: 'Ex_Print',
    -Priority='Low',
    -Required='myParam']
  [Return: '(Unknown: ' + (String: #myParam) + ')]
[/Define_Tag]
```

When this tag is called with a parameter of type date for which no individual version of the tag has been created the Unknown output is generated.

```
[Ex_Print: (Date)] ➜ (Unknown: 5/15/2002 12:34:56)
```

The real power of this type of system of tags—which are only used when called with a parameter of a certain data type—is that it can be expanded by third parties to include their own custom data types. For example, if a new data type is created that represents currency then a new version of the [Ex_Print] tag could be created as well. The end-user will see that [Ex_Print] now works for the currency data type and doesn't have to be aware of the mechanism which has been used to extend this tag to the additional data type.

Libraries

Libraries can be used to package custom tags and custom types into a format which is easy for any Lasso developer to incorporate into a Lasso-powered Web site.

The following types of libraries can be created:

- **Library Format File** – A set of custom tag and custom type declarations can be stored in a format file and then included in any other Lasso format file using the [Library: 'library.lasso'] tag. This is a good way to create and use a library file whose defined tags and types will only be needed on a few pages in a site.
- **LassoStartup Format File** – A set of custom tag and custom type declarations can be stored in a format file placed within the LassoStartup folder. After Lasso Service is restarted all tags, types, and page variables which are defined within the format file will be available to all format files which are executed on the server.

4

Chapter 4
Custom Types

This chapter introduces custom data types and shows how they can be created using LDML tags.

- *Overview* introduces the concepts behind custom data types.
- *Custom Types* describes how to create new data types including information about callback tags and inheritance.
- *Libraries* describes how to package sets of custom types for distribution.

Overview

Lasso Professional 7 allows Web developers to extend Lasso Dynamic Markup Language (LDML) by creating custom data types programmed using LDML tags.

LDML custom data types have the following features:

- Tags for custom types operate just like built-in LDML member tags. They can be used in nested expressions, return data of any type, and allow the use of encoding keywords.
- Custom types are fully object-oriented. Custom types can inherit properties from other custom types.
- Custom types can provide support for the LDML comparison symbols and automatic casting.
- They can be created in any Lasso format file and used instantly.
- They are written in LDML. No programming experience or knowledge of a programming language other than LDML is required.
- They can be collected into libraries of tags which can be loaded into any format file using the [Library] tag.

- They can be defined in a format file or library within the LassoStartup folder, making them available to all pages processed by Lasso.

Naming Conventions

Custom types should be named using a combination of letters, numbers, and the underscore character, but should never start with the underscore character. In order to prevent confusion between custom types created by different developers, all custom type names should start with an identifier for the author of the custom type, followed by an underscore, then the name of the custom type.

The member tags of a custom type do note need a prefix since member tags only need to be unique within each data type. In fact, it is recommended to use the same names as built-in member tags for custom member tags if the functionality is equivalent. For example, a custom data type might implement [Type->Get] and [Type->Size] member tags.

If either a member tag or instance variable of a custom tag starts with an underscore then the tag or variable will be private to the data type. Private member tags and instance variables will not be listed in the [Null->Properties] or [Null->Dump] output for the type and can only be accessed using the [Self] tag within the data type.

Table 1: Tags for Creating Custom Data Types

Tag	Description
[Define_Tag]	Defines a new new member tag within a type definition.
[Define_Type]	Defines a new data type. Requires a single parameter, the name of the type to be defined. Optional second parameter defines a custom type which should be inherited from.
[Local]	Sets or retrieves the value of a member variable within a custom type definition.
[Local_Defined]	Checks to see if a member variable has been defined within a custom type definition.
[Locals]	Returns a map of all the member variables which have been defined within a custom type definition.
[Params]	Returns an array of all the parameters which were passed to the custom tag.
[Self]	Returns a reference to the current data type instance.
[Self->Parent]	Returns a reference to the parent type for the current data type instance. For use within custom type declarations with inheritance.

Note: In addition to the listed tags all of the tags which are used for creating custom tags are used when creating member tags.

Custom Types

Custom data types can be created in LDML using the [Define_Type] ... [/Define_Type] tags. Newly defined types will be available below the point where they are defined in a format file.

See the section on *Libraries* for information about how to create libraries of types, load types in LassoStartup, and create types which can be used by any format file.

Defining a Type

A new data type is defined by specifying its name in the opening [Define_Type] tag. The body of the [Define_Type] ... [/Define_Type] tags contains code which will be executed each time a new instance of the data type is created.

For example, a new data type Ex_Dollar could be created which will store dollar amounts. The basic type definition is as follows. Each of the parts of this definition are discussed in more detail in the sections that follow.

```
[Define_Type: 'Ex_Dollar']
  [Local: 'Amount' = 0]
  [(Local: 'Amount')->(SetFormat: -DecimalChar='.', -Precision=2)]

  [Define_Tag: 'onCreate', -Optional='Amount']
    [If: (Local_Defined: 'Amount')]
      [Self->'Amount' = (Decimal: #Amount)]
      [Self->'Amount'->(SetFormat: -DecimalChar='.', -Precision=2)]
    [/If]
  [/Define_Tag]

  ... Member Tags ...

[/Define_Type]
```

The [Define_Type] ... [/Define_Type] tags define a tag with the same name as the data type. Each time a new instance of [Ex_Dollar] is created the [Ex_Dollar->onCreate] tag is called to initialize the instance.

The code within [Define_Type] ... [/Define_Type] should be used to set up a generic instance of the type. The code within [Ex_Dollar->onCreate] should be used to create a specific instance of the t ype based on the parameters passed to the [Ex_Dollar] tag.

Instance Variables

A data type can contain definitions for local variables within the [Define_Type] ... [/Define_Type] tags. These local variables are called instance variables since their values are stored separately for each instance of the data type which is created.

In the example above, the local variable Amount is created. This variable will store a dollar amount, the current value of the data type. Each time a new instance of [Ex_Dollar] is created, a new instance of the Amount instance variable will be created. For example, the following two lines create two variables each of which stores a value of type Ex_Dollar. Each stores its own independent Amount.

```
[Variable: 'Price' = (Ex_Dollar: 10)]

[Variable: 'Tax' = (Ex_Dollar: 0.93)]
```

Instance variables can be referenced explicitly by name using the member symbol -> with the name of the instance variable. The values for the Amount instance variable can be retrieved from each of the Ex_Dollar amounts defined above using the following code.

```
[Output: (Variable: 'Price')->Amount] ➜ 10.00

[Output: (Variable: 'Tax')->Amount] ➜ 0.93
```

The quotes around the variable name Amount can be omitted if the type does not define a tag with the same name as the member variable. Usually, $Price->'Amount' is equivalent to $Price->Amount.

Private Variables

Instance variables which start with an underscore are private variables that can only be accessed from within the custom type. Private variables should be used for any values which are stored internally and do not need to be acessed from outside the custom type.

Private variables are not copied with the data type or serialized. Private variables should only be used to store static data that does not need to be propagated to new instances of the data type and does not need to survive being stored in a session and retrieved.

For example, in the prior example, the Amount variable can be renamed _Amount to make it private to the data type.

```
[Define_Type: 'Ex_Dollar']
  [Local: '_Amount' = 0]
  [(Local: '_Amount')->(SetFormat: -DecimalChar=',', -Precision=2)]
```

```
[Define_Tag: 'onCreate', -Optional='Amount']
  [If: (Local_Defined: 'Amount')]
    [Self->'_Amount' = (Decimal: #Amount)]
    [Self->'_Amount'->(SetFormat: -DecimalChar=',', -Precision=2)]
  [/If]
[/Define_Tag]

... Member Tags ...

[/Define_Type]
```

Now, the following code will produce an error since the _Amount private variable cannot be seen outside of the custom data type.

```
[Output: (Variable: 'Price')->_Amount] ➜ Syntax Error
```

Once private variables have been defined they can be accessed from within the custom type using Self->'_VariableName'.

Member Tags

Built-In Member Tags

Each custom type can automatically make use of any of the tags of the null data type. These tags are detailed in *Table 2: Built-In Tags*. These tags are used by Lasso to provide information about data of any type and to provide efficient storage for custom data types. None of the tags in this table should be overriden by the custom data type.

In addition to these built-in member tags there are several tags that are defined as placeholders on the null data type. The [Null->Size] and [Null->SetFormat] tags are defined for every data type. [Null->Size] always returns 0 and [Null->SetFormat] is a placeholder that returns no value if called. Either of these tags can be overridden using custom member tags.

Note: It is desirable for custom data types to create custom [Type->Get] and [Type->Size] member tags so the [Iterate] ... [/Iterate] tags will function properly.

Table 2: Built-In Member Tags

Tag	Description
[Null->DetachReference]	Detaches the variable from the instance of the data type.
[Null->FreezeType]	Freezes the type of a variable. After calling this tag the current variable cannot be cast to another data type.
[Null->FreezeValue]	Freezes the value of a variable, essentially creating a read-only variable. After calling this tag the current variable cannot have its value changed.
[Null->IsA]	Accepts a single parameter which is the name of a type. Returns True if the parameter matches the name of the current data type or any of its parent data types.
[Null->Parent]	Returns a references to the parent type of the current data type instance.
[Null->Properties]	Returns a pair which contains a map of all the instance variables and a map of all the member tags defined for the data type.
[Null->RefCount]	Returns the number of variables that currently point at the instance of the data type.
[Null->Serialize]	Returns a bit-stream representation for the data type. This tag can be used to store a custom data type in a database or to pass it from page to page as an action parameter.
[Null->Type]	Returns the type which was specified when the custom type was created.
[Null->Unserialize]	Accepts a single parameter which is a bit-stream created by [Null->Serialize]. This tag modifies the variable on which it is called by setting it to the custom data type represented by the bit-stream parameter.

See the Lasso 7 Language Guide for more information about using these tags.

Custom Member Tags

Each custom type can define member tags which can be called to modify the value stored in an instance of the custom type or to output values from an instance of the custom type.

Member tags are defined within the [Define_Type] ... [/Define_Type] tags for the custom type using [Define_Tag] ... [/Define_Tag] tags. The syntax for creating member tags is the same as that for creating custom tags. However, member tags cannot be called asynchronously. The [Define_Tag] tag for a member tag should never have -Async as the value for the second parameter.

The [Self] tag allows member tags to reference the current instance of the data type. This allows member tags to call other defined member tags or to set or retrieve values stored in instance variables. See the example of defining a custom member tag below for more information.

The [Self] tag also allows access to instance variables and private variables that are stored within the custom data type. The [Self] tag is the only method of accessing private variables since they are not availablef rom outside the data type.

Custom member tags which are named starting with an underscore are private member tags. These tags can only be called using the [Self] tag. Private member tags should be used for helper tags that are called by public member tags, but do not need to be called directly from outside the data type.

To define custom member tags:

Two custom tags will be defined for the Ex_Dollar custom type. The [Ex_Dollar->Set] tag will accept a single parameter, cast it to decimal, and store it in the Amount instance variable. The [Ex_Dollar->Get] tag will simply return the value of the Amount instance variable formatted as a dollar amount.

- The [Ex_Dollar->Set] member tag is defined within the body of the [Define_Type] ... [/Define_Type] tags. It checks that there is at least one parameter in the [Params] array. The [Self] tag is a reference to the current instance of the Ex_Dollar data type, so the (Self->'Amount') statement is a reference to the Amount instance variable.

```
<?LassoScript
  Define_Tag: 'Set';
    If: (Params) && ((Params)->Size > 0);
      (Self->'Amount') = (Decimal: (Params)->(Get:1));
    /If;
  /Define_Tag;
?>
```

- The [Ex_Dollar->Get] member tag is defined within the body of the [Define_Type] ... [/Define_Type] tags. It appends a dollar sign $ to the value in the Amount instance variable and returns the value. The [Self] tag is a reference to the current instance of the Ex_Dollar data type, so the (Self->'Amount') statement is a reference to the Amount instance variable.

```
<?LassoScript
  Define_Tag: 'Get';
    Return: '$' + (Self->'Amount');
  /Define_Tag;
?>
```

To call a custom member tag:

Custom member tags are called in the same way that the member tags of the built-in data types are called. The Ex_Dollar type has two member tags [Ex_Dollar->Get] and [Ex_Dollar->Set]. They are used to set and retrieve dollar amounts in the following example.

```
[Variable: 'Price' = (Ex_Dollar: 100)]
<br>[Output: (Variable: 'Price')->Get]
[(Variable: 'Price')->(Set: 19.95)]
<br>[Output: (Variable: 'Price')->Get]

➜ <br>$100.00
<br>$19.95
```

Callback Tags

Each custom type can define a number of callback tags using the [Define_Tag] ... [/Define_Tag] tags within the [Define_Type] ... [/Define_Type] definition for the type. These callback tags will be executed with appropriate parameters when the data type is cast to another type, a new instance is created, or an instance is destroyed.

Table 3: Callback Tags details the tags that are available. These tag names are reserved. No member tags with these names should be defined. These tags are not normally called by a Lasso developer, they are called automatically by Lasso in the specified situation. Although there is no protection to prevent a Lasso developer from calling these tags directly, results should be considered undefined if they do.

The primary callback tags are shown in *Table 3: Callback Tags*. Additional callback tags allow the overriding of built-in symbols. These tags are described in the next section.

Table 3: Callback Tags

Tag	Description
[Null->onConvert]	Called when the instance is cast to another data type. Accepts a single parameter, the name of the type to which the value should be converted. The return value should be the converted value or Null if no conversion was possible.
[Null->onCreate]	Called immediately after a new instance is created. This tag has full access to the variables and member tags defined within the [Define_Type] … [/Define_Type] tags.
[Null->onDestroy]	Called before the custom tag is destroyed, usually at the end of the current format file or tag execution.
[Null->_UnknownTag]	Called when an unknown member tag is referenced. The [Tag_Name] tag can be used to decide what tag name was referenced.

Note: These callback tags are not included in the LDML 7 tag list. They are intended to be called by Lasso automatically rather than being called like other member tags.

onCreate Callback

The [Null->onCreate] callback tag is called after a new instance of a type is created. It is called once for each instance of a type with any parameters that were passed to the tag that created the type.

For example, when the tag [Ex_Dollar] is used to create a new instance of the dollar type the following steps are performed.

1. The code within the [Define_Type] ... [/Define_Type] container is executed, creating all the custom tags and instance variables for the type.
2. The [Ex_Dollar->onCreate] tag is called with the parameters passed to the [Ex_Dollar] tag to set up the particular instance of the type.

Since the callback tag is called after the code within the [Define_Type] ... [/Define_Type] container is processed, the [Null->onCreate] tag has access to the [Self] tag and to each of the member tags which have been defined for the current type.

Order of operation:

A new instance of a custom type is created by calling the creator tag for the type which has the same name as the type. For example, to create a new Ex_Dollar type the [Ex_Dollar] tag must be called.

```
[Ex_Dollar: 10]
```

1 The body of the [Define_Type] ... [/Define_Type] tags for the Ex_Dollar type are executed. Local instance variables are defined and all member tags are defined.

2 If the [Ex_Dollar->onCreate] callback tag is defined then it is called.

To define a [Null->onCreate] callback tag:

The Ex_Dollar data type is too simple to require an [Ex_Dollar->onCreate] callback tag. All the initialization which is needed is performed in the creator tag. However, for debugging purposes it might be nice to know each time an instance of the new data type is created. The following [Ex_Dollar->onCreate] tag logs the current value of the instance variable Amount each time a new instance of the data type is created.

```
[Define_Tag: 'onCreate']
  [Log: -Window] Create Ex_Dollar: [Output: Self->'Amount'].[/Log]
[/Define_Tag]
```

onConvert Callback

The [Null->onConvert] callback tag is called when an instance of a custom type is cast to another data type. This tag will be called when an instance of a custom type is used in an expression with built-in data types that requires an integer, decimal, or string value. Each custom type must support being cast to the string data type and should support being cast to the decimal or integer data types if possible.

The [Null->onConvert] callback is called with the name of the type to which the current instance is being converted. The type is the same value as returned by the [Null->Type] tag and could identify any built-in type or any custom type.

If the name of the type is not recognized then the [Null->onConvert] tag should return Null. Lasso will attempt to convert the custom data type using another method or will throw an error.

To define a [Null->onConvert] callback tag:

The [Ex_Dollar->onConvert] callback tag is called when an Ex_Dollar amount is cast to any other data type. If the value is cast to a decimal or an integer then the callback tag will cast the value in the Amount instance variable to the appropriate data type. If the value is cast to a string then the [Ex_Dollar->Get] member tag which was defined previously will be called. Otherwise, the callback tag will return Null instructing Lasso that the conversion is not supported.

```
<?LassoScript
  Define_Tag: 'onConvert';
    Local: 'Type' = (Params)->(Get: 1);
    If: (Local: 'Type') == 'String';
      Return: (Self->Get);
    Else: (Local: 'Type') == 'Integer';
      Return: (Integer: (Self->'Amount'));
    Else: (Local: 'Type') == 'Decimal';
      Return: (Decimal: (Self->'Amount'));
    /If;
    Return Null;
  /Define_Tag;
?>
```

In the following code a variable Price is set to a value of type Ex_Dollar. Then that variable is cast to different data types. Notice that the [Output] tag automatically casts all values to the string data type.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[Output: (Variable: 'Price')]
<br>[Output: (Integer: (Variable: 'Price'))]
<br>[Output: (Decimal: (Variable: 'Price'))]

➜ <br>$19.95
<br>20
<br>19.95
```

onDestroy Callback

The [Null->onDestroy] callback tag is the last member tag called for each instance of a custom type. The [Null->onDestroy] callback tag allows any cleanup code that needs to be performed to be executed before the tag is purged from memory. The [Null->onDestroy] tag is called once for each instance of a custom type.

The [Null->onDestroy] callback tag is called in the following instances.

- If a custom type literal is created and not stored in a variable, the instance is destroyed as soon as the current tag completes.

  ```
  [Output: (Ex_Dollar: 10.0)]
  ```

- If a custom tag is created within the [Define_Tag] ... [/Define_Tag] tags of a custom tag declaration and stored in a local variable then the instance is destroyed as soon as the custom tag completes.
- If a custom tag is created within the [Define_Type] ... [/Define_Type] tags of a custom type or is stored in an instance variable within a custom type then the instance is destroyed as soon as the custom type within which it is stored is destroyed.

- If a custom type is stored within a page variable then it will be destroyed as soon as the page finishes executing, but before it is served to the site visitor.

To define a [Null->onDestroy] callback tag:

The Ex_Dollar data type is too simple to require an [Ex_Dollar->onDestroy] callback tag. The instance variable Amount and each of the member tags will be destroyed automatically by Lasso. However, for debugging purposes it might be nice to know each time an instance of the new data type is destroyed. The following [Ex_Dollar->onDestroy] tag logs the current value of the instance variable Amount each time a new instance of the data type is destroyed.

```
[Define_Tag: 'onDestroy']
  [Log: -Window] Destroy Ex_Dollar: [Output: Self->'Amount'].[/Log]
[/Define_Tag]
```

Unknown Tag Callback

The [Null->_UnknownTag] callback tag is called when a tag that does not exist for the current data type is referenced. This callback tag allows a custom data type to respond to member tags which are not explicitly created. The tag name which was called can be retrieved using the [Tag_Name] tag.

None of the callback tags are ever passed to the [Null->_UnknownTag] callback. Callback tags must be defined explicitly in order to be implemented.

Order of operation:

When a member tag is called on a custom type:

1. If a member tag with that name is defined then it is executed.
2. If an instance variable with that name is defined then its value is returned.
3. If no member tag or instance variable with that name is defined then the [Null->_UnknownTag] callback is executed.
4. If the unknown tag callback is not defined then an error is returned.

To define a [Null->_UnknownTag] callback tag:

The Ex_Dollar data type could implement a conversion to different currencies using the unknown tag callback.

Assume that there is a tag [Currency_Convert] which accepts a value, a -From parameter with the code for what currency to convert from, and a -To parameter with the code for what currency to convert to. The tag uses data from a site on the Internet to get accureate real-time conversion rates.

Rather than coding in all currency codes explicitly and unknown tag callback can be used to pass any unknown member tags to the [Currency_Convert] tag. An error will be returned if the tag name is not a valid currency code.

```
[Define_Tag: '_UnknownTag']
  [Local: 'Code' = (Tag_Name)]
  [Local: 'Result' = (Currency_Convert: (Decimal: Self->'Amount'),
    -From='USD', -To=#Code)]
  [Return: (Decimal: #Result)]
[/Define_Tag]
```

The following code would now work to convert the U.S. currency represented by the [Ex_Dollar] type to U.K. Pounds represented by UKP.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[Output: (Variable: 'Price')->(UKP)]
```

➜
31.24

Symbol Overloading

Lasso allows complex expressions using math and string symbols to be specified as tag parameters. In addition, a set of assignment symbols allow a variable to be modified in place without returning a value. A list of common symbols is shown in *Table 4: Overloadable Symbols*.

Each data type can assign its own meanings to each of the symbols that Lasso provides. For example, the built-in integer and decimal data types use the + symbol for addition while the built-in string data type uses the + symbol for concatenation. In general it is wisest to match the common meanings of the symbols whenever possible. Ideally, the user will be able to use each data type's custom symbols interchangeably with the symbols provided by the built-in data types.

The meaning of corresponding assignment symbols, unary symbols, and binary symbols should be compatible whenever possible. The operation [(Variable: 'myVariable') += 'Value'] should be the same as the operation [Variable 'myVariable' = $myVariable + 'Value'].

Table 4: Overloadable Symbols

Symbol	Description
+	Unary/Binary symbol for addition or concatenation.
-	Unary/Binary symbol for subtraction or deletion.
*	Binary symbol for multiplication or repetition.
/	Binary symbol for division.
%	Binary symbol for modulus.
++	Unary increment symbol prefix or postfix.
--	Unary decrement symbol prefix or postifx.
==	Binary symbol for equality. Returns boolean.
!=	Binary symbol for inequality. Returns boolean.
>	Binary symbol for greater than. Returns boolean.
>=	Binary symbol for greater or equal. Returns boolean.
<	Binary symbol for less than. Returns boolean.
<=	Binary symbol for less or equal. Returns boolean.
>>	Binary symbol for contains. Returns boolean.
=	Assignment symbol.
+=	Addition assignment symbol.
-=	Subtraction assignment symbol.
*=	Multiplication assignment symbol.
/=	Division assignment symbol.
%=	Modulus assignment symbol.

Each of these symbols can be redefined or overloaded for a custom data type. The data type of the left parameter to a binary operator determines which tag is used to perform the operation. If a data type does not support the symbol then the parameter is cast to string and the string symbol is used instead.

Other symbols such as $, #, @ cannot be overloaded. These are core language constructs. The logical symbols ||, &&, and ! cannot be overloaded, but a custom behavior can be defined when a custom data type is cast to boolean.

Callback Tags

Each custom type can define a number of callback tags using the [Define_Tag] ... [/Define_Tag] tags within the [Define_Type] ... [/Define_Type] definition for the type. These callback tags will be executed with appropriate parameters when the data type is used in a complex expression.

Table 5: Comparison Callback Tags, *Table 6: Symbol Callback Tags*, and *Table 7: Assignment Callback Tags* detail the tags that are available. These tag names are reserved. No member tags with these names should be defined. These tags are not normally called by a Lasso developer, they are called automatically by Lasso in the specified situation. Although there is no protection to prevent a Lasso developer from calling these tags directly, results should be considered undefined if they do.

Table 5: Comparison Callback Tags

Tag	Description
[Null->onCompare]	Called when the current instance is used in a comparison expression. Accepts a single parameter, the value to be compared against. Should return 0 if the parameter is equal to the current instance, a positive number if the parameter is greater, or a negative number if the parameter is less. Called for the ==, !=, <, <=, >, >= symbols.
[Null->>>]	Called when an instance is used as the left parameter of a contains symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return True if the right parameter is contained in the current instance.

Note: These callback tags are not included in the LDML 7 tag list. They are intended to be called by Lasso automatically rather than being called like other member tags.

onCompare Callback

The [Null->onCompare] callback tag is called when an instance of a custom type is used as the left parameter of a comparison symbol ==, !=, <, <=, >, or >=. The callback tag is called with the value of the right parameter of the symbol. The result of the tag should be one of the following.

- **Equality** – If the value of the right parameter is equal to the value of the current instance of the custom type then the return value should be 0. This will evaluate to True for the ==, <=, and >= symbols.
- **Less Than** – If the value of the right parameter is less than the value of the current instance of the custom type then the return value should be any number less than 0. This will evaluate to True for the <, <=, and != symbols.
- **Greater Than** – If the value of the right parameter is greater than the value of the current instance of the custom type then the return value

should be any number greater than 0. This will evaluate to True for the >, >=, and != symbols.

If a comparison cannot be made then Null should be returned instead. Lasso will attempt to perform a cast in order to compare the two values instead. If no [Null->onCompare] callback tag is defined then Lasso will attempt to perform a cast in order to compare the two values as well.

The value of the left parameter determines the type of comparison which is used. If a custom type is used as the right parameter in a comparison expression and a built-in data type is used as the left parameter then the custom type is cast to the appropriate built-in data type and the values are compared.

Note: The [Array->Find] and [Array->Sort] member tags use comparisons to determine the found set or order of elements in the array. A custom data type will be searched or sorted according to the results of the [Null->onCompare] callback tag.

To define a [Null->onCompare] callback tag:

The [Ex_Dollar->onCompare] callback tag will simply cast any value that is assigned to it to the decimal data type then compare that value to the value stored in the Amount instance variable.

```
<?LassoScript
  Define_Tag: 'onCompare';
    Local: 'Temp' = (Decimal: (Params)->(Get: 1));
    If: (Local: 'Temp') == (Self->'Amount');
      Return: 0;
    Else: (Local: 'Temp') < (Self->'Amount');
      Return -1;
    Else: (Local: 'Temp') > (Self->'Amount');
      Return 1;
    /If;
  Return Null;
  /Define_Tag;
?>
```

In the following code a variable Price is set to a value of type Ex_Dollar. Then that variable is compared to different data types.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[Output: (Variable: 'Price') == (String: '19.95')]
<br>[Output: (Variable: 'Price') == (Integer: 20)]
<br>[Output: (Variable: 'Price') == (Decimal: 19.95)]

➜ <br>True
<br>False
<br>True
```

Contains Callback

The [Null->>>] callback tag is called when an instance of a custom type is used as the left parameter of a >> comparison symbol. The callback tag is called with the value of the right parameter of the symbol. The result of the tag should be one of the following.

- **True** – If the value of the right parameter is contained within the current instance.
- **False** – If the value of the right parameter is not contained within the current instance.

If the contains operation cannot be performed then Null should be returned instead. Lasso will attempt to perform a cast in order to perform the contains operation. If no [Null->>>] callback tag is defined then Lasso will attempt to perform a cast in order to perform the contains operation as well.

If a custom type is used as the right parameter in a contains expression and a built-in data type is used as the left parameter then the custom type is cast to the appropriate built-in data type and the values are compared.

To define a [Null->>>] callback tag:

The [Ex_Dollar->>>] callback tag will cast any value that is assigned to it to the string data type. If the output from [Ex_Dollar->Get] run on the [Self] tag contains the parameter then True is returned.

```
<?LassoScript
  Define_Tag: '>>';
    Local: 'Temp' = (String: (Params)->(Get: 1));
    Return: (Self->Get) >> #Temp;
  /Define_Tag;
?>
```

In the following code a variable Price is set to a value of type Ex_Dollar. Then that variable is checked to see if it contains $ which it does.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[Output: (Variable: 'Price') >> '$']
```

➜
True

Table 6: Symbol Callback Tags

Tag	Description
[Null->+]	Called when an instance is used as the left parameter of an addition symbol. Accepts a single parameter which is the right parameter of the symbol. If no parameter is specified then the unary symbol is being used.
[Null->-]	Called when an instance is used as the left parameter of a subtraction symbol. Accepts a single parameter which is the right parameter of the symbol. If no parameter is specified then the unary symbol is being used.
[Null->*]	Called when an instance is used as the left parameter of a multiplication symbol. Accepts a single parameter which is the right parameter of the symbol.
[Null->/]	Called when an instance is used as the left parameter of a division symbol. Accepts a single parameter which is the right parameter of the symbol.
[Null->%]	Called when an instance is used as the left parameter of a modulus symbol. Accepts a single parameter which is the right parameter of the symbol.
[Null->++]	Called when an instance is used as the left or right parameter of a unary increment symbol.
[Null->--]	Called when an instance is used as the left or right parameter of a unary decrement symbol.

Note: These callback tags are not included in the LDML 7 tag list. They are intended to be called by Lasso automatically rather than being called like other member tags.

Symbol Callback Tags

The symbol callback tags are called whenever the custom data type is used as the left parameter to one of the built-in symbols +, -, *, /, or % or when the custom data type is used as the lone parameter to the + , ++, - or -- unary symbols. These tags usually return a value of the custom data type, but can return a value of any data type.

For the binary operators, the right parameter to the symbol is provided as the parameter of the callback function and could be of any data type. For the unary operators, no parameter is specified.

If no callback tag is defined for a given symbol then Lasso will attempt to cast values to string and will use the built-in string symbols.

To define a [Null->-] callback tag:

The [Ex_Dollar->-] callback tag will create a new [Ex_Dollar] data type. The value of the new type will be found by either subtracting a value from the Amount instance variable if a parameter is specified or by changing the sign of the Amount instance variable if no parameter is specified.

```
<?LassoScript
  Define_Tag: '-';
  If: (Params->Size > 0);
    Return: (Ex_Dollar: (Self->'Amount') - (Decimal: Params->(Get: 1)));
  Else;
    Return: (Ex_Dollar: (Self->'Amount') * (-1));
  /If;
  /Define_Tag;
?>
```

In the following code a variable Price is initialized with a value of 19.95. Then, 5.95 is subtracted from variable and the result is output. Notice that even though the amount subtracted is a decimal, the result is of type Ex_Dollar and outputs with proper formatting.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[Output: (Variable: 'Price') - 5.95]

➜ <br>$14.00
```

Table 7: Assignment Callback Tags

Tag	Description
[Null->onAssign]	Called when an assignment is made to the current instance from any other data type using the = symbol. This tag should return True if the assignment was successful.
[Null->+=]	Called when an instance is used as the left parameter of an addition assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.
[Null->-=]	Called when an instance is used as the left parameter of a subtraction assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.
[Null->*=]	Called when an instance is used as the left parameter of a multiplication assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.
[Null->/=]	Called when an instance is used as the left parameter of a division assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.
[Null->%=]	Called when an instance is used as the left parameter of a modulus assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.

Note: These callback tags are not included in the LDML 7 tag list. They are intended to be called by Lasso automatically rather than being called like other member tags.

onAssign Callback

The [Null->onAssign] callback tag is called when an instance of a custom type is used as the left parameter of the assignment symbol =. The callback tag is called with the value of the right parameter of the symbol. The tag should attempt to store the value of the right parameter as the new value of the current instance of the custom type. It should return one of the following values.

- **True** – The callback tag should return True if the assignment was successful. This is the sign to Lasso that no further work needs to be done.
- **False** – If for any reason the assignment cannot be performed then the callback tag should return False. Lasso will instead attempt to cast the value of the right parameter to the data type of the left parameter and try the assignment again.

If no [Null->onAssign] callback tag is defined then Lasso will attempt to cast values to the current data type by calling the [Null->onConvert] tag of the right parameter of the assignment operator. For maximum compatibility, each data type should support at least all built-in data types for assignment and conversion.

To define a [Null->onAssign] callback tag:

The [Ex_Dollar->onAssign] callback tag will simply cast any value that is assigned to it to the decimal data type then store that value in the Amount instance variable. This mimics the behavior of the [Ex_Dollar->Set] member tag which was defined previously.

```
<?LassoScript
  Define_Tag: 'onAssign';
    (Self->'Amount') = (Decimal: (Params)->(Get: 1));
  /Define_Tag;
?>
```

In the following code a variable Price is initialized with a value of type Ex_Dollar. The variable is then assigned a string value 19.95 which is cast to a decimal value by the [Ex_Dollar->onAssign] tag called implicitly by Lasso to perform the assignment operator.

```
[Variable: 'Price' = (Ex_Dollar)]
[(Variable: 'Price') = '19.95']
<br>[Output: (Variable: 'Price')->Get]
➜ <br>$19.95
```

Assignment Symbols Callbacks

The [Null->+=], [Null->-=], [Null->*=], [Null->/=], and [Null->%] callback tags are called when an instance of a custom type is used as the left parameter of the corresponding assignment symbol +=, -=, *=, /=, or %=. The callback tag is called with the value of the right parameter of the symbol. The tag should attempt to perform the desired operation and store the value of the right parameter as the new value of the current instance of the custom type. It should return one of the following values.

- **True** – The callback tag should return True if the assignment was successful. This is the sign to Lasso that no further work needs to be done.
- **False** – If for any reason the assignment cannot be performed then the callback tag should return False. Lasso will instead attempt to cast the value of the right parameter to the data type of the left parameter and try the assignment again.

If no callback tag for a given assignment symbol is defined then Lasso will attempt to cast values to the current data type by calling the [Null->onConvert] tag of the right parameter of the assignment operator.

To define a [Null->+=] callback tag:

The [Ex_Dollar->=+] callback tag will simply cast any value that is assigned to it to the decimal data type and add that value to the Amount instance variable.

```
<?LassoScript
  Define_Tag: '+=';
    (Self->'Amount') += (Decimal: (Params)->(Get: 1));
  /Define_Tag;
?>
```

In the following code a variable Price is initialized with a value of type Ex_Dollar and a value of 19.95. Finally, the += symbol is used to add an additional 5.95 to the variable.

```
[Variable: 'Price' = (Ex_Dollar: '19.95')]
[(Variable: 'Price') += '5.95']
<br>[Output: (Variable: 'Price')->Get]
```

➜
$19.95

Inheritance

Custom types can be created which inherit properties from other custom types. Each type which the custom type should inherit from is specified after the name of the custom type in the opening [Define_Type] tag. These are called parent types and the current type being defined is called a child type. Usually, only one parent type is specified.

All instance variables and member tags of the parent types are inherited by the child type. If the child type defines an instance variable or member tag with the same name as one of the parent types then the child's definition overrides the parent's definition.

Custom types can inherit properties from built-in data types. A custom type will inherit any member tags which the built-in type defines, but will not inherit any of the features that require callback functions. It will be necessary to create custom casting and assignment callbacks and to implement any symbols which are desired.

All custom data types inherit from the null data type. The tags of the null data type such as [Null->Type] can be used by any data type within Lasso. These tags can be overridden, but doing so can cause unexpected results.

The member tags and instance variables of the parent tag can be accessed using the [Parent] tag. This tag works like the [Self] tag, but returns the value of the current data type instance as it would be if it were of the parent type.

The creator tag [Null->onCreate] and destructor tag [Null->onDestroy] for each parent data type is called automatically when a new instance of the child data type is created.

To define a custom type that inherits from another custom type:

The Ex_Dollar type which is defined in this chapter only works with U.S. currency and outputs values using the dollar sign $. It is possible to create a sub-type that works with a different type of currency. For example, a new type Ex_UKPounds could be created which inherited from Ex_Dollar, but output values with a British pound symbol £. by overriding the [Ex_Dollar->Get] tag with a new [Ex_UKPounds->Get] tag.

The type is defined as inheriting from Ex_Dollar by specifying Ex_Dollar after the name of the new type in the opening [Define_Type] tag. All the member tags of Ex_Dollar are automatically defined as is the instance variable Amount.

The [Ex_UKPounds->Get] member tag is defined and overrides the equivalent [Ex_Dollar->Get] member tag. The [Self->Parent] tag is used to reference the Amount instance variable from the parent type.

```
<?LassoScript
  Define_Type: 'Ex_UKPounds', 'Ex_Dollar';

    Define_Tag: 'Get';
      Return: '£' + (Self->Parent->'Amount');
    /Define_Tag;

  /Define_Type;
?>
```

The following example sets two variables, one to a value of Ex_Dollar type and the other to a value of Ex_UKPounds type, then outputs both values. The types are converted to strings when they are output and the appropriate [Ex_Dollar->Get] or [Ex_UKPounds->Get] tag is called to format the output.

```
[Variable: 'American'= (Ex_Dollar: 100)]
<br>[Variable: 'American']
[Variable: 'British'= (Ex_UKPounds: 100)]
<br>[Variable: 'British']

➜ <br>$100.00
<br>£100.00
```

Libraries

Libraries can be used to package custom tags and custom types into a format which is easy for any Lasso developer to incorporate into a Lasso-powered Web site.

The following types of libraries can be created:

- **Library Format File** – A set of custom tag and custom type declarations can be stored in a format file and then included in any other Lasso format file using the [Library: 'library.lasso'] tag. This is a good way to create and use a library file whose defined tags and types will only be needed on a few pages in a site.
- **LassoStartup Format File** – A set of custom tag and custom type declarations can be stored in a format file placed within the LassoStartup folder. After Lasso Service is restarted all tags, types, and page variables which are defined within the format file will be available to all format files which are executed on the server.

5

Chapter 5
Advanced Programming Topics

This chapter documents advanced programming techniques.

- *References* shows how references to data can be used to optimize Lasso's speed and memory usage.
- *Global Variables* describes server-wide variables and tags which make working with them easier.
- *Bytes Type* describes the data type which Lasso uses for binary data.
- *Tag Data Type* introduces the tag data type and its member tags.
- *Compound Expressions* shows how tags can be created using simple expression notation.
- *Thread Tools* explains the methodology for synchronizing and sharing resources between threads.
- *Thread Communication* explains how to send messages and data between threads.
- *Network Communication* provides an introduction to the [Net] type which allows for TCP and UDP communication.
- *Post Processing* provides information about how to schedule custom code to run at the end of page execution.

References

References in Lasso Professional 7 allow multiple variables to point to the same value or object. When the shared value or object is changed, all variables that reference that value or object change. A reference can be created using the [Reference] tag or the @ reference symbol.

An example will serve to illustrate how references can be used in Lasso. The following LDML code creates two variables and sets them to default values, then outputs those values. Each variable is independent. Changing the value of the one variable will not change the value of the other variable.

```
[Variable: 'Alpha'= 1]
[Variable: 'Beta'= 2]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']

➜ <br>Alpha: 1
<br>Beta: 2
```

However, if we instead define the second variable to be a reference to the first variable then the two variables will share a single value. In the following example the variable Alpha is set to 3 and the variable Beta is set to be a reference to the variable Alpha. When output, both variables return 3.

```
[Variable: 'Alpha'= 3]
[Variable: 'Beta'= (Reference: $Alpha)]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']

➜ <br>Alpha: 3
<br>Beta: 3
```

Now that the two variables are linked, changing either variable will effect a change in both. For example, setting Alpha to 4 will also result in a change to Beta.

```
[Variable: 'Alpha'= 4]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']

<br>Alpha: 4
<br>Beta: 4
```

Similarly, setting Beta to 5 will also result in a change to Alpha.

```
[Variable: 'Beta' = 5]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']
```

```
➜ <br>Alpha: 5
  <br>Beta: 5
```

This simple example serves to illustrate the basic principle behind Lasso's references. The remainder of this section will provide demonstrations of how references can be used to reduce the amount of memory that Lasso needs to process complex pages and to increase page processing speed.

It is impossible to have a reference to a reference. Lasso always resolves references back to the original object so if one variable is set as a reference to a second variable, then a third variable is set as reference to the first variable, all three variables end up pointing to the same object. A change to any of the three variables results in the values of all three variables being changed.

References can be detached using the [Null->DetachReference] tag. If a variable is defined as a reference to a value then calling [Null->DetachReference] will set the variable's value to Null and detach it from the referenced object. The variable can then be safely re-assigned without affecting the referenced object.

A reference can also be detached by assigning a new reference to a variable. If a variable is assigned a reference using the @ symbol or the [Reference] tag then it will be linked to the new reference and any previous links will be severed.

Types of References

References can be used to refer to any of the following objects within Lasso.

- **Variables** – A reference to a variable allows the same underlying data to be accessed through two different names. Changing the value of either of the linked variables will result in the values of both varaibles being changed. The data referenced by both variables is only stored once.

  ```
  [Variable: 'Ref_Variable' = @$First_Variable]
  ```

- **Local Variables** – A reference to a page variable can be made within a custom tag. Rather than copying the page variable into a local variable, the page variable can be referenced. This prevents duplicating data and allows any changes made to the local variable to be automatically applied to the page variable.

  ```
  [Local: 'Local_Variable' = @$First_Variable]
  ```

- **Array Elements** – A reference can be made to an array element. This allows one or more array elements to be referenced as variables separate from the array. Any changes made to the variables will be reflected in the array. The [Array->Get] tag is used to identify the array element.

```
[Variable: 'Ref_Variable' = @($Array_Variable->(Get: 1)]
```

- **Map Elements** – A reference can be made to the value of a map element. This allows the values of one or more map elements to be referenced as variables separate from the map. Any changes made to the variables will be reflected in the map. The [Map->Find] tag is used to identify the map element.

  ```
  [Variable: 'Ref_Variable' = @($Map_Variable->(Find: 'Key')]
  ```

- **Tag Parameters** – In a custom tag a reference can be made to a tag parameter rather than copying the parameter into a local variable. This allows a referenced parameter to be modified in place.

  ```
  [Local: 'Local_Variable' = @(Params->(Get: 1))]
  ```

Table 1: Reference Tags and Symbols

Tag / Symbol	Description
@	Creates a reference to an object rather than copying the object. Usually used in a [Variable] tag to assign a variable as a link to an object.
[Reference]	Creates a reference to an object rather than copying the object. Equivalent to the @ symbol.
[Null->DetachReference]	Can be called on a variable of any data type to detach the variable from the linked object. The variable ends up with a value of Null.
[Null->RefCount]	Returns the number of references that refer to a value.

To create a custom tag that works on an array directly:

The following example creates a custom tag that works on the elements of an array in place. Using this principle can greatly speed up the execution speed of LDML code since Lasso does not have to copy each element of the array multiple times.

References are used twice in this tag. The first parameter to the tag (which is expected to be an array) is referenced by a local variable theArray. This prevents the values of the array from being copied into the local variable. Within the [Loop] ...[/Loop] tags. The variable theItem is set to a reference to each element of the tag in turn.

```
[Define_Tag: 'Ex_Square']
  [Local: 'theArray' = @(Params->(Get: 1))]
  [Loop: #theArray->Size]
    [Local: 'theItem' = @(#theArray->(Get: Loop_Count))]
    [#theItem *= #theItem]
  [/Loop]
[/Define_Tag]
```

This tag is used as follows to modify the items in an array in place. Note that the tag does not have a [Return] tag so it does not return any value.

```
[Variable: 'myArray' = (Array: 1, 2, 3)]
[Ex_Square: $myArray]
[Variable: 'myArray']
```

➜ (Array: 1, 4, 9)

Lasso automatically uses references when referencing -Required or -Optional tag parameters and when using the [Iterate]... [/Iterate] tags. It is possible to rewrite the [Ex_Square] tag using these implicit references as follows. This tag will function identically to the previous example.

```
[Define_Tag: 'Ex_Square', -Required='theArray']
  [Iterate: #theArray, (Local: 'theItem')]
    [#theItem *= #theItem]
  [/Itereate]
[/Define_Tag]
```

Global Variables

Lasso maintains a stack of environments as it processes LDML code. The first environment is created when Lasso starts up and includes global, server-wide variables. Each page has its own environment created when it is parsed which includes normal, page-wide variables. Finally, each custom tag and data type has its own environment that includes local variables. At any point, tags can be used to examine and modify values in the environment above the current environment.

The globals tags allow direct access to global variables from any environment. These are the preferred way of setting and retrieving global values. Globals can also be accessed implicitly from the page and local environments following the rules described in the sections below.

Note: Many global variables are used to set preferences for internal Lasso processes such as the email queue, the session handler, and the scheduler. Global variables which start with an underscore should never be modified.

Table 2: Global Tags

Tag	Description
[Global]	If called with a string parameter, retrieves the value of a global variable. If called with a name/value pair sets the value of a global variable.
[Global_Defined]	Accepts a single string parameter. Returns True if the global variable has been defined or False otherwise.
[Globals]	Returns a map of all global variables that are currently defined.

Startup Environment

When code is executed in LassoStartup it is executed in the startup or global environment. Any variables which are set using the [Variable] tag at this level will end up as global variables when pages are executed. Similarly, any tags which are defined at this level will be made available to all pages that are executed on the server.

To set a global variable at startup:

At startup, global variables can be set either using the [Global] tag or using the [Variable] tag. All variables set at this level are implicitly global.

- Use the [Global] tag to set the value of a global variable. The global variable will be available to any page subsequently executed by Lasso. In the following example a variable Administrator_Email is created and set with the value of the administrator's email address.

  ```
  [Global: 'Administrator_Email' = 'administrator@example.com']
  ```

- Use the [Variable] tag to set the values of global variables from code which is executed in the LassoStarup folder. In the following example a variable Administrator_Email is created and set with the value of the administrator's email address.

  ```
  [Variable: 'Administrator_Email' = 'administrator@example.com']
  ```

Page Environment

From the page level the values of global variables can be retrieved using the [Variable] tag provided that no page variable has been defined with the same name or can be retrieved explicitly using the [Global] tag. The [Variable] tag cannot be used to set a global variable. Instead, global variables should be set using the [Global] tag.

To retrieve the value of a global variable:

- Use the [Global] tag. In the following example the global variable Administrator_Email which is set above is retrieved.

```
[Global: 'Administrator_Email']
```

```
➜ administrator@example.com
```

- If the desired variable has not been overridden by a page variable of the same name then use the [Variable] tag to retrieve the value of the global variable. In the following example the global variable Administrator_Email which is set above is retrieved.

```
[Variable: 'Administrator_Email']
```

```
➜ administrator@example.com
```

To set the value of a global variable:

Either of the two following techniques can be used to set the value of a global variable from an LDML format file. The first method is prefered.

- Use the [Global] tag to set the value of a global variable. The global variable will be immediately available on any page executing by Lasso through the [Global] or [Globals] tags.

```
[Global: 'Administrator_Email' = 'new_administrator@example.com']
<br>Global: [Global: 'Administrator_Email']
```

```
➜ <br>Global: new_administrator@example.com
```

- Set the value of a global variable by reference. In the following example, the variable Administrator_Email has not been overriden on the current page. Using the $ and = symbols the global variable can be changed.

```
$Administrator_Email = 'new_administrator@example.com']
<br>Global: [Global: 'Administrator_Email']
```

```
➜ <br>Global: new_administrator@example.com
```

To override the value of a global variable:

Use the [Variable] tag to set a variable of the same name. The global variable will not be modified, but subsequent uses of the [Variable] tag will return the page variable's value. The [Global] tag can still be used to retrieve the value of the global variable.

In the following example the global variable Administrator_Email is overridden by a page variable of the same name. The values of both the page variable and the global variable are displayed.

```
[Variable: 'Administrator_Email' = 'page_administrator@example.com']
<br>Page: [Variable: 'Administrator_Email']
<br>Global: [Global: 'Administrator_Email']
```

```
➜ <br>Page: page_administrator@example.com
  <br>Global: administrator@example.com
```

Local Environment

When a custom tag is executing variables from any environment can be accessed. Global variables should be accessed and set using the [Global] tag, page variables should be accessed and set using the [Variable] tag or $ symbol, and local variables should be accessed and set using the [Local] tag or # symbol.

Bytes Types

All string data in Lasso is processed as double-byte Unicode characters. The [Bytes] type is used to represent strings of single-byte binary data. The [Bytes] type is often referred to as a byte-stream or binary data.

Lasso tags return data in the [Bytes] type in the following situations.

- The [Field] tag returns a byte stream from MySQL BLOB fields.
- When the -Binary encoding type is used on any tag.
- The [Bytes] tag can be used to allocate a new byte stream.
- Other tags that return binary data. See the LDML Reference for a complete list.

Table 3: Byte Stream Tag

Tag	Description
[Bytes]	Allocates a byte stream. Accepts two parameters. The first is the initial size in bytes for the stream. The second is the increment to use to grow the stream when data is stored that goes beyond the current allocation.

Byte streams are similar to strings and support many of the same member tags. In addition, byte streams support a number of member tags that make it easier to deal with binary data. These tags are listed in *Table 4: Byte Stream Member Tags*.

Table 4: Byte Stream Member Tags

Tag	Description
[Bytes->Size]	Returns the number of bytes contained in the stream.
[Bytes->Get]	Returns a single byte from the stream. Requires a parameter which specifies which byte to fetch.
[Bytes->SetSize]	Sets the byte stream to the specified number of bytes.
[Bytes->GetRange]	Gets a range of bytes from the byte stream. Requires a single parameter which is the position to start at. An optional second parameter specifies how many bytes to get.
[Bytes->SetRange]	Sets a range of characters within a byte stream. Requires two parameters: An integer offset into the base stream and the binary data to be inserted. An optional third and fourth parameter specify an offset and length into the binary data to be inserted.
[Bytes->ExportString]	Returns a string represeting the byte stream. Accepts a single parameter which is the character encoing for the export. A parameter of Binary will perform a byte for byte export of the stream.
[Bytes->Export8bits]	Returns the first byte as an integer.
[Bytes->Export16bits]	Returns the first 2 bytes as an integer.
[Bytes->Export32bits]	Returns the first 4 bytes as an integer.
[Bytes->Export64bits]	Returns the first 8 bytes as an integer.
[Bytes->ImportString]	Imports a string parameter. A second parameter specifies the encodingmethod to use for the import. A second parameter of Binary will perform a byte for byte import of the string.
[Bytes->Import8Bits]	Imports the first byte of an integer parameter.
[Bytes->Import16Bits]	Imports the first 2 bytes of an integer parameter.
[Bytes->Import32Bits]	Imports the first 4 bytes of an integer parameter.
[Bytes->Import64Bits]	Imports the first 8 bytes of an integer parameter.
[Bytes->SwapBytes]	Swaps each 2 bytes.

See the LDML Reference for examples of the use of these tags.

Tag Data Type

Tags are represented by Lasso as objects which belong to a data type. Just like arrays or maps, tag objects have member tags which allow them to be manipulated. Tags can be stored in variables or in complex data types such as maps or arrays.

Since calling a tag, e.g. [Action_Params], returns the value that results when the tag is run rather than a reference to the tag, special steps must be taken to get a reference to the tag istelf. Tag objects can be found in four ways.

- **\ Symbol** – The \ symbol can be used to find a tag object. For example, \Field will return a reference to the [Field] tag and \Action_Params will return a reference to the [Action_Params] tag. The following code stores a reference to [Action_Params] in a variable.

  ```
  [Variable: 'myActionParamsTag' = \Action_Params]
  ```

- **Tags Map** – Lasso maintains a global tag map which can be retrieved using the [Tags] tag. An individual tag can be referenced using the [Map->Find] tag. For example, the following code stores a reference to [Action_Params] in a variable.

  ```
  [Variable: 'myActionParamsTag' = Tags->(Find: 'Action_Params')]
  ```

  [Tags] returns a reference to the global variable __tags__ which contains all substitution, container, and process tags defined in Lasso. Custom tags defined on the current page can be found in the page variable __tags__.

- **Data Type Properties** – Each instance of a data type maintains a list of properties for that instance which can be retrieved using the [Null->Properties] tag. These include both instance variables and member tags. For example, the following code stores a reference to the [Array->Get] tag in a variable.

  ```
  [Variable: 'myGetTag' = Array->Properties->Second->(Find: 'Get')]
  ```

  The member tags of the built-in data types can also be found in the global variable __prototypes__.

- **Compound Expressions** – These are discussed in the next section. Compound expressions allow tags to be created on the fly. For example, the following code stores a compound expression that returns the number 5 in a variable.

  ```
  [Variable: 'myTag' = { Return: 5; }]
  ```

Table 3: Tag Data Type Member Tags

Tag	Description
[Tag->Run]	Executes the tag as if it had been called normally. The parameters to this tag are discussed in the table that follows.
[Tag->Eval]	Evaluates a tag or compound expression in the current context. No parameters can be passed to the tag or compound expression.
[Tag->asType]	Executes the tag as a type initializer. Accepts the same parameters as [Tag->Run]
[Tag->asAsync]	Executes the tag in a new thread. Accepts the same parameters as [Tag->Run].
[Tag->Description]	Returns the description of the tag if defined.
[Tag->ParamInfo]	Returns an array of information about the parameters which the tag requires. Each element of the array has members ParamName, ParamType, and IsRequired.
[Tag->ReturnType]	Returns the type of value the tag will return.

The [Tag->Run] tag is most commonly used with built-in LDML tags and with custom tags. This tag accepts the parameters outlined in the following table.

Table 4: [Tag->Run] Parameters

Parameter	Description
-Params	An array of parameters to pass to the tag. Can be omitted if the tag does not require any parameters.
-Owner	Identifies the variable which contains the data type that should be operated on, i.e. the object that would be specified to the left of the -> symbol. Required for member tags.
-Name	Name of the tag. Many built-in LDML tags such as [Math_…], [String_…], [Server_…], etc. behave differently depending on what tag name they are called with. The -Name parameter is required for these tags to operate properly.

To run a tag:

Use the [Tag->Run] tag on a stored reference to the tag which is to be run. The following examples each retrieve a tag from the [Tags] or [Null->Properties] map and then run it using appropriate parameters.

- The [Action_Params] tag can be called as follows. First a reference to the tag is stored in a variable, then [Tag->Run] is called on the stored reference. It

is always best to specify the -Name parameter explicitly since it is required by many built-in tags.

```
[Variable: 'myActionParamsTag' = Tags->(Find: 'Action_Params')]
[$myActionParamsTag->(Run: -Name='Action_Params')]
```

➜ (Array: (Pair: (-Nothing)=()), (Pair: (-OperatorLogical)=(and)), (Pair: (-MaxRecords)=(50)), (Pair: (-SkipRecords)=(0)))

- The [Array->Get] tag can be called by retrieving the tag from the [Array->Properties] map and then calling it using [Tag->Run] with the array that is to be acted upon referenced in the -Owner parameter.

```
[Variable: 'myArray' = (Array: 'Alpha', 'Beta', 'Gamma')]
[Variable: 'myGetTag' = Array->Properties->Second->(Find: 'Get')]
[$myGetTag->(Run: -Params=2, -Owner=$myArray, -Name='Get')]
```

➜ Beta

The previous examples demonstrate how to use the tag member tags to execute tags, but each example is easy enough to write using simple LDML. The following example demonstrates how tag references can be used to create a new type of custom tag that can operate on each element of an array.

To run a tag on each element of an array:

Create a custom tag which accepts an array and a reference to a tag as parameters. The referenced tag will be used on each element of the array in turn. The custom tag [Ex_VisitArray] is defined as follows.

```
[Define_Tag: 'Ex_VisitArray', -Required='myArray', -Required='myTag']
  [Iterate: #myArray, (Local: 'myItem')]
    [#myItem= #myTag->(Run: -Params=(Array: #myItem))]
  [/Iterate]
[/Define_Tag]
```

This tag can now be used to apply a tag to each element of an array. For example, it could be used to replace each element of an array by the value of a variable of the same name. An array and three variables are created and the [Variable] tag is found in the [Tags] map.

```
[Variable: 'theArray' = (Array: 'Alpha', 'Beta', 'Gamma')]

[Variable: 'Alpha' = 100, 'Beta' = 1234, 'Gamma' = 987]

[Ex_VisitArray: $theArray, Tags->(Find: 'Variable')]

[Variable: 'theArray']
```

➜ (Array: 100, 1234, 987)

Combined with the use of compound expressions which are described in the next section this can be a very powerful technique for batch processing of data which is stored in an array.

To get information about a tag:

The [Tag->Description], [Tag->ParamInfo], and [Tag->ReturnType] tags can be used to get information about a tag. For properly defined tags this information can prove invaluable in determing how to use an unknown tag.

The following example shows the definition of a tag [myTag] and then the information that can be retrieved about it.

```
[Define_Tag: 'Ex_Repeat',
    -Required='String', -Type='string',
    -Optional='Repeat', -Type='integer',
    -ReturnType='String',
    -Description='[Ex_Repeat: String, Integer] => String']
  [Return: #String * (Integer: (Local: 'Repeat'))]
[/Define_Tag]

<br>Description: [Output: \Ex_Repeat->Description]
<br>Returns: [Output: \Ex_Repeat->ReturnType]
<br>Params [Iterate: \Ex_Repeat->ParamInfo, (Var: 'param')]
  <br>[Loop_Count]: [Output: $param->ParamName]
  [If: $param->ParamType == 'null'] (Any) [Else] ([Output: $param->ParamType]) [/If]
  [If: $param->IsRequired]Required[/If]
[/Iterate]
```

```
➜ Description: [Ex_Repeat: String, Integer] => String
  Returns: String
  Params:
  1: String (string) Required
  2: Repeat (integer)
```

Compound Expressions

Compound expressions allow for tags to be created within LDML code and executed immediately. Compound expressions can be used to process brief snippets of LDML code inline within another tag's parameters or can be used to create reusable code blocks.

Evaluating Compound Expressions

A compound expression is defined within curly braces { }. The syntax within the curly braces should match that for LassoScripts using semi-colons between each LDML tag. For example, a simple compound expression that

adds 6 to a variable myVariable would be written as follows. The expression can reference page variables.

```
[Variable: 'myExpression' = { $myVariable += 6; }]
```

The compound expression will not run until it is asked to execute using the [Tag->Eval] tag. The expression defined above can be executed as follows.

```
[Variable: 'myVariable' = 5]
[$myExpression->Eval]
[Variable: 'myVariable']
```

➜ 11

A compound expression returns values using the [Return] tag just like a custom tag. A variation of the expression above that simply returns the result of adding 6 to the variable, without modifying the original variable could be written as follows.

```
[Variable: 'myExpression' = { Return: ($myVariable + 6); }]
```

This expression can then be called using the [Tag->Eval] tag and the result of that tag will be the result of the stored calculation.

```
[Variable: 'myVariable' = 5]
[$myExpression->Eval]
```

➜ 11

Alternately, the expression can be defined and called immediately. For example, the following expression checks the value of a variable myTest and returns Yes if it is True or No if it is False. Since the expression is created and called immediately using the [Tag->Eval] tag it cannot be called again.

```
[Variable: 'myTest'= True]
[Output: { If: $myTest; Return: 'Yes'; Else; Return: 'No'; /If; }->Eval]
```

➜ Yes

Running Compound Expressions

The same conventions for custom tags may be used within a compound expression provided it is executed using the [Tag->Run] tag. Compound expressions which are run can access the [Params] array and define local variables.

For example, the following expression accepts a single parameter and returns the value of that parameter multiplied by itself. The expression is formatted similar to a LassoScript using indentation to make the flow of logic clear.

```
[Variable: 'myExpression' = {
  Local: 'myValue' = (Params->(Get: 1));
  Return: #myValue * #myValue;
}]
```

This expression can be used as a tag by calling it with the [Tag->Run] tag with an appropriate parameter. The following example calls the stored tag with a parameter of 5.

```
[Output: $myExpression->(Run: -Params=(Array: 5))]
```

➜ 25

When combined with the [Ex_VisitArray] tag that was defined in the previous section, a compound expression can be used to modify every element of an array in place. In the following example, the compound expression above is used to square every element of an array.

```
[Variable: 'myArray' = (Array: 1, 2, 3)]
[Ex_VisitArray: $myArray, $myExpression]
```

➜ (Array: (1), (4), (9))

Thread Tools

Lasso is a fully multi-threaded environment. Each page is parsed and executed within its own thread, asynchronous custom tags are executed in their own threads, and background processes such as the email queue or schedule watcher are executed in their own threads.

It is important in a multi-threaded environment to synchronize access to resources such as files, global variables, or database records so that two threads do not attempt to modify the same resource at the same time. Communication between threads is discussed in the section that follows.

Consider an LDML format file which maintains a global variable recording how many times the page has been visited. At the top of the page the variable is displayed to the visitor. At the bottom of the page the varaible is incremented by one. Everything will work fine as long as the page is only loaded by one visitor at a time. However, if the page is loaded by two visitors who overlap a situation can develop where the following sequence of events happens.

Thead Example

1 Visitor A loads the format file by loading the URL in their Web browser.

2 Page A starts processing with the value of the global variable, e.g. 100.

3 Visitor B loads the format file by loading the URL in their Web browser.

4 Page B starts processing with the value of the global variable, e.g. 100. This is the same value as what visitor A received.

5 Page A finishes processing and the global variable is set to a new value, e.g. 101. The new value is based on the value of the variable that was fetched at the top of the page.

6 Page B finishes loading and the global variable is set to a new value, also 101. The new value is based on the value of the variable that was fetched at the top of the page and does not take into account the fact that visitor A's page load has already modified the variable.

At the end of the process the global variable has effectively lost track of one visitor. This particular example could be fixed by reading and incrementing the variable at the top of the page, but for other resources it is necessary to restrict access so only one thread or page can have access to the resource at a time.

Table 4: Thread Tools

Type	Description
[Thread_Lock]	A simple per-thread lock which allows sequential access to a shared resource.
[Thread_Semaphore]	A counter that can be incremented or decremented to provide multiple threads access to a shared resource.
[Thread_RWLock]	A lock that allows multiple readers, but only one writer for a shared resource.

Thread Lock

A [Thread_Lock] allows multiple pages or asynchronous tags to use a shared resource sequentially. A [Thread_Lock] is usually created and stored in a global variable so all pages or tags can access it. The [Thread_Lock] has two member tags.

Table 5: [Thread_Lock] Member tags:

Tag	Description
[Thread_Lock->Lock]	Accepts an optional parameter which is the number of milliseconds to wait before timing out. Returns True if the lock was successful or False if the timeout value was reached.
[Thread_Lock->Unlock]	Unlocks a previously established lock. If there is a thread waiting for a lock then it will be allowed to continue.

To control concurrent access to a shared resource:

In the following example, a global variable Counter is used by a Web page to store the number of times that the Web page has been accessed. A [Thread_Lock] is used to ensure that only one page accesses the variable at a time. A timeout of 1000 (one second) is used to ensure that no page ends up waiting too long for access to the variable.

In a page in the LassoStartup folder the following two variables are defined. Counter is the global variable that will store the number of times the page has been loaded. Counter_Lock is the [Thread_Lock] that allows for sequential access to the variable.

```
[Variable: 'Counter' = 0]
[Variable: 'Counter_Lock' = (Thread_Lock)]
```

In the format file that visitors will load the following code attempts to lock Counter_Lock. The Counter is only modified if the attempt to get the lock is successful.

```
[If: $Counter_Lock->(Lock: 1000) == True]
  [$Counter += 1]
  [$Counter_Lock->Unlock]
[/If]
```

The timeout can be used to weigh the importance of having an accurate counter against the length of delay that a site visitor should be subjected to in a busy site. With a simple example like this the timeout will likely never be reached even on a very busy site.

Use of [Thread_Lock] is entirely voluntary and can be used to handle access to any shared resource. It is up to the site designer to create the necessary [Thread_Lock] variables and then use them when accessing shared resources.

Thread Semaphore

A [Thread_Semaphore] is a thread lock which has a counter. The [Thread_Semaphore] is initialized with a maximum number of concurrent accesses that can occur. [Thread_Semaphore] has two member tags which are used to increment or decrement the number of current accesses.

Table 6: [Thread_Semaphore] Member Tags

Tag	Description
[Thread_Semaphore->Increment]	Requires a single parameter which is the amount to increment the semaphore. Does not return until the semaphore can be incremented by that amount. A second, optional parameter specifies the number of milliseconds to wait before timing out.
[Thread_Semaphore->Decrement]	Requires a single parameter which is the amount to decrement the semaphore.

To allow a fixed number of accesses to a shared resource:

A [Thread_Semaphore] can be used with an appropriate maximum value. For example, a page which displays site-wide statistics might take a long time to load so it is desirable to only allow five users to access the page at the same time. A semaphore can be used to block any additional users from seeing the page until one or more of the other users' page loads complete.

The following code would be placed into the LassoStartup folder in order to store the semaphore in a global variable. The semaphore is set to only allow five concurrent users.

```
[Variable: 'Page_Semaphore' = (Thread_Semaphore: 5)]
```

On the page which displays the site wide statistics the semaphore is incremented at the top of the page (with a timeout of 5 seconds) and then decremented at the bottom. If more than five users are already loading the page then the increment at the top will pause until one of the users' page finishes.

```
[If: $Page_Semaphore->(Increment: 1, 5000)]
   … Contents of the Page …
  [$Page_Semaphore->(Decrement: 1)]
[Else]
  <p>Page is busy. Try again later.
[/If]
```

Thread Read/Write Lock

A [Thread_RWLock] is a thread lock which allows an unlimited number of simultaneous reads to occur on a shared resource, but only allows one thread to write to the resource at a time. Write access will not be granted until all reads and writes have completed. Read access will not be granted as long as the write access is currently in use. [Thread_RWLock] has four member tags which are used to establish and release read access and write access.

Table 7: [Thread_RWLock] Member Tags

Tag	Description
[Thread_RWLock->ReadLock]	Establishes a read lock. If a write lock is currently in place then the tag will pause until the write lock is released. Since read locks are not exclusive it will not pause if additional read locks have already been granted.
[Thread_RWLock->ReadUnlock]	Releases a read lock.
[Thread_RWLock->WriteLock]	Establishes a write lock. If one or more read locks or a write lock is in place then the tag will pause until the locks are released.
[Thread_RWLock->WriteUnlock]	Releases a write lock.

To control write access to a resource while allowing multiple reads:

Most resources can be accessed by multiple threads which only need to read from the resource, but require that only one client write to the resource at the same time. In the following example a global variable contains a set of server-wide preferences that can be read by many pages at the same time, but must only be modified by one page at a time.

In a page in the LassoStartup folder the following two variables are defined. Preferences is the global variable that will store server-wide preferences such as the administrator's email address and a count of how many page loads there have been. Preferences_Lock is the [Thread_RWLock] that controls access to the variable.

```
[Variable: 'Preferences' = (Map: 'Email' = 'administrator@example.com' )]
[Variable: 'Preferences_Lock' = (Thread_Lock)]
```

In each format file in the site a read lock is established on the preferences. As many format files as are needed can concurrently read the preferences.

```
[$Preferences_Lock->(ReadLock)]
  ... Contents of the Page ...
[$Preferences_Lock->(ReadUnlock)]
```

In a page which modifies the preferences a write lock needs to be established. The following code first releases the read lock, then establishes a write lock, modifies the global Preferences variable, and finally releases the write lock and re-establishes the read lock for the remainder of the page.

```
[$Preferences_Lock->(ReadUnlock)]
  [$Preferences_Lock->(WriteLock)]
    [$Preferences->(Insert: 'Email' = (Action_Param: 'Email'))]
  [$Preferences_Lock->(WriteUnlock)]
[$Preferences_Lock->(ReadLock)]
```

The [Thread_RWLock] tags do not have a timeout value like the [Thread_Lock] page. In the example above the page which is loaded by the visitor who wants to change the preferences will simply idle until each of the pages which have established a read lock are finished loading.

Thread Communication

The previous section documented methods for sharing data between threads using global variables. Often it is desirable to not just share data, but to push data from thread to thread. This section documents techniques for sending signals and data between threads.

Table 8: Thread Communication

Type	Description
[Thread_Event]	A simple signalling method which allows threads to idle until they receive a signal to continue.
[Thread_Pipe]	Allows variables and data objects to be sent from thread to thread. The foundation of more complex messaging systems.

Thread Events

Thread events are simple signals that are either in an active or inactive state. One or more threads can wait for a signal to occur. A triggering thread can cause one or all of the threads waiting for the signal to continue processing. No data can be passed using thread events.

Table 9: [Thread_Event] Member Tags:

Tag	Description
[Thread_Event->Wait]	Accepts an optional timeout value in milliseconds. If a signal is received before the timeout then True is returned otherwise False is returned. With no timeout value the tag will pause forever.
[Thread_Event->Signal]	Allows one thread which is waiting for this signal to continue.
[Thread_Event->SignalAll]	Allows all threads which are waiting for this signal to continue.

To create an asynchronous tag that waits for a signal:

Create a [Thread_Event] signal in a page variable. Within a custom asynchronous tag, wait until the signal is triggered before continuuing.

In the following example a custom tag waits for one second for the page to reach the end. It then logs whether the page completed in one second or not to the console using the [Log_Warning] tag. At the top of the page the custom tag and the signal are defined. The custom tag is called to start the one second timer. At the bottom of the page the signal is triggered.

```
[Variable: 'mySignal' = (Thread_Event)]
[Define_Tag: 'OneSecond']
  [If: $mySignal->(Wait: 1000) == True]
    [Log_Warning: 'The page took less than 1 second to load.']
  [Else]
    [Log_Warning: 'The page took more than 1 second to load.']
  [/If]
[/Define_Tag]
[OneSecond]

 ... Page Contents ...

[$mySignal->Signal]
```

Each time the page loads one of the messages will be logged to the console depending on how long the page took to process.

Thread Pipes

Thread pipes allow data to be passed from thread to thread. The [Thread_Pipe] type has two member tags.

Table 10: [Thread_Pipe] Member Tags:

Tag	Description
[Thread_Pipe->Set]	Accepts a value which will be placed into the pipe. A subsequent (or waiting) call to [Thread_Pipe->Get] will retrieve the value.
[Thread_Pipe->Get]	Accepts an optional parameter which is a timeout value in milliseconds. If an object is waiting in the pipe or is placed in the pipe before the timeout value then it is returned. Otherwise null is returned when the timeout value is reached. If the timeout value is omitted then the tag will wait forever for an object to arrive.

To create an asynchronous tag that will process messages:

Create a [Thread_Pipe] in a global variable and an asynchronous tag in the LassoStartup folder. The asynchronous tag will idle until an event is placed

into the pipe. For this example, the tag will log each received event to the console, but more complex processing is possible.

In a page in the LassoStartup folder the [Thread_Pipe] and custom tag [Ex_Watcher] are defined. The custom tag has a [While: True] loop that ensures that it loops forever since the condition will always be true. The [Thread_Pipe->Get] tag has a timeout value of 10000 (10 seconds) so it can send a Still Waiting... message to the console. If a message of Abort is received then the tag aborts without doing any further processing.

```
[Variable: 'myPipe'= (Thread_Pipe)]
[Define_Tag: 'Ex_Watcher']
  [While: True]
    [Local: 'Message' = $myPipe->(Get: 10000)]
    [Select: #Message]
      [Case: 'Abort']
        [Return]
      [Case: Null]
        [Log_Warning: 'Message: Still Waiting…']
      [Case]
        [Log_Warning: 'Message: ' + #Message]
    [/Select]
  [/While]
[/Define_Tag]
[Ex_Watcher]
```

In a format file a message can be sent to the watcher by placing it into the pipe using the [Thread_Pipe->Set] tag. For example, the following code places a message saying I Got It! into the pipe. The message will appear in the console immediately, but no results will be returned to the page.

```
[$myPipe->(Set: 'I Got It!')]
```

Network Communication

Network communication in Lasso are provided by the [Net] type and its member tags. These tags allow for direct communiction between Lasso and remote servers using low-level communication standards. These tags are the foundation for the implementation of specific protocols such as HTTP, RPC, or SMTP communication.

Note: Using the [Net] type requires an understanding of Internet communication standards. The examples in this chapter are purely for demonstration purposes of the [Net] tags.

The [Net] type supports the following features:

- **TCP (Transmission Control Protocol)** – Connection oriented communication with remote servers. TCP allows communication with full duplex capabilities and guaranteed delivery of data.
- **UDP (User Datagram Protocol) –** Connectionless communication with remote servers. UDP is a lightweight format that allows communication without guaranteed delivery of data.
- **Listeners** – Lasso allows sockets to be opened to listen for either TCP or UDP traffic. Lasso can act as either the source or the recipient of TCP or UDP communication.
- **Non-Blocking** – Connections can be non-blocking so data is sent and received without synchronization with the remote host.
- **Timeouts** – Lasso has an efficient set of timeout controls that allow different timeout periods to be used when establishing connections and when participating in communication.

Note: The [TCP_...] tags from prior versions of Lasso have been deprecated. Solutions which rely on the [TCP_...] tags should be rewritten to make use of the new functionality afforded by the [Net] type.

The [Net] tag and the constants that are returned from some network operations are detailed in *Table 11: [Net] Tags*. The member tags of the [Net] type are split into three categories. The tags which are used to control connections for either TCP or UDP communication are listed in *Table 12: [Net] Type Members Tags*. The tags specific to TCP communication are listed in *Table 13: [Net] TCP Member Tags* and the tags specific to UDB communication are listed in *Table 14: [Net] UDP Member Tags*.

The discussion that follows is split into three sections: *TCP Communication*, *TCP Listening*, and *UDP Communication*.

Table 11: [Net] Tags

Tag	Description
[Net]	Create a new network data type. Requires no parameter. All interaction with the network data type is performed through the member tags detailed below.
[Net_ConnectOK]	Returned by [Net->Connect] if the connection was established.
[Net_Connect InProgress]	Returned by [Net->Connection] if another connection is in progress.
[Net_TypeTCP]	Passed to [Net->SetType] to establish TCP communication.
[Net_TypeSSL]	Passed to [Net->SetType] to establish SSL over TCP communication.
[Net_TypeUDP]	Passed to [Net->SetType] to establish UDP connectionless communication.
[Net_WaitRead]	Passed into and/or returned from [Net->Wait] to signal that bytes are available for reading from a connection.
[Net_WaitWrite]	Passed into and/or returned from [Net->Wait] to signal that bytes can be written into a connection.
[Net_WaitTimeout]	Returned from [Net->Wait] to signal that a timeout occurred.

Note: All of the [Net_...] tags represent values that are either passed into [Net] type member tags or returned from them. None of these tags are used on their own.

Table 12: [Net] Type Member Tags

Tag	Description
[Net->Bind]	Binds to a specific port on the local machine. Requires a single parameter which is the port on which to bind. Required for establishing a listener or reading bytes from a connectionless protocol (like UDP).
[Net->Close]	Closes an open or bound connection. Every connection which is opened should be explicilty closed when its use is completed.
[Net->LocalAddress]	Returns the address of the local host.
[Net->RemoteAddress]	Returns the address of the remote host.
[Net->SetBlocking]	Specifies whether connects, sends, and receives should block until the operation completes. Requires a single boolean parameter. The default is True to require blocking.
[Net->SetType]	Specifies whether the connection should use SSL or UDP. Requires a single parameter either [Net_TypeSSL] or [Net_TypeUDP]. Defaults to TCP communication [Net_TypeTCP].
[Net->Wait]	Waits for a specified number of seconds for the connection to enter a state. Requires one parameter which is the number of seconds to wait before timing out. A negative value will cause the tag to wait forever. An optional second parameter can be either [Net_WaitRead] or [Net_WaitWrite] specifying the state to wait for, otherwise either state will trigger a return. The tag returns the current state of the connection [Net_WaitRead] or [Net_WaitWrite] or [Net_WaitTimeout] if the timeout value was reached.

SSL Communication

SSL connections and listeners are established in exactly the same fashion as TCP connections and listeners. All of the same member tags are used except that [Net->(SetType: Net_TypeSSL)] must be called to instruct Lasso that SSL-based communication is required.

Notes are provided throughout the examples for TCP connections and listeners which provide details of how to establish SSL communication.

TCP Communication

TCP connections are some of the most common on the Internet. They are used for communication with Web servers, email servers, FTP servers, and for protocols like SSH and Telnet.

Table 13: [Net] TCP Member Tags

Tag	Description
[Net->Accept]	Accepts a single connection and returns a new [Net] instance for the connection.
[Net->Connect]	Connects to a remote host. Requires two parameters. The first is the DNS host name or IP address of the remote host. The second is the port on which to connect. Returns [Net_ConnectOK] if the connection was established or [Net_ConnectInProgress] if a connection attempt is already in progress.
[Net->Listen]	Switches the connection to an incoming, listening socket.
[Net->Read]	Reads bytes from the connection. Requires a single parameter which is the maximum number of bytes to be read. Returns the bytes read from the connection.
[Net->Write]	Writes bytes into the connection. Requires a single parameter which is the string to be written into the connection. Optional second and third parameters specify an offset and count of characters from the string to be written into the connection. Returns the number of bytes written.

To use a blocking TCP connection:

By default a TCP connection uses blocking to ensure that each communication completes before the next begins. This mode works best for command/response protocols in which commands are issued to the remote host and then the response to those commands is received back. Many standard Internet protocols like HTTP, SMTP, and FTP rely on this mechanism.

The basic outline of a TCP communication session is as follows.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with a remote server.

```
[Var: 'myConnection' = (Net)]
```

Note: If SSL communication is desired for the TCP connection then [$myConnection->(SetType: Net_TypeSSL)] should be called immediately after creating the [Net] object.

2 A connection to a remote server is established. The connection requires the DNS host name or IP address of the remote server and the port on which to connect.

```
[$myConnection->(Open: 'localhost', 80)]
```

3 At this point the remote server might send a welcome message. HTTP servers (port 80) don't send any message. SMTP servers (port 25) send a message like the following. The parameter to [Net->Read] is the maximum number of characters to fetch. There are only about 32 characters in the connection buffer so that is all that is returned.

```
[$myConnection->(Read: 1024)]
```

```
➜ 220 localhost Mail Ready for action
```

4 A message can be sent through the channel to the remote server using the [Net->Write] tag. For example, sending GET / (followed by \r\n) to an HTTP server will get the HTML of the home page of the default site.

```
[$myConnection->(Write: 'GET /\r\n')]
```

5 The return value from the Web server can be read using [Net->Read]. Since this is a blocking connection the [Net->Read] tag will wait until the response from the remote server is complete before returning. The parameter to [Net->Read] is the maximum number of characters to fetch and should be larger than the expected result. In this case we will fetch the first 32 kilobytes of the Web page.

```
[$myConnection->(Read: 32768)]
```

```
➜ <html>\r<head>\r\t<title>Default Page</title>\r</head>\r<body>…</body>\r</html>
```

6 The connection should be closed once communication is complete.

```
[$myConnection->Close]
```

To use a non-blocking TCP connection:

When using a non-blocking TCP connection each [Net->Read] tag will return immediately with whatever data is currently available to be read. This means that if no data has been received from the remote server [Net->Read] will return with no bytes. Rather than repeatedly calling [Net->Read], the [Net->Wait] tag can be used to wait until there are bytes available to be read.

The basic outline of a non-blocking TCP session is as follows.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with a remote server.

```
[Var: 'myConnection' = (Net)]
```

Note: If SSL communication is desired for the TCP connection then [$myConnection->(SetType: Net_TypeSSL)] should be called immediately after creating the [Net] object.

2 A connection to a remote server is established. The connection requires the DNS host name or IP address of the remote server and the port which is to be connected to.

```
[$myConnection->(Open: 'localhost', 80)]
```

3 The connection is switched over to non-blocking mode using the [Net->SetBlocking] tag.

```
[$myConnection->(SetBlocking: False)]
```

4 A message can be sent through the channel to the remote server using the [Net->Write] tag. For example, sending GET / (followed by \r\n) to an HTTP server will get the HTML of the home page of the default site.

```
[$myConnection->(Write: 'GET /\r\n')]
```

5 A [Net->Wait] tag is used to wait until there is data which can be read through the connection. The [Net->Wait] tag takes two parameters. The first is the condition which is being waited for, in this case [Net_WaitRead], and the second is the number of seconds to wait. The tag below will wait for 60 seconds for data to be available.

```
[$myConnection->(Wait: Net_WaitRead, 60)]
```

This tag should be incorporated into a conditional statement so its return value can be checked. The return value from [Net->Wait] will be either [Net_WaitRead] if a read is not possible or [Net_WaitTimeout] if the 60 seconds timeout was reached. The following code will perform a read from the connection only if data is available.

```
[If: ($myConnection->(Wait: Net_WaitRead, 60) == Net_WaitRead)]
  [$myConnection->(Read: 32768)]
[Else]
  Timeout!
[/If]
```

➜ <html>\r<head>\r\t<title>Default Page</title>\r</head>\r<body>…</body>\r</html>

6 The connection should be closed once communication is complete.

```
[$myConnection->Close]
```

TCP Listening

The [Net] type can be used to listen for connections coming in from remote clients. This allows Lasso to act as the server for different protocols. In

theory, with this functionality Lasso itself could be used as an HTTP server or SMTP server.

The basic outline of a TCP listening session is as follows.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with a remote server.

```
[Var: 'myListener' = (Net)]
```

Note: If SSL communication is desired for the TCP listener then [$myListener->(SetType: Net_TypeSSL)] should be called immediately after creating the [Net] object.

2 The connection must be switched into listening mode and bound to a port on the local machine. This is the port that remote clients will access in order to communicate with the new service. In this example port 8000 is used.

```
[$myListener->(Bind: 8000)]
[$myListener->(Listen)]
```

3 Since this is a listener no further action is required until a remote client attempts a connection. The [Net->Accept] tag is used to wait for and accept a connection when one comes in. The result of the [Net->Accept] tag is a new [Net] object specific for the remote client that has connected. The listener is then free to call [Net->Accept] again and wait for the next connection.

```
[Var: 'myConnection' = $myListener->(Accept)]
```

4 Now, using the connection that has been established with the remote host, the particular needs of the protocol that is being implemented must be met. For this example, the connecction will wait for a command from the remote client and then return a Web page in response.

```
[Var: 'myCommand' = $myConnection->(Read: 1024)]

[If: ($myCommand >> 'GET")]
  [$myConnection->(Write: '<html> … </html>')]
[Else]
  [$myConnection->(Write: 'Error: Unrecognized Command')]
[/If]
```

➜ <html>\r<head>\r\t<title>Default Page</title>\r</head>\r<body>…</body>\r</html>

6 The connection should be closed once communication is complete and if no further connections will be processed by the listener it should be closed as well.

```
[$myConnection->Close]
[$myListener->Close]
```

A listener can be blocking or non-blocking and can use the [Net->Wait] command to implement timeouts. The [Net] type can be used to create a listener that only accepts one connection at a time or to create a listener that spawns an asynchrnous tag for each incoming connection so many connections can be handled simultaneously.

UDP Connections

UDP connections are generally used for simpler protocols on the Internet. UDP is considered connectionless. Rather than establishing a connection and then sending data, data will simply be sent to the remote host and a response listened for. UDP is an excellent method for one way communication, such as a status logging service, or for single command/response communication. UDP connections make use of the general [Net->Bind] and [Net->Wait] and [Net->Close] tags as well as two UDP specific tags, [Net->ReadFrom] and [Net->WriteTo].

Table 14: [Net] UDP Member Tags

Tag	Description
[Net->ReadFrom]	Reads whatever data is available from a UDP connection. Requires one parameter which is the maximum number of bytes to read. Returns a pair where the first part is the data and the second part is the name of the host that sent the data.
[Net->WriteTo]	Sends data to a specified host and port. Requires three parameters. The DNS host name or IP address of the remote host, the port to connect to, and the string data to be written. Optional additional parameters allow an offset and count into the string data to be specified. Returns the number of bytes written.

To send a message using UDP:

A message can be sent to a remote host with UDP using only the [Net->WriteTo] tag. This tag includes the connection information and message to send all in one call. This example implements a fictional TIME command that is sent to port 8000 on a remote machine. The remote machine will then send back the current time on port 8000.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with any remote UDP servers.

```
[Var: 'myConnection'= (Net)]
```

2 The [Net] object must be switched to UDP mode using the [Net->SetType] tag.

```
[$myConnection->(SetType: Net_TypeUDP)]
```

3 A message is sent to the remote server. The [Net->WriteTo] tag requires the DNS host name or IP address of the remote server and the port which is to be connected to as well as the message which is to be sent.

```
[$myConnection->(WriteTo: 'time.example.com', 8000, 'TIME')]
```

4 The connection should be closed once all UDP communication have completed. However, this same connection can be used to communicate with many different servers.

```
[$myConnection->Close]
```

Once a UDP message has been sent a listener must be established to wait for a reply. Since no connection is established there is no way to simply hold the channel open so the remote host can reply immediately.

To listen for a message using UDP:

Listening for a UDP message involves opening a port and then waiting for a message using the [Net->ReadFrom] tag. Messages can come in from any machine on the Internet. The incoming data is returned as the first part of the result from [Net->ReadFrom] and the address of the remote host is sent as the second part of the result.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with any remote UDP servers.

```
[Var: 'myListener'= (Net)]
```

2 The [Net] object must be switched to UDP mode using the [Net->SetType] tag.

```
[$myListener->(SetType: Net_TypeUDP)]
```

3 The [Net] object is bound to a local port using the [Net->Bind] tag. The local port is the port that other machines will send messages on. For this example, the listener is bound to port 8000.

```
[$myListener->(SetType: Net_TypeUDP)]
```

4 Now the listener must be wait for a message to come in from a remote server. The [Net->ReadFrom] tag will wait until a message comes in and then return a pair containing the data that has been received and the address of the host that sent the data. The parameter is the maximum number of bytes to read.

```
[Var: 'myMessage' = $myListener->(ReadFrom: 32768)]
[Var: 'myData' = $myMessage->First]
[Var: 'myHost' = $myMessage->Second]
```

The current time can now be output by displaying the data that was sent from the remote host.

```
The current time is: [Var: 'myData'].
```

5 The connection should be closed once all UDP communication have completed. However, this same connection can be used to communicate with many different servers.

```
[$myListener->Close]
```

```
 -Op='bw', 'store_key'='log';
Records;
 Var: 'key' = (integer: (field: 'data'));
 Var: 'file' = 0, 'database' = 0, 'console' = 0;
```

Post Processing

Lasso will perform all tags or compound expressions which are stored in the _at_end variable immediately after the rest of the format file has completed executing, but before the formatted page is sent to the user. This allows post processing to be performed on the page.

Note: Several internal functions may be scheduled for post processing using the _at_end variable. The default contents of the _at_end variable should never be modified.

To post process a format file:

There are two methods to add code to the _at_end variable:

- Custom tags can be added to the _at_end variable by reference. They will be called without any parameters. IN the following example a tag [Ex_PostProcess] is defined and then inserted into _at_end using a reference \Ex_PostProcess.

```
<?LassoScript
  Define_Tag: 'Ex_PostProcess';
    ...
  /Define_Tag;
  $_at_end->(Insert: \Ex_PostProcess);
?>
```

- Compound expressions can be inserted into the _at_end variable directly.

```
<?LassoScript
  $_at_end->(Insert: { ... });
?>
```

6

Chapter 6
Lasso C/C++ API 7

This chapter documents Lasso C/C++ API 7 (LCAPI 7), which can be used to develop new Lasso data source connectors, data types, and LDML tags.

- *Overview* introduces the API and describes the types of extensions that can be built.
- *What's Changed* describes what is new and what has changed between LCAPI versions 6 and 7.
- *Requirements* includes platform-specific development environment details.
- *Getting Started* is a quick-start guide to building the samples included with the Extending Lasso 7 Guide.
- *Debugging* includes platform-specific information about how to debug your projects.
- *Substitution Tag Operation* introduces the theory of operation behind creating substitution tags using LCAPI.
- *Substitution Tag Tutorial* describes authoring and building a substitution tag.
- *Data Source Connector Operation* introduces the theory of operation behind creating data source connectors using LCAPI.
- *Data Source Connector Tutorial* walks through authoring and building a Lasso data source connector.
- *Data Type Operation* introduces the theory of operation behind creating custom data types using LCAPI.
- *Data Type Tutorial* walks through authoring and building custom data types.

- *LCAPI Function Reference* includes details of every function used in LCAPI.
- *LCAPI Data Type Reference* includes details of every data type used in LCAPI.
- *Frequently Asked Questions* includes troubleshooting information and common questions.

Overview

LCAPI lets you write C or C++ code to add new LDML substitution tags, data types, and data source connectors to Lasso Professional 7. LCAPI is similar to LJAPI, but is optimized for C and C++ developers.

It is generally recommended that data source connectors be developed using LCAPI because they will offer superior performance and easier installation over connectors built using LJAPI. Writing tags in LCAPI offers advantages over LJAPI and custom LDML tags in speed and system performance. However, tags must be compiled separately for Windows 2000/XP and Mac OS X in order to support each platform. Alternately, custom tags written in LDML instantly support each platform. See *Chapter 3: Custom Tags and Types* for more information on writing custom tags in LDML.

This chapter provides a walk-through for building an example substitution tag, data source connector, and data type in LCAPI. Source code for the Lasso MySQL module as well as the code for the substitution tag, data type, and data source connector examples are included in the Lasso Professional 7/Documentation/4-ExtendingLasso/LCAPI folder on the hard drive.

What's Changed

LCAPI 6 expands on LCAPI 5 by providing functions that allow tag parameters to be retrieved while preserving the parameter's data type, and it allows page variables of any type to be set and retrieved. It also provides functions to aid in logging critical errors and debugging messages, and functions for querying Lasso security for access permissions.

LCAPI 6 includes new facilities for creating new data types, container tags, and asynchronous tags that run in their own thread. Data types created via LCAPI 6 may utilize tag members, data members, and callbacks. LCAPI 6 also provides the ability to manipulate native data types, LDML data types, or other LCAPI data types by accessing their data members or running their member tags. For instance, using LCAPI 6 functions, developers can

build custom arrays and maps that can be used in LDML scripts. LCAPI 6 also allows any type of data to be returned from an LCAPI tag, including binary data and complex data types such as maps.

Requirements

In order to write your own LDML substitution tags or data source connectors in C or C++, you need the following:

Windows

- Microsoft Windows 2000 or Microsoft Windows XP Professional.
- Microsoft Visual C++ 7.
- Lasso Professional 7 for Windows 2000/XP.

Mac OS X

- Mac OS X 10.2 or 10.3 with GNU C++ compiler and linker (Dev Tools) installed.
- Lasso Professional 7 for Mac OS X.

Getting Started

This section provides a walk-through for building sample LCAPI tag modules in Windows 2000/XP and Mac OS X.

To build a sample LCAPI tag module in Windows 2000/XP:

1 Locate the following folder in the hard drive.

C:\Program Files\Blue World Communications\Lasso Professional 7\Documentation\4-ExtendingLasso\Tags\MathFuncsTags

2 In the MathFuncsTags folder, double-click the MathFuncsCAPI.dsp project file (you need Microsoft Visual C++ 7 in order to open it).

3 Choose *Build > Rebuild All* to compile and make the MathFuncsCAPI.DLL.

4 After building, Debug and Release folders will have been created inside your MathFuncsCAPI project folder.

5 Open the MathFuncsTags/Debug folder and drag MathFuncsCAPI.DLL into the Lasso Professional 7/LassoModules folder on the hard drive.

6 Stop and then restart Lasso Service.

7 New tags [Example_Math_Abs], [Example_Math_Sin] and [Example_Math_Sqrt] are now part of the LDML language.

8 Drag the sample Lasso format file called MathFuncsCAPI.lasso into your Web server root.

9 In a Web browser, view http://localhost/MathFuncsCAPI.lasso to see the new LDML tags in action. The source code for MathFuncsCAPI.cpp can be read in the MathFuncsCAPI.lasso page.

To build a sample LCAPI tag module in Mac OS X:

1 Open a Terminal window.

2 Change the current folder to the Lasso Professional 7/Documentation folder using the following command:

```
cd /Library/Lasso Professional 7/Documentation/4-ExtendingLasso/LCAPI/Tags/
MathFuncsTags
```

3 Build the sample project using the provided makefile. You must be logged in as the root user to run this command.

```
make
```

4 After building, a Mac OS X dynamic library file named libMathFuncs.DYLIB will be in the current folder. This is the LCAPI module you'll install into the LassoModules folder.

5 Copy the newly-created module to the Lasso modules folder using the following command:

```
cp libMathFuncs.DYLIB /Applications/Lasso Professional 7/LassoModules
```

6 Quit Lasso Service if it's running, so that the next time it starts up, it will load the new module you just built (you'll need to know a root password to use sudo).

```
cd /Applications/Lasso Professional 7/Tools/
sudo ./stopLassoService.command
```

7 Start Lasso Service so it will load the new module.

```
cd /Applications/Lasso Professional 7/Tools/
sudo ./startLassoService.command
```

New tags [Example_Math_Abs], [Example_Math_Sin] and [Example_Math_Sqrt] are now part of the LDML language.

8 Copy the sample Lasso format file called MathFuncsCAPI.lasso from your Lasso Professional 7/Documentation/4-ExtendingLasso/LCAPI folder into your Web server document root.

9 Use a Web browser to view http://localhost/MathFuncsCAPI.lasso to see the new LDML tags in action. The source code for MathFuncsCAPI.cpp can be read in the MathFuncsCAPI.lasso page.

Debugging

You can set breakpoints in your LCAPI DLLs or DYLIBs and perform source-level debugging for your own code. In order to set this up, add path information to your project so it knows where to load executables from. For this section, we will use the provided substitution tag project as the example.

To debug in Windows 2000/XP:

1. In Microsoft Visual C++ 7, choose the *Project > Settings* menu item.
2. Make sure the Win32 Debug pull-down menu is selected, if available.
3. Click the Debug tab.
4. In the Executable for debug session box, enter the full path to your currently-installed Lasso Professional application, typically as follows:

   C:\Program Files\Blue World Communications\Lasso Professional 7\ LassoService.exe.

5. In the Working folder box, enter the same path as step 4, but remove LassoService.exe from the end of the path.

   C:\Program Files\Blue World Communications\Lasso Professional 7\.

6. Now click the Link tab.
7. In the Output file name box, enter a full path to your module name in the LassoModules folder.

   C:\Program Files\Blue World Communications\Lasso Professional 7\LassoModules\ MathFuncsCAPI.DLL.

8. Set a breakpoint in your source code; one good place to break is in the registerLassoModule() function, which will cause a break during Lasso Professional startup.
9. Stop Lasso Service if it's running so that the DLL may launch Lasso Professional from the Visual C++ project.
10. Run the DLL. A console window should now appear with the startup information as Lasso Professional begins to execute from your project.

To debug in Mac OS X:

1. From a Terminal window, change folder into the example LCAPI source code folder by entering the following:

   ```
   cd /Applications/Lasso\ Professional\ 7/Documentation/4-ExtendingLasso/LCAPI/Tags/MathFuncsTags
   ```

2. Build with debug options turned on by entering the following:

   ```
   make "DEBUG += -g3 -O0"
   ```

Note: The last two characters of the command are a letter O followed by a zero.

3 Copy the built DYLIB into the LassoModules folder by entering the following:

```
cp libMathFuncs.dylib /Applications/Lasso\ Professional\ 7/LassoModules/
```

4 Change folder into the Lasso Professional 7/Tools folder:

```
cd /Applications/Lasso\ Professional\ 7/Tools/
```

5 Quit Lasso Service if it's running, so that the next time it starts up, it will load the new module you just built (you'll need to know a root password to use sudo).

```
./stopLassoService.command
```

6 Start the Lasso Service back up, so it will load the new module.

```
./startLassoService.command
```

7 Find out the process ID number of Lasso Service so you can attach to it later with GNU Debugger. Make a note of the process id for ./LassoService.

```
ps aux|grep LassoService
```

8 Start the GNU Debugger as a root user. You must be root in order to attach to the running Lasso Service process.

```
sudo gdb
```

9 From within GNU Debugger's command line, attach to the Lasso Service process ID by entering the following:

```
attach <type the process id from step 7 here>
```

10 Instruct GNU Debugger to break whenever the function tagMathAbsFunc() is called by entering the following:

```
break tagMathAbsFunc
```

11 Use a Web browser to access the sample http://localhost/MathFuncsCAPI.lasso. An example Lasso format file is provided in the LCAPI folder; you must first copy it into your Web server's Documents folder, which is typically Library/WebServer/Documents.

12 GNU Debugger breaks at the first line in tagMathAbsFunc() as soon as Lasso Service executes that tag in the format file

13 Type help in GNU Debugger for more information about using the GNU Debugger, or search for gdb tutorial on the Web for more in-depth tutorials.

Substitution Tag Operation

When Lasso Professional first starts up, it looks for module files (Windows DLLs or Mac OS X DYLIBS) in its LassoModules folder. As it encounters each module, it executes that module's registerLassoModule() function once and only once. LCAPI developers must write code to register each of the new custom tag (or data source) function entry points in this registerLassoModule() function. The following function is required in every LCAPI module. It gets called once when Lasso Professional starts up.

```
void registerLassoModule()
{
lasso_registerTagModule( "CAPITester", "test_tag", myTagFunc, flag_
typeSubstitutionTag, "simple test LCAPI tag" );
}
```

The following example registers a C function called myTagFunc to execute whenever the LDML [test_tag] is encountered inside an xxx.lasso format file. The first parameter CAPITester is just an arbitrary name that lets you group similar tags together into groups of functions.

Once the tag function is registered, Lasso will call it at appropriate times while parsing and executing Lasso format files. The custom tag functions will not be called if none of the custom tags are encountered while executing a format file. When Lasso Professional 7 encounters one of your custom tags, it will be called with two parameters: an opaque data structure called a "token", and an integer "action". LCAPI provides many function calls which you can use to get information about the environment, variables, parameters, etc., when provided with a token.

The passed-in token can also be used to return error codes and text from your custom tag function. LCAPI provides several functions for setting error numbers, error text, and returning string data. Any string data you return from your function will be displayed in place of the raw LDML tag in the format file, behaving just like any other built-in LDML tag.

To build a basic custom tag function:

Enter the following code:

```
osError myTagFunc( lasso_request_t token, tag_action_t action )
{
  if( action == tagFormatSubstitution )
  lasso_outputTagData( token, "Hello, World" );
}
```

Below is the LDML needed in a Lasso format file in order to get the custom tag to execute:

```
<html>
<body>
 Here's the custom tag:
 [test_tag] <!-- This should display "Hello, World" when this page executes -->
</body>
</html>
```

This will produce the following:

➜ Hello, World

Substitution Tag Tutorial

This section provides a walk-through of building an example tag to show how LCAPI features are used. This code will be most similar to the sample MathFuncsCAPI project, so in order to build this code, copy the MathFuncsCAPI project folder and edit the project files inside it.

The tag will simply display its parameters, and will look like the example below when called from an LDML format file. Notice the required convention of placing a dash in front of all named parameters, which is used to make the parameters easier to spot in the LDML code and prevent ambiguities in the LDML parser.

Example of sample tag LDML syntax:

```
[sample_tag: 'some text here', -option1='named param', -option2=12.5]
```

Notice the tag takes one unnamed parameter, one string parameter named -option1, and a numeric parameter named -option2. In general, LDML does not care about the order in which you pass parameters, so plan to make this tag as flexible as possible by not assuming anything about the order of parameters. The following variations should work exactly the same:

Example of sample tag with different ordered parameters:

```
[sample_tag: -option2=12.5, 'some text here', -option1='named param']

[sample_tag: -option2=12.5, -option1='named param', 'some text here']
```

Substitution Tag Module Code

Shown below is the code for the substitution tag module. Line numbers are provided to the left of each line of code, and are referenced in the *Substitution Tag Module Walk Through* section.

Note: The line numbers shown refer to the line numbers of the code in the actual file being created, not as shown in this page. Some single lines of code may carry into two or more lines as shown on this page.

Substitution Tag Module Code

```
1  void registerLassoModule()
2  {
3  lasso_registerTagModule( "myModule", "sample_tag", myTagFunc, flag_
   typeSubstitutionTag, "sample test" );
4  }
5  osError myTagFunc( lasso_request_t token, tag_action_t action )
6  {
7    if( action == tagFormatSubstitution ) {
8    auto_lasso_value_t   v;
9    if( lasso_findTagParam( token, "-option1", &v ) == osErrNoErr ) {
10     lasso_outputTagData( token, "The value of -option1 is " );
11     lasso_outputTagData( token, v.data );
12   }
13   if( lasso_findTagParam( token, "-option2", &v ) == osErrNoErr ) {
14     double tempValue;
15     char  tempText[32];
16     sscanf( v.data, "%lf", &tempValue );
17     sprintf( tempText, "%.15lg", tempValue );
18     lasso_outputTagData( token, "The value of -option2 is " );
19     lasso_outputTagData( token, tempText );
20   }
21   int  count;
22   lasso_getTagParamCount( token, &count );
23   for (int i = 0; i < count; ++i) {
24     lasso_getTagParam( token, i, &v );
25     if( strcmp( v.data, "" ) == 0 ) {
26       lasso_outputTagData( token, "The value of unnamed param is " );
27       lasso_outputTagData( token, v.name );
28     }
29   }
30 }
31 return osErrNoErr;
32 }
```

Substitution Tag Module Walk Through

This section provides a step-by-step walk through for building the substitution tag module.

To build a sample LCAPI tag module:

1 First, register the new tag in the required registerLassoModule() export function, as shown in lines 1-4.

```
1  void registerLassoModule()
2  {
3  lasso_registerTagModule( "myModule", "sample_tag", myTagFunc, flag_
   typeSubstitutionTag, "sample test" );
4  }
```

2 Implement myTagFunc, which gets called when [sample_tag] is encountered. All tag functions have this prototype. When the tag function is called, it's passed an opaque "token" data structure and an integer "action" telling it what it should do.

```
5  osError myTagFunc( lasso_request_t token, tag_action_t action )
```

3 Check to see if tagFormatSubstitution is being called, which means the process should be executed now, and all parameters are available for perusal. The auto_lasso_value_t variable named v will be our temporary variable for holding parameter values.

```
6  if( action == tagFormatSubstitution ) {
7    auto_lasso_value_tv;
8    if( lasso_findTagParam( token, "-option1", &v ) == osErrNoErr ) {
9      lasso_outputTagData( token, "The value of -option1 is " );
10     lasso_outputTagData( token, v.data );
11   }
12 }
```

4 Call lasso_FindTagParam() in order to get the value of the other named parameter -option2 and store its value into variable v. Only display it if the parameter was actually found (no error while finding the named parameter). Declare a temporary floating-point (double) value to hold the number passed in and then declare a temporary string to hold the converted number for display.

```
13 if( lasso_findTagParam( token, "-option2", &v ) == osErrNoErr ) {
14   double tempValue;
15   char  tempText[32];
16   sscanf( v.data, "%lf", &tempValue );
17   sprintf( tempText, "%.15lg", tempValue );
18   lasso_outputTagData( token, "The value of -option2 is " );
19   lasso_outputTagData( token, tempText );
20 }
```

All parameters passed to your custom tag function are strings. If you are expecting a number, then it is up to you to convert it from a string to a floating point. The variable v has a data member which can be treated as a null-terminated C string, and that's what we'll pass to sscanf() for conversion.

The sprintf(tempText,"%.15lg", tempValue) line essentially takes care of any problems with the passed-in number, such as leading or trailing spaces or other non-numeric text. You could also take the opportunity here to

validate the data, and output an error message if the parameter is not the correct type or is out of range.

The lasso_outputTagData displays a descriptive line of text which precedes the actual value for -option2, and then the value of the parameter. Notice we are actually displaying our newly-converted string value, which should be exactly the same as the passed-in number if the number was formatted perfectly from the start.

5 Now, we're going to look for the unnamed parameter. Because there's no way to ask for unnamed parameters, we're going to enumerate through all the parameters looking for one without a name. The integer count will hold the number of parameters found. Use lasso_getTagParamCount() to find out how many parameters were passed into our tag. Variable count now contains the number 3, if we were indeed passed three parameters.

```
21 int  count;
22 lasso_getTagParamCount( token, &count );
23 for (int i = 0; i < count; ++i) {
24   lasso_getTagParam( token, i, &v );
25   if( strcmp( v.data, "" ) == 0 ) {
26     lasso_outputTagData( token, "The value of unnamed param is " );
27     lasso_outputTagData( token, v.name )
28   }
29 }
```

Use lasso_getTagParam() to retrieve a parameter by its index. If you design tags that require parameters to be in a particular order, then use this function to retrieve parameters by index, starting at index 0. If the parameter is unnamed, that means it's the one needed. Note that if the user passes in more than one unnamed parameter, this loop will find all of them, and will ignore any named parameters.

Again, lasso_outputTagData displays a descriptive line of text which precedes the actual value for the unnamed parameter. Output the actual value of the unnamed parameter. Notice that the name member of the variable is what holds the text we're looking for, and the data member is an empty string.

6 Returning an error code is very important. If you return a non-zero error code, then the interpreter will throw an exception indicating that this tag failed fatally, and Lasso's standard page error routines will display an error message. Use this code for the error message.

```
30 }
31   return osErrNoErr;
32 }
```

For non-fatal errors, you can use lasso_setResultCode() and lasso_setResultMessage() to provide error codes for LDML programmers;

just make sure your tag function returns osErrNoErr from your function, otherwise Lasso's fatal error routines will be triggered.

Data Source Connector Operation

When Lasso Professional 7 starts up, it looks for module files (Windows DLLs or Mac OS X DYLIBS) in the LassoModules folder. As Lasso encounters each module, it executes the module's registerLassoModule() function once and only once. It is your job as an LCAPI developer to write code to register each of your new data source (or custom tag) function entry points in this registerLassoModule() function. Both substitution tags and data sources may be registered at the same time, and the code for them can reside in the same module. The only difference between registering a data source and a substitution tag is whether you call lasso_registerTagModule() or lasso_registerDSModule().

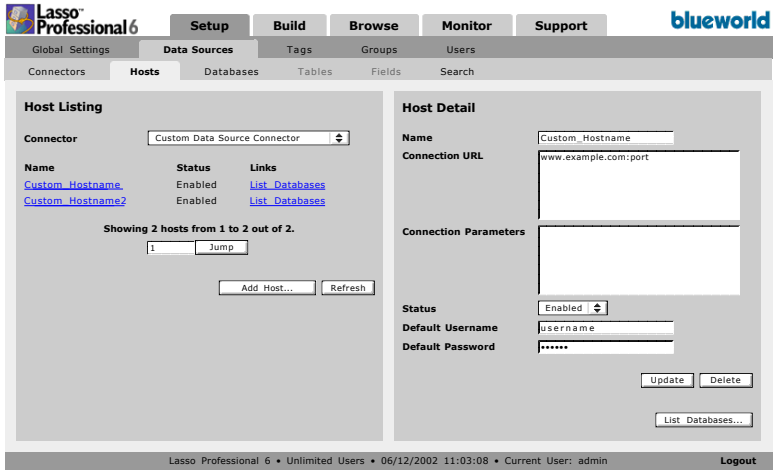
Data sources are a bit more complex than substitution tags because Lasso Service calls them with many different actions during the course of various database operations. Whereas a substitution tag only needs to know how to format itself, a data source needs to enumerate its tables, search through records, add new records, delete records, etc. Even so, this added complexity is easily handled with a single switch() statement, as you will see in the following tutorial.

Data Source Connectors and Lasso Administration

Once a custom data source connector module is registered by Lasso, it will appear in the *Setup > Data Sources > Connectors* section of Lasso Administration. If a connector appears here, then it has been installed correctly.

The administrator adds the data source connection information to the *Setup > Data Sources > Hosts* section of Lasso Administration, which sets the parameters by which Lasso connects to the data source via the connector. This information is stored in the Lasso_Internal Lasso MySQL database, where the connector can retrieve and use the data via function calls.

Figure 1: Custom Data Source Host Screen



The data that the administrator can submit in the *Setup > Data Sources > Hosts* section of Lasso Administration includes the following:

- **Name** – The administrator-defined name of the data source host.
- **Connection URL** – The URL string required for Lasso to connect to a data source via the connector. This typically includes the IP address of the machine hosting the data source.
- **Connection Parameters** – Additional parameters passed with the Connection URL. This can include the TCP/IP port number of the data source.
- **Status** – Allows the administrator to enable or disable the connector in Lasso Professional 5.
- **Default Username** – The data source username required for Lasso to gain access to the data source.
- **Default Password** – The data source password required for Lasso to gain access to the data source.

The Connection URL, Connection Parameters, Default Username, and Default Password values are passed to the data source via the lasso_getDataHost function, which is described later in this chapter.

```
LCAPICALL osError lasso_getDataHost( lasso_request_t token, auto_lasso_value_t *
host, auto_lasso_value_t * usernamepassword );
```

Data Source Connector Tutorial

This section provides a walk-through of an example data source to show how some of the LCAPI features are used. This code will be most similar to the sample SampleDataSource project, so if you want to actually build this code, then you should copy that project folder and edit the project files inside it.

The data source will simply display some simple text as each portion is called from an LDML inline which does a simple database search. It is not an effective or useful data source; it's meant to just provide an overview of what functions must be implemented. The sample data source will simulate a data source which has two databases, an Accounting database and a Customers database. Each of those databases in turn will report that it has a few tables within it. For a more complete example of a data source that is useful, look at the MySQLDataSource project.

Data Source Connector Code

Below is the code for the substitution tag module. Line numbers are provided to the left of each line of code, and are referenced in the *Data Source Connector Walk Through* section.

Data Source Connector Code

```
1  void registerLassoModule()
2  {
3    lasso_registerDSModule( "SampleDatasource", sampleDS_func, 0 );
4  }
5  osError sampleDS_func( lasso_request_t token, datasource_action_t action, const
   auto_lasso_value_t *param )
6  {
7    osError   err = osErrNoErr;
8    auto_lasso_value_t   v1, v2;
9    switch( action )
10   {
11     case datasourceInit:
12       break;
13     case datasourceTerm:
14       break;
15     case datasourceNames:
16       lasso_addDataSourceResult( token, "Accounting" );
17       lasso_addDataSourceResult( token, "Customers" );
18       break;
19     case datasourceExists:
20       if( (strcmp( param->data, "Accounting" ) != 0)
21         && (strcmp( param->data, "Customers" ) != 0) )
```

```
22          err = osErrWebNoSuchObject;
23        break;
24      case datasourceTableNames:
25        if( strcmp( param->data, "Accounting" ) == 0 ) {
26          lasso_addDataSourceResult( token, "Payroll" );
27          lasso_addDataSourceResult( token, "Payables" );
28          lasso_addDataSourceResult( token, "Receivables" );
29        }
30        if( strcmp( param->data, "Customers" ) == 0 ) {
31          lasso_addDataSourceResult( token, "ContactInfo" );
32          lasso_addDataSourceResult( token, "ItemsPurchased" );
33        }
34        break;
35      case datasourceSearch:
36        lasso_getDataSourceName( token, &v1 );
37        lasso_getTableName( token, &v2 );
38        if( strcmp( v1.data, "Accounting" ) == 0 ) {
39          int  count, i;
40          lasso_getInputColumnCount( token, &count );
41          for( i=0; i<count; i++ ) {
42            auto_lasso_value_t         columnItem;
43            lasso_getInputColumn( token, i, &columnItem );
44          }
45          if( strcmp( v2.data, "Payroll" ) == 0 ) {
46            char *row1[] = {"Samuel Goldwyn", "1955-03-27", "15000.00"};
47            unsigned int sizes1[3] = {14, 10, 8};
48            lasso_addColumnInfo( token, "Employee", false, typeChar, kProtectionNone );
49            lasso_addColumnInfo( token, "StartDate", false, typeDateTime,
kProtectionNone );
50            lasso_addColumnInfo( token, "Wages", false, typeDecimal, kProtectionNone );
51            lasso_addResultRow( token, (const char **)&row1, (unsigned int *)&sizes1,
(int)3 );
52            lasso_setNumRowsFound( token, 1 );
53          }
54        }
55        if( strcmp( v1.data, "Customers" ) == 0 ) {
56        }
57        break;
58      case datasourceAdd:
59        lasso_outputTagData( token, "datasourceAdd was called to append a record<br>"
);
60        break;
61      case datasourceUpdate:
62        lasso_outputTagData( token, "datasourceUpdate was called to replace a
record<br>" );
63        break;
64      case datasourceDelete:
```

```
65          lasso_outputTagData( token, "datasourceDelete was called to remove a
    record<br>" );
66          break;
67        case datasourceInfo:
68          lasso_outputTagData( token, "datasourceInfo was called<br>" );
69          break;
70        case datasourceExecSQL:
71          lasso_outputTagData( token, "datasourceExecSQL was called<br>" );
72          break;
73      }
74      return err;
75  }
```

Data Source Connector Walk Through

This section provides a step-by-step walk through for building the data source connector.

To build a sample LCAPI Data Source Connector:

1 Register the new data source in the required registerLassoModule() export function, as shown inlines 1-4. It's similar to the way you register a substitution tag.

```
1  void registerLassoModule()
2  {
3    lasso_registerDSModule( "SampleDatasource", sampleDS_func, 0 );
4  }
```

2 Now implement sampleDS_func, the function which gets called when any database operations are encountered.

```
5  osError sampleDS_func( lasso_request_t token, datasource_action_t action, const
   auto_lasso_value_t *param )
```

All data source functions have this prototype. When your data source function is called, it's passed an opaque "token" data structure, an integer "action" telling it what it should do, and an optional parameter which sometimes contains extra information (like a database name) needed by the action being requested at that time.

3 Set a default error return value that indicates no error. Returning a non-zero value will cause the Lasso Professional engine to report a fatal error and stop processing the page.

```
6  {
7  osError   err = osErrNoErr;
8    auto_lasso_value_t v1, v2;
9    switch( action )
10 {
```

Declare a couple of temporary variables to be used later to retrieve important values such as database names and table names. This function gets called with various different actions as Lasso Professional requests information from our data source. This switch statement distinguishes between those various actions.

4 datasourceInit is called once when Lasso Professional starts up. This gives us a chance to initialize any communications with our database back-end, and set any global variables (including semaphores) we'll need later. This is called once when Lasso Professional starts up. Because this data source is so simple, it needs no special initialization calls.

```
11  case datasourceInit:
12    break;
13  case datasourceTerm:
14    break;
15  case datasourceNames:
16      lasso_addDataSourceResult( token, "Accounting" );
17      lasso_addDataSourceResult( token, "Customers" );
18    break;
19  case datasourceExists:
20    if( (strcmp( param->data, "Accounting" ) != 0)
21      && (strcmp( param->data, "Customers" ) != 0) )
22      err = osErrWebNoSuchObject;
23    break;
```

datasourceTerm is called once when Lasso Professional shuts down. Because this data source is so simple, it needs no special shutdown code. Normally you would close your connection to your back-end data source and release any semaphores you created.

datasourceNames is called whenever Lasso Professional needs to get a list of databases which your data source provides access to. The developer must write code that discovers a list of all the databases your database 'knows about' and call lasso_addDataSourceResult() once for each found database, passing the name of the database. If the data source deals with five databases, then you would call lasso_addDataSourceResult() five times, once for each database name.

Because we are simulating a data source which knows about the Accounting and Customers databases, call lasso_addDataSourceResult() to add Accounting and Customers to the returned list of database names.

For datasourceExists, Lasso Professional is asking use if we know a particular database exists (meaning, do we control this database). The name of the database we should look up is passed in the C-string param->data. If we don't know about the database in question, then return osErrWebNoSuchObject. The conditional statement does a simple string comparison against our hard-coded database name Accounting,

and then against our hard-coded database name Customers. If neither of the previous string comparisons matched, then return the error code osErrWebNoSuchObject indicating that we do not know anything about the requested database.

5 Lasso Professional will also need to call on the database tables once per database, passing the database name in the param->data value. datasourceTableNames enumerates the list of tables within that named database.

```
24 case datasourceTableNames:
25   if( strcmp( param->data, "Accounting" ) == 0 ) {
26     lasso_addDataSourceResult( token, "Payroll" );
27     lasso_addDataSourceResult( token, "Payables" );
28     lasso_addDataSourceResult( token, "Receivables" );
29   }
```

The conditional statement checks to see if we are being asked about our Accounting database, and if so adds the Payroll table to the list of known tables by calling lasso_addDataSourceResult(), and so forth.

6 Next, Lasso Professional will need to check to see if there are inquiries regarding the Customers database.

```
30 if( strcmp( param->data, "Customers" ) == 0 ) {
31   lasso_addDataSourceResult( token, "ContactInfo" );
32   lasso_addDataSourceResult( token, "ItemsPurchased" );
33 }
34   break;
```

Lasso Professional adds the ContactInfo table to the list of known tables by calling lasso_addDataSourceResult().Continue adding table names to the Customers database by calling lasso_addDataSourceResult(), this time for the ItemsPurchased table.

7 Use datasourceSearch to perform a search on the database.

```
35 case datasourceSearch:
36   lasso_getDataSourceName( token, &v1 );
37   lasso_getTableName( token, &v2 );
38     if( strcmp( v1.data, "Accounting" ) == 0 ) {
39       int  count, i;
40       lasso_getInputColumnCount( token, &count );
41         for( i=0; i<count; i++ ) {
42         auto_lasso_value_t columnItem;
43         lasso_getInputColumn( token, i, &columnItem );
44         }
45     }
```

All of the information (database and table names, search arguments, sort arguments, etc.) can be retrieved, and a search can be performed by calling various LCAPI functions such as lasso_getDataSourceName() and

lasso_getTableName() to get the name of the database and table, respectively. A complete list of data source functions is here.

lasso_getDataSourceName asks Lasso Professional to give us the database name which is to be searched. This is often the value of the -Database parameter value in an inline tag. lasso_getTableName asks Lasso Professional to give us the table name to be searched. This is often the value from the -Layout or -Table parameter value from an inline tag.

The conditional statement checks to see if the database being searched is Accounting. If so, declare a couple of temporary integers, one for holding the number of search parameters. lasso_getInputColumnCount asks Lasso how many search fields (columns) were specified by the user for this search. For instance, if the LDML inline tag passed three different fields to be searched, then lasso_getInputColumnCount() returns 3.

Declare a temporary variable which will receive the name/value pair information from the next line of code. Retrieve the name/value text for the n'th requested search parameter. For instance, an inline will fill the columnItem variable with the values Employee, fred the first time through the loop, and Wages, 15000 the second time through the loop.

```
[Inline: -Database='Accounting', -Table='Payroll', 'Employee'='fred',
'Wages'='15000']
```

8 Next, set a conditional statement to ask if the Payroll table is being searched. If so, we'll set up some fake hard-coded data in the next few lines of code. Declare an array of strings which represents the three fields we will return for this search. Declare an array of field sizes to match the lengths of the strings created on the previous line.

```
46  if( strcmp( v2.data, "Payroll" ) == 0 ) {
47    char *row1[] = {"Samuel Goldwyn", "1955-03-27", "15000.00"};
48    unsigned int sizes1[3] = {14, 10, 8};
49    lasso_addColumnInfo( token, "Employee", false, typeChar, kProtectionNone );
50    lasso_addColumnInfo( token, "StartDate", false, typeDateTime, kProtectionNone
      );
51    lasso_addColumnInfo( token, "Wages", false, typeDecimal, kProtectionNone );
52    lasso_addResultRow( token, (const char **)&row1, (unsigned int *)&sizes1(int)3
      );lasso_setNumRowsFound( token, 1 );
53    }
54  }
55  if( strcmp( v1.data, "Customers" ) == 0 ) {
56  }
57    break;
58  case datasourceAdd:
59    lasso_outputTagData( token, "datasourceAdd was called to append a record<br>"
      );
60    break;
```

```
61  case datasourceUpdate:
62      lasso_outputTagData( token, "datasourceUpdate was called to replace a
    record<br>" );
63    break;
64  case datasourceDelete:
65    lasso_outputTagData( token, "datasourceDelete was called to remove a
    record<br>" );
66    break;
67  case datasourceInfo:
68    lasso_outputTagData( token, "datasourceInfo was called<br>" );
69    break;
70  case datasourceExecSQL:
71    lasso_outputTagData( token, "datasourceExecSQL was called<br>" );
72    break;
73  }
74    return err;
75  }
```

lasso_addColumnInfo tells LCAPI what the column names and data types are. Do this by calling lasso_addColumnInfo() once per column. In this line, the Employee column is described as text (typeChar) with no protection (kProtectionNone). In the next line, the StartDate column is described as date (typeDateTime) with no protection (kProtectionNone).

The last column Wages is described as being numeric (typeDecimal), with no protection (kProtectionNone). Now lasso_addResultRow() can be called as many times as there are rows of data to return. In this case, only one row is returned. Now LCAPI must be told how many total rows were found.

Data Type Operation

Creating a new data type in LCAPI 6 is similar to creating a substitution tag. When Lasso Professional 7 starts up, it scans the LassoModules folder for module files (Windows DLLs or Mac OS X DYLIBS). As it encounters each module, it executes the registerLassoModule() function for that module. The developer registers the LCAPI data types or tags implemented by the module inside this function. Registering data type initializers differs from registering normal substitution tags in that the third parameter in lasso_registerTagMode is the value flag_typeInitializer.

```
void registerLassoModule()
{
  lasso_registerTagModule( “CAPITester”, “test_type”, myTypeInitFunc,
  flag_typeInitializer, “simple test LCAPI type” );
}
```

The prototype of a LCAPI type initializer is the same as a regular LCAPI substitution tag function. Lasso will call the type initializer each time a new instance of the type is created.

```
osError myTypeInitFunc( lasso_request_t token, tag_action_t action );
```

When the type initializer function is called, a new instance of the type is created using lasso_typeAllocCustom. This new instance will be created with no data or tag members.

```
osError myTypeInitFunc( lasso_request_t token, tag_action_t action )
{
  lasso_type_t theNewInstance = NULL;
  lasso_typeAllocCustom( token, &theNewInstance, "test_type" );
```

Once the type is created, new data and tag members can be added to it using lasso_typeAddMember. Data members can be of any type and should be allocated using any of the LCAPI type allocation calls. Tag members are allocated using lasso_typeAllocTag. LCAPI tag member functions are implemented just like any other LCAPI tag. In the example below, myTagMemberFunction is a function with the standard LCAPI tag prototype.

```
  const char * kStringData = "This is a string member.";
  lasso_type_t stringMember = NULL;
  lasso_typeAllocString( token, &stringMember, kStringData, strlen(kStringData) );
  lasso_typeAddMember( token, theNewInstance, "member1", stringMember );
  lasso_type_t tagMember = NULL;
  lasso_typeAllocTag( token, &tagMember, myTagMemberFunction );
  lasso_typeAddMember( token, theNewInstance, "member2", tagMember );
```

The final step in creating a new LCAPI data type instance is to return the new type to Lasso as the tag's return value. After the type is returned, Lasso will complete the creation of the type by instantiating the new type's parent types.

```
  lasso_returnTagValue( token, theNewInstance );
  return osErrNoErr;
}
```

Data Type Tutorial

This tutorial walks through the main points of creating a custom data type using LCAPI 6. The resulting data type is a "file" type, and the ability to open, close, read and write to the file are implemented via the following member tags:

```
[File->Open]
[File->Close]
[File->Read]
[File->Write]
```

Data Types Code

The example project and source files contain over 800 lines of code, and are located in the following folder:

Lasso Professional 7/Documentation/4-ExtendingLasso/LCAPI/Tags/CAPIFile

Do to the length of the project file (CAPIFile.cpp), the entire code is not shown here. The *Data Type Walk Through* section provides a conceptual overview of the operation behind the file type example, and describes the basic LCAPI functions used to implement it.

Data Type Walk Through

This section provides a step-by-step conceptual walk through for building a custom file data type.

To build a custom data type:

1 The first step in creating a custom type is to register the type's initializer. Type initializers are registered in the same way that regular tag functions are registered. The only difference being that flag_typeInitializer should be passed for the fourth (flags) parameter:

```
void registerLassoModule()
{
  lasso_registerTagModule("CAPIFile", "file", file_init, flag_typeInitializer, "Initializer
for the file type.");
}
```

This concept is illustrated in lines 95-129 of the CAPIFile.cpp file.

```
95  void registerLassoModule()
96  {
    ...
128   lasso_registerTagModule("CAPIFile", "file", file_init, flag_typeInitializer,
    "Initializer for the file type.");
129 }
```

2 The registered type initializer will be called each time a new file type is created. In the above case, the LCAPI function file_init was registered as being the initializer. The prototype for file_init should look like any other LCAPI function:

```
osError file_init(lasso_request_t token, tag_action_t action);
```

This concept is illustrated in line 272 of the CAPIFile.cpp file.

```
272  osError file_init(lasso_request_t token, tag_action_t action)
```

3 The file_init function will now be called whenever file is used in a script. Within the type initializer, the type's member tags are added. Each member tag is implemented by its own LCAPI tag function. However, before members can be added, the new blank type must be created using lasso_typeAllocCustom.

```
osError file_init(lasso_request_t token, tag_action_t)
{
  lasso_type_t file;
  lasso_typeAllocCustom(token, &file, "file");
```

lasso_typeAllocCustom can only be used within a properly registered type initializer. The value it produces should always be the return value of the tag as set by the lasso_returnTagValue function.

This concept is illustrated in lines 273-277 of the CAPIFile.cpp file.

```
273  {
274    lasso_type_t file;
 ...
277    lasso_typeAllocCustom(token, &file, KFileTypeName);
```

4 Once the blank type has been created, members can be added to it. Members will be added for open, close, read and write.

```
lasso_type_t mem;
lasso_typeAllocTag(token, &mem, file_open);
lasso_typeAddMember(token, file, "open", mem);

lasso_typeAllocTag(token, &mem, file_close);
lasso_typeAddMember(token, file, "close", mem);

lasso_typeAllocTag(token, &mem, file_read);
lasso_typeAddMember(token, file, "read", mem);

lasso_typeAllocTag(token, &mem, file_write);
lasso_typeAddMember(token, file, "write", mem);
```

This concept is illustrated in lines 289-299 of the CAPIFile.cpp file.

```
289   #define ADD_TAG(NAME, FUNC) {   lasso_type_t mem;\
290                         lasso_typeAllocTag(token, &mem, FUNC);\
291                         MAKE_REF(mem);\
292                         lasso_typeAddMember(token, file, NAME, mem);\
293             }
 ...
296   ADD_TAG(kMemOpen, file_open);
297   ADD_TAG(kMemClose, file_close);
298   ADD_TAG(kMemRead, file_read);
299   ADD_TAG(kMemWrite, file_write);
```

5 The final member tag to add is the onDestroy member. This tag will be called automatically by Lasso when the type goes away. Adding this tag will ensure that the file on disk is closed properly if the member tag function file_close is not called.

```
lasso_typeAllocTag(token, &mem, file_close);
lasso_typeAddMember(token, file, "onDestroy", mem);
```

This concept is illustrated in line 313 of the CAPIFile.cpp file.

```
313   ADD_TAG(kMemOnDestroy, file_onDestroy);
```

6 At this point, the return value should be set. Keep in mind that the new file type is completely blank except for the members that were added above. No inherited members are available at this point. Inherited members are only added after the LCAPI type initializer returns.

```
lasso_returnTagValue(token, file);
```

This concept is illustrated in line 284 of the CAPIFile.cpp file.

```
284  lasso_returnTagValue(token, file);
```

Note: For brevity, this example will not cover accepting parameters in the type initializer. The full CAPIFile project illustrates accepting parameters in the initializer to open the file under various read and write permissions.

7 There were no errors in the type initialization process, so return a "no error" code to Lasso, completing the type's initialization.

```
    return osErrNoErr;
}
```

This concept is illustrated in line 358 of the CAPIFile.cpp file.

```
358  return osErrNoErr;
```

8 The new file type has now been initialized and made available to the caller in the script. The first member of the file type is [File->Open], which is implemented as the LCAPI function file_open.

```
osError file_open(lasso_request_t token, tag_action_t)
{
```

This concept is illustrated in lines 361-362 of the CAPIFile.cpp file.

```
361  osError file_open(lasso_request_t token, tag_action_t action)
362  {
```

9 The first step in implementing a member tag is to acquire the "self" instance. The self is the instance upon which the member call was made.

```
lasso_type_t self = NULL;
lasso_getTagSelf(token, &self);
if ( self )
{
```

This concept is illustrated in lines 434-437 of the CAPIFile.cpp file.

```
434 lasso_type_t self = NULL;
435 lasso_getTagSelf(token, &self);
436 if ( self )
437 {
```

10 Once the self is successfully acquired and is not null, the rest of the member tag can proceed. This member tag accepts one parameter, which is the path to the file that will be opened. Since the path is a string value, it can be acquired using lasso_getTagParam. If the path parameter was not passed to the open member tag, an error should be returned and indicated to the user.

```
auto_lasso_value_t path;
if ( lasso_getTagParam(token, 0, &path) != osErrNoErr )
{
  lasso_setResultMessage(token, "file->open requires the path to the file to
open.");
  return osErrInvalidParameter;
}
```

This concept is illustrated in lines 363-390 of the CAPIFile.cpp file.

```
363 file_desc_t * desc = GetFileDesc(token, false);
364 if ( desc )
365 {
366   if ( desc->fFileStr != NULL ) // close it
367     fclose(desc->fFileStr);
368   // delete the struct
369   desc->fFileStr = NULL;
370
371   // see what parameters we are being initialized with
372   int count;
373   lasso_getTagParamCount(token, &count);
374
375   if ( count < 2 )
376   {
377     lasso_setResultMessage(token, "file->open requires at least a file path and
    open mode.");
378     lasso_setResultCode(token, osErrInvalidParameter);
379     return osErrInvalidParameter;
380   }
381
382   if ( count > 0 ) // we are given *at the least* a path
383   {
384     // first param is going to be a string, so use the LCAPI 5 call to get it
385     auto_lasso_value_t pathParam;
386     pathParam.name = "";
387     lasso_getTagParam(token, 0, &pathParam);
388
```

```
389       desc->fPath = pathParam.name;
390    }
```

11 Now that the path parameter has been successfully acquired, permissions should be checked to make sure access to the file is permitted by Lasso security.

```
if ( lasso_operationAllowed(token, kWriteFiles, (void*)path.name) != osErrNoErr )
{
  lasso_setResultMessage(token, "file->open: Permission to open the file was
denied by Lasso security.");
  return osErrNoPermission);
}
```

This concept is illustrated in lines 232-237 of the CAPIFile.cpp file.

```
232 if ( lasso_operationAllowed(token, op, const_cast<char*>(path)) != osErrNoErr )
233 {
234   lasso_setResultMessage(token, "Permission to open the file was denied by
    Lasso security.");
235   lasso_setResultCode(token, osErrNoPermission);
236   return NULL;
237 }
```

12 If the current user has permission, the Lasso internal path should be converted to the platform specific path. This is a three-step process that begins with fully qualifying the path. This will ensure that relative paths are converted to root paths. The second step is to resolve the path. This converts root path to a complete path which will include the hard drive name, or /// if used on a Unix platform. The final step is to convert the path into a platform-specific format that will be understood by the platform-specific [File->Open] calls.

```
osPathname qualifiedPath;
osPathname resolvedPath;
osPathname platformPath;

lasso_fullyQualifyPath( token, path.name, qualifiedPath );
lasso_resolvePath( token, qualifiedPath, resolvedPath );
lasso_getPlatformSpecificPath( resolvedPath, platformPath );
```

This concept is illustrated in lines 197-203 of the CAPIFile.cpp file.

```
197 {
198 osPathname qualifiedPath;
199 osPathname resolvedPath;
200 lasso_fullyQualifyPath( token, inPath, qualifiedPath );
201 lasso_resolvePath( token, qualifiedPath, resolvedPath );
202 lasso_getPlatformSpecificPath( resolvedPath, outPath );
203 }
```

13 Once security is checked and the path is properly converted, the actual file can be opened using the file system calls supplied by the operating system.

```
FILE * f = fopen(platformPath, "r+b");
```

This concept is illustrated in line 242 of the CAPIFile.cpp file.

```
242  FILE * f = fopen(xformPath, openMode);
```

14 If the file is opened without error, we want to save the resulting FILE pointer for other member calls. LCAPI 6 provides a convenient storage location for custom types. This location is not used by Lasso itself, and the value will never be modified.

```
lasso_typeSetCustomPtr(self, (void*)f);
```

This concept is illustrated in line 281 of the CAPIFile.cpp file.

```
281  lasso_typeSetCustomPtr(file, desc);
```

15 The FILE pointer can now be retrieved using the lasso_typeGetCustomPtr LCAPI function. No error has occurred while opening the file, so complete the function call and return "no error".

```
    }
    return osErrNoErr;
}
```

This concept is illustrated in lines 416-418 of the CAPIFile.cpp file.

```
416  }
417  return osErrNoErr;
418   }
```

16 The next member tag to implement is [File->Read]. This member takes one parameter and returns a value. The parameter is the number of bytes to read from the file. The return value is the resulting file data, if any.

```
osError file_read(lasso_request_t token, tag_action_t action)
{
  lasso_type_t self = NULL;
  lasso_getTagSelf(token, & self);
  if ( self )
  {
```

This concept is illustrated in lines 432-437 of the CAPIFile.cpp file.

```
432  osError file_read(lasso_request_t token, tag_action_t action)
433   {
434    lasso_type_t self = NULL;
435    lasso_getTagSelf(token, &self);
436    if ( self )
437    {
```

17 The parameter of this tag is an integer. Use the lasso_getTagParam2 function to get the tag's parameter, which will produce the parameter as a lasso_type_t and convert it to a 64-bit integer.

```
lasso_type_t numT = NULL;
if ( lasso_getTagParam2(token, 0, &numT) != osErrNoErr )
{
  lasso_setResultMessage(token, "file->read required one parameter: the
number of bytes to read from the file.");
  return osErrInvalidParameter;

}
osInt64 num = 0;
lasso_typeGetInteger(token, numT, &num);
```

This concept is illustrated in line 438-446 of the CAPIFile.cpp file.

```
438  lasso_type_t num = NULL;
439  if ( lasso_getTagParam2(token, 0, &num) == osErrNoErr )
440  {
441    file_desc_t * desc = NULL;
442    lasso_typeGetCustomPtr(self, reinterpret_cast<void**>(&desc));
443    if ( desc && desc->fFileStr != NULL )
444    {
445      osInt64 n;
446      lasso_typeGetInteger(token, num, &n);
```

18 Now fetch the pointer to the FILE that was stored when [File->Open] was called and allocate a buffer into which we will read the data.

```
FILE * f = NULL;
lasso_typeGetCustomPtr(self, (void**)&f);

int rNum = (int)num;
char * buffer = new char[rNum];

rNum = fread(buffer, sizeof(char), rNum, f);
```

This concept is illustrated in lines 242-248 and lines 448-456 of the CAPIFile.cpp file.

```
242  FILE * f = fopen(xformPath, openMode);
243  if ( f == NULL )
244  {
245    lasso_setResultCode(token, osErrFile);
246    SetResultMessage(token, errno);
247  }
248  return f;
 ...
448        char * readInto = new char[(size_t)n];
449
450        size_t wasRead = fread(readInto, sizeof(char), (size_t)n, desc->fFileStr);
451        if ( wasRead > 0 )
```

```
452         {
453           lasso_type_t ret;
454           lasso_getTagReturnValue(token, &ret);
455           lasso_typeSetString(token, ret, readInto, (int)wasRead);
456         }
```

19 The final step is to return the read data. Because the data is a string, we can use lasso_outputTagData2, which will allow us to return a string along with it's length.

```
      lasso_outputTagData2(token, buffer, rNum);
      delete [] buffer;
    }
    return osErrNoErr;
  }
```

This concept is illustrated in lines 458-475 of the CAPIFile.cpp file.

```
458 delete [] readInto;
459         }
460       else
461         {
462         lasso_setResultMessage(token, "file->read requires that the file be open with
    read permission.");
463         lasso_setResultCode(token, osErrInvalidParameter);
464         return osErrInvalidParameter
465         }
466       }
467     else
468       {
469       lasso_setResultMessage(token, "file->read requires a single parameter: the
    number of bytes to read.");
470       lasso_setResultCode(token, osErrInvalidParameter);
471       return osErrInvalidParameter;
472       }
473     }
474     return osErrNoErr;
475 }
```

20 The remaining tag functions are implemented in a similar manner. Study the CAPIFile example for a more in-depth and complete example of how to properly construct custom data types in LCAPI 6.

LCAPI Function Reference

This section lists all functions provided in LCAPI 6. These functions can be used to register tags and data sources, allocate memory, return error messages, get tag or parameter information, get client and server environment information, output text, read and set MIME headers, access LDML

variables, interpret and execute arbitrary LDML tags, store persistent data, check if a user is an administrator, perform data source functions, and safely access multiuser and multi-threaded resources.

Registration

registerLassoModule()

You must have a function of this name defined in your module's exported functions. Lasso calls this once at startup to give your module a chance to register its tags and data sources.

```
void registerLassoModule();
```

lasso_registerTagModule()

Your code must call this once at startup (from within your registerLassoModule() function) to register a tag with Lasso. When Lasso encounters a custom tag of tagName, it calls the C function func. Tags can be registered as being either a type initializer, asynchronus tag, container tag, or normal substitution tag. The tag type is specified by the flags parameter and must be one of the following values: flag_typeInitializer, flag_typeSubstitutionTag, flag_typeAsync, or flag_typeContainerTag.

```
LCAPICALL osError lasso_registerTagModule(
const char * moduleName,
const char * tagName,
lasso_tag_func func,
int flags,
const char * description );
```

lasso_registerTagModuleW()

Same as lasso_registerLassoModule(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_registerTagModuleW(
const UChar * moduleName,
const UChar * tagName,
lasso_tag_func func,
int flags,
const UChar * description);
```

lasso_registerDSModule()

Your code must call this once at startup (from within your registerLassoModule() function) to register a data source with Lasso

Professional. When Lasso encounters a data source request for moduleName, it calls the C function func.

```
LCAPICALL osError lasso_registerDSModule(
const char * moduleName,
lasso_ds_func func,
int flags );
```

lasso_registerDSModuleW()

Same as lasso_registerDSModule(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_registerDSModuleW(
const UChar * moduleName,
lasso_ds_func func, int flags );
```

lasso_registerConstant()

This function registers a constant with Lasso.

```
LCAPICALL osError lasso_registerConstant(
const char * name,
lasso_type_t val );
```

lasso_registerConstantW()

Same as lasso_registerConstant(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_registerConstantW(
const UChar * name,
lasso_type_t val );
```

Memory Allocation

malloc(), free()

You may use malloc() and free() as you would in any normal program, just make sure you don't leave any memory leaks.

```
malloc(), free()
```

lasso_allocValue()

This is rarely used, but if you need to create your own lasso_value_t, you can use this function to do it. lasso_allocValue() allocates a lasso_value_t with the indicated data. Anything allocated with this function will not be garbage collected by Lasso and must be freed using lasso_freeValue.

```
LCAPICALL osError lasso_allocValue(
lasso_value_t * value,
const char * name,
unsigned int nameSize,
const char * data,
unsigned int dataSize,
LP_TypeDesc dataType );
```

lasso_allocValueW()

Same as lasso_allocValue(), but is Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_allocValue(
lasso_value_w_t * result,
const UChar * name,
unsigned int nameSize,
const UChar * data,
unsigned int dataSize,
LP_TypeDesc type);
```

lasso_allocValueConv()

Converts a lasso_value_t value to Unicode.

```
LCAPICALL osError lasso_allocValueConv(
lasso_value_t * result,
const UChar * name,
unsigned int nameSize,
const char * nameEncoding,
const UChar * data,
unsigned int dataSize,
const char * dataEncoding,
LP_TypeDesc type);
```

lasso_freeValue()

This function frees all values allocated using lasso_allocValue(). Do not pass an auto_lasso_value_t to this function.

```
LCAPICALL osError lasso_freeValue(
lasso_value_t * result );
```

lasso_freeValueW()

Same as lasso_freeValue(), but is Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_freeValueW(
lasso_value_w_t * result);

// Allows storage of an opaque value
```

lasso_setPtrMember()

This function sets a pointer member.

```
LCAPICALL osError lasso_setPtrMember(
lasso_request_t token,
lasso_type_t self,
const char * name,
void * data);
```

lasso_setPtrMemberW()

Same as lasso_setPtrMember(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_setPtrMemberW(
lasso_request_t token,
lasso_type_t self,
const UChar * name,
void * data);
```

lasso_getPtrMember()

This function gets a pointer member.

```
LCAPICALL osError lasso_getPtrMember(
lasso_request_t token,
lasso_type_t self,
const char * name,
void ** data);
```

lasso_getPtrMemberW()

Same as lasso_getPtrMember(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_getPtrMemberW(
lasso_request_t token,
lasso_type_t self,
const UChar * name,
void ** data);
```

Internal Values

lasso_getRequestParam()

Fetches an internal server value such as server port, cookies, root path, username, etc. You may request the parameters shown below, not all of which are available on all HTTP servers.

```
LCAPICALL osError lasso_getRequestParam(
lasso_request_t token,
RequestParamKeyword key,
auto_lasso_value_t * result );
```

Parameter	Description
rpSearchArgKeyword	All text in URL after the question mark.
rpUserKeyword	Username sent from browser.
rpPasswordKeyword	Password sent from browser.
rpAddressKeyword	IP address of client browser.
rpPostKeyword	HTTP object body (form data, etc.).
rpMethodKeyword	GET or POST, depending on <form method>.
rpServerName	IP address of server on which the Web server is running.
rpServerPort	IP port this hit came to (80 is common, 443 for SSL).
rpScriptName	Relative path from server root to this Lasso format file.
rpContentType	MIME header sent from client browser.
rpContentLength	The length in bytes of the POST data sent from <form POST>.
rpReferrerKeyword	URL of referring page.
rpUserAgentKeyword	Browser name and type.
rpClientIPAddress	IP address of client browser.
rpFullRequestKeyword	All MIME headers, uninterpreted.

Error Messages and Result Codes

lasso_setResultCode()

Sets the result code that can be displayed if the LDML programmer inserts [Error_CurrentError: ErrorCode] into the format file after executing a custom LCAPI tag.

```
LCAPICALL osError lasso_setResultCode(
lasso_request_t token,
osError err );
```

lasso_setResultMessage()

Sets the error message that can be displayed if the LDML programmer inserts [Error_CurrentError: ErrorMessage] into the format file after executing a custom LCAPI tag.

```
LCAPICALL osError lasso_setResultMessage(
lasso_request_t token,
const char * msg );
```

lasso_setResultMessageW()

Same as lasso_setResultMessage(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_setResultMessageW(
lasso_request_t token,
const UChar * msg );
```

Tag and Parameter Info

lasso_getTagName()

Fetches the name of the tag that triggered this call (e.g. in the case of [my_tag: ...] the resulting value would be my_tag). This makes it possible to design a single tag function which can perform the duties of many different LDML tags, perhaps ones that all have similar functionality but different names.

```
LCAPICALL osError lasso_getTagName(
lasso_request_t token,
auto_lasso_value_t * result);
```

lasso_getTagNameW()

Same as lasso_getTagName(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_getTagNameW(
lasso_request_t token,
auto_lasso_value_w_t * result);
```

lasso_getTagParamCount()

Fetches the number of parameters that were passed to the tag. For instance, [my_tag: 'hello', -option=1, -hilite=false] will report that three parameters were passed (unnamed parameters are treated just like any other parameter).

```
LCAPICALL osError lasso_getTagParamCount(
lasso_request_t token,
int * result );
```

lasso_getTagParam()

Gets the name and value of a parameter given its index. Parameters are numbered left-to-right, starting at index 0: [my_tag: -param0='value0', -param1='value1', -param2=2].

```
LCAPICALL osError lasso_getTagParam(
lasso_request_t token,
int paramIndex,
auto_lasso_value_t * result );
```

lasso_getTagParamW()

Same as lasso_getTagParam(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_getTagParamW(
lasso_request_t token,
int paramIndex,
auto_lasso_value_w_t * result );
```

lasso_getTagParam2()

Get the parameter using the parameter index. This function differs from lasso_getTagParam() in that it preserves the actual type of the parameter instead of automatically converting it to a string. Keyword/value pairs are returned as a typePair type.

```
LCAPICALL osError lasso_getTagParam2(
lasso_request_t token,
int paramIndex,
lasso_type_t * result )
```

lasso_tagParamIsDefined()

Returns osErrNoErr if the parameter was defined, anything else means it wasn't.

```
LCAPICALL osError lasso_tagParamIsDefined(
lasso_request_t token,
const char * paramName );
```

lasso_tagParamIsDefinedW()

Same as lasso_tagParamIsDefined(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_tagParamIsDefinedW(
lasso_request_t token,
const UChar * paramName );
```

lasso_findTagParam()

Finds and fetches a tag parameter by name. A return value of osErrNoErr means the parameter was found successfully.

```
LCAPICALL osError lasso_findTagParam(
lasso_request_t token,
const char * paramName,
auto_lasso_value_t * result );
```

lasso_findTagParamW()

Same as lasso_findTagParam(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_findTagParamW(
lasso_request_t token,
const UChar * paramName,
auto_lasso_value_w_t * result );
```

lasso_findTagParam2()

Finds and returns a tag parameter by name while preserving the original type. A returned value of osErrNoErr means the parameter was successfully found.

```
LCAPICALL osError lasso_findTagParam2(
lasso_request_t token,
const char * paramName,
lasso_type_t * result );
```

lasso_findTagParam2W()

Same as lasso_findTagParam2(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_findTagParam2W(
lasso_request_t token,
const UChar * paramName,
lasso_type_t * result );
```

lasso_getTagSelf()

This function is used in LCAPI tags that are members of a custom type. It returns the type instance of which the current call is a member.

```
LCAPICALL osError lasso_getTagSelf(
lasso_request_t token,
lasso_type_t * self );
```

lasso_childrenRun()

Used to execute the contents of a container tag. Tags become containers when the flag_typeContainerTag flag is used. The result parameter will contain the combined result data for all tags contained.

```
LCAPICALL osError lasso_childrenRun(
lasso_request_t token,
lasso_type_t * result );
```

lasso_runRequest()

Creates and runs a new LCAPI call on the given lasso_tag_func. If there is already an active request on the current thread, the lasso_tag_func will be run within the context of that thread. If there is no active request on the current thread, a new request will be created and run based on the global context. The tag_action_t parameter is passed to the lasso_tag_func and can be used to signal or pass information to the function.

```
LCAPICALL osError lasso_runRequest(
lasso_tag_func func,
tag_action_t action,
int unused );
```

Returning Tag Data

lasso_returnTagValue()

Specifies the return value for the tag. Note that only a single lasso_returnTagValue can be used from within a tag. lasso_returnTagValue is the prefered method for returning tag data as it allows data of any type to be returned (including binary data), while lasso_outputTagData is restricted to printable text data. lasso_returnTagValue() can also be optimized for boolean, integer, decimal, string, and Unicode strings using specialized function variations for each type, as shown below.

```
LCAPICALL osError lasso_returnTagValue(
lasso_request_t token,
lasso_type_t value );

LCAPICALL osError lasso_returnTagValueBoolean(
lasso_request_t token,
bool b );

LCAPICALL osError lasso_returnTagValueInteger(
lasso_request_t token,
osInt64 i );
```

```
LCAPICALL osError lasso_returnTagValueString(
lasso_request_t token,
const char * p,
int l );

LCAPICALL osError lasso_returnTagValueStringW(
lasso_request_t token,
const UChar * p,
int l );

LCAPICALL osError lasso_returnTagValueDecimal(
lasso_request_t token,
double d );
```

lasso_getTagReturnValue()

This function provides direct access to the tag's return value. In many cases, directly manipulating this value can yield better performance than lasso_returnTagValue().

```
LCAPICALL osError lasso_getTagReturnValue(
lasso_request_t token,
lasso_type_t * outValue );
```

lasso_outputTagData()

Output some data onto the page. Lasso will take care of encoding. This can be called as many times as needed.

```
LCAPICALL osError lasso_outputTagData(
lasso_request_t token,
const char * data );
```

lasso_outputTagData2()

Same as lasso_outputTagData(), but outputs a bytes data type and allows the data length to be defined.

```
osError lasso_outputTagData2(
lasso_request_t token,
const char * data,
int length );
```

lasso_outputTagData2W()

Same as lasso_outputTagData2(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_outputTagData2W(
lasso_request_t token,
const UChar * data,
int length );
```

lasso_outputTagBytes()

Outputs a bytes data type. This can be called as many times as needed.

```
osError lasso_outputTagBytes(
lasso_request_t token,
const char * data,
int length );
```

lasso_outputTagDataF()

lasso_outputTagDataF takes the same formatting flags as printf().

```
LCAPICALL osError lasso_outputTagDataF(
lasso_request_t token,
const char * format, ... );
```

Data Types

lasso_typeAlloc()

This function will allocate a new type instance. The type is specified by the typeName parameter. An array of parameters can be passed to the type initializer. Types created through this function will be automatically destroyed after the LCAPI call has returned. In order to prevent this, lasso_typeDetach should be used.

```
LCAPICALL osError lasso_typeAlloc (
 lasso_request_t token,
const char * typeName,
int paramCount,
lasso_type_t * paramsArray,
lasso_type_t * outType );
```

lasso_typeAllocW()

Same as lasso_typeAlloc(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeAllocW (
lasso_request_t token,
const UChar * typeName,
int paramCount,
lasso_type_t * paramsArray,
lasso_type_t * outType);
```

lasso_typeFree()

Attempts to free a type created using lasso_typeAlloc or any other method. The lasso_request_t token may be NULL if the provided type has been detached using lasso_typeDetach.

```
LCAPICALL osError lasso_typeFree(
 lasso_request_t token,
lasso_type_t inType);
```

lasso_typeDetach()

Prevents the type from being destroyed once the LCAPI call returns. Types that have been detached must eventually be destroyed using lasso_typeFree (passing NULL as the request token) or a memory leak will occur.

```
LCAPICALL osError lasso_typeDetach(
lasso_request_t token,
lasso_type_t toDetach );
```

lasso_typeAllocNull()

This function allows new instances of NULL data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocNull(
lasso_request_t token,
lasso_type_t * outNull );
```

lasso_typeAllocString()

This function allows new instances of string data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocString(
lasso_request_t token,
lasso_type_t * outString,
const char * value,
int length );
```

lasso_typeAllocStringW()

Same as lasso_typeAllocString(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeAllocStringW (
lasso_request_t token,
lasso_type_t * outString,
const UChar * value,
int length );
```

lasso_typeAllocInteger()

This function allows new instances of integer data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocInteger(
lasso_request_t token,
lasso_type_t * outInteger,
osInt64 value );
```

lasso_typeAllocDecimal()

This function allows new instances of decimal data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocDecimal(
lasso_request_t token,
lasso_type_t * outDecimal,
double value );
```

lasso_typeAllocPair()

This function allows new instances of pair data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocPair(
lasso_request_t token,
lasso_type_t * outPair,
lasso_type_t first,
lasso_type_t second );
```

lasso_typeAllocReference()

This function allows new instances of reference data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocReference(
lasso_request_t token,
lasso_type_t * outRef,
lasso_type_t referenced );
```

lasso_typeAllocTag()

This function allows new instances of tag data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocTag(
lasso_request_t token,
lasso_type_t * outTag,
lasso_tag_func nativeTagFunction);
```

lasso_typeAllocArray()

This function allows new instances of array data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocArray(
lasso_request_t token,
lasso_type_t * outArray,
int count,
lasso_type_t * elements );
```

lasso_typeAllocMap()

This function allows new instances of map data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocMap(
lasso_request_t token,
lasso_type_t * outMap,
int count,
lasso_type_t * elements );
```

lasso_typeAllocBoolean()

This function allows new instances of boolean data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
LCAPICALL osError lasso_typeAllocBoolean(
lasso_request_t token,
lasso_type_t * outBool,
bool inValue );
```

lasso_typeGetString()

This function gets the data from a previously created string instance.

```
LCAPICALL osError lasso_typeGetString(
lasso_request_t token,
lasso_type_t type,
auto_lasso_value_t * val );
```

lasso_typeGetStringW()

Same as lasso_typeGetString(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeGetStringW(
lasso_request_t token,
lasso_type_t type,
auto_lasso_value_w_t * val);
```

lasso_typeGetStringConv()

This function converts a string instance to Unicode.

```
LCAPICALL osError lasso_typeGetStringConv(
lasso_request_t token,
lasso_type_t type,
auto_lasso_value_t * val,
const char * conv );
```

lasso_typeGetInteger()

This function gets the data from a previously created integer instance.

```
LCAPICALL osError lasso_typeGetInteger(
lasso_request_t token,
lasso_type_t type,
osInt64 * out );
```

lasso_typeGetDecimal()

This function gets the data from a previously created decimal instance.

```
LCAPICALL osError lasso_typeGetDecimal(
lasso_request_t token,
lasso_type_t type,
double * out );
```

lasso_typeGetBoolean()

This function gets the data from a previously created boolean instance.

```
LCAPICALL osError lasso_typeGetBoolean(
lasso_request_t token,
lasso_type_t type,
bool * out );
```

lasso_typeSetString()

This function sets the data from a previously created string instance.

```
LCAPICALL osError lasso_typeSetString(
lasso_request_t token,
lasso_type_t type,
const char * val,
int len );
```

lasso_typeSetStringW()

Same as lasso_typeSetString(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeSetStringW(
lasso_request_t token,
lasso_type_t type,
const UChar * val,
int len );
```

lasso_typeSetStringConv()

This function converts a string instance to Unicode.

```
LCAPICALL osError lasso_typeSetStringConv(
lasso_request_t token,
lasso_type_t type,
const char * val,
int len,
const char * conv );
```

lasso_typeAppendString()

This function appends a string to an existing stirng instance.

```
LCAPICALL osError lasso_typeAppendString(
lasso_request_t token,
lasso_type_t type,
const char * val,
int len );
```

lasso_typeAppendStringW()

Same as lasso_typeAppendString(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeAppendStringW(
lasso_request_t token,
lasso_type_t type,
const UChar * val,
int len );
```

lasso_typeSetInteger()

This function sets the data from a previously created integer instance.

```
LCAPICALL osError lasso_typeSetInteger(
lasso_request_t token,
lasso_type_t type,
osInt64 val );
```

lasso_typeSetDecimal()

This function sets the data from a previously created decimal instance.

```
LCAPICALL osError lasso_typeSetDecimal(
lasso_request_t token,
lasso_type_t type,
double val );
```

lasso_typeSetBoolean()

This function sets the data from a previously created boolean instance.

```
LCAPICALL osError lasso_typeSetBoolean(
lasso_request_t token,
lasso_type_t type,
bool val );
```

lasso_arrayGetSize()

This function gets the size of a previously created array instance.

```
LCAPICALL osError lasso_arrayGetSize(
lasso_request_t token,
lasso_type_t array,
int * len );
```

lasso_arrayGetElement()

This function gets an array element from a previously created array instance.

```
LCAPICALL osError lasso_arrayGetElement(
lasso_request_t token,
lasso_type_t array,
int index,
lasso_type_t * out );
```

lasso_arraySetElement()

This function sets an array element in a previously created array instance.

```
LCAPICALL osError lasso_arraySetElement(
lasso_request_t token,
lasso_type_t array,
int index,
lasso_type_t elem );
```

lasso_arrayRemoveElement()

This function removes an element from a previously created array instance.

```
LCAPICALL osError lasso_arrayRemoveElement(
lasso_request_t token,
lasso_type_t array,
int index );
```

lasso_mapGetSize()

This function gets the size of a previously created map instance.

```
LCAPICALL osError lasso_mapGetSize(
lasso_request_t token,
lasso_type_t mp,
int * len );
```

lasso_mapFindElement()

This function finds an element in a previously created map instance.

```
LCAPICALL osError lasso_mapFindElement(
lasso_request_t token,
lasso_type_t mp,
lasso_type_t key,
lasso_type_t * out );
```

lasso_mapGetNthElement()

This function gets an element from a previously created map instance using the element number.

```
LCAPICALL osError lasso_mapGetNthElement(
lasso_request_t token,
lasso_type_t mp,
int index,
lasso_type_t * outPair );
```

lasso_mapSetElement()

This function sets an element in a previously created map instance.

```
LCAPICALL osError lasso_mapSetElement(
lasso_request_t token,
lasso_type_t mp,
lasso_type_t key,
lasso_type_t value );
```

lasso_mapRemoveElement()

This function removes an element from a previously created map instance.

```
LCAPICALL osError lasso_mapRemoveElement(
lasso_request_t token,
lasso_type_t mp,
lasso_type_t key );
```

lasso_pairGetFirst()

This function gets the first element from a previously created pair instance.

```
LCAPICALL osError lasso_pairGetFirst(
lasso_request_t token,
lasso_type_t pr,
lasso_type_t * out );
```

lasso_pairGetSecond()

This function gets the second element from a previously created pair instance.

```
LCAPICALL osError lasso_pairGetSecond(
lasso_request_t token,
lasso_type_t pr,
lasso_type_t * out );
```

lasso_pairSetFirst()

This function sets the first element in a previously created pair instance.

```
LCAPICALL osError lasso_pairSetFirst(
lasso_request_t token,
lasso_type_t pr,
lasso_type_t frst );
```

lasso_pairSetSecond()

This function sets the second element in a previously created pair instance.

```
LCAPICALL osError lasso_pairSetSecond(
lasso_request_t token,
lasso_type_t pr,
lasso_type_t scnd );
```

lasso_typeGetMember()

This function is used to retrieve a member from a type instance. Members are searched by name with tag members searched first. Data members are searched if no tag member is found with the given name.

```
LCAPICALL osError lasso_typeGetMember(
lasso_request_t token,
lasso_type_t from,
const char * named,
lasso_type_t * out );
```

lasso_typeGetMemberW()

Same as lasso_typeGetMember(), but Uncode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeGetMemberW(
lasso_request_t token,
lasso_type_t from,
const UChar * named,
lasso_type_t * out );
```

lasso_typeGetProperties()

This function has two uses. If the targetType parameter is not NULL, it is used to get all data and tag members from a given type. They are returned as a pair of arrays in the outPair value. The first element of each pair is the map of data members for the type. The second element is the map of tag members. Each element in the array represents the members of each type inherited by the targetType.

If the targetType parameter is NULL, lasso_typeGetProperties will return an array containing the variable maps for the currently active request.

```
LCAPICALL osError lasso_typeGetProperties (
lasso_request_t token,
lasso_type_t targetType,
lasso_type_t * outPair );
```

lasso_typeGetName()

Retrieves the name of the target type.

```
LCAPICALL osError lasso_typeGetName(
lasso_request_t token,
lasso_type_t target,
auto_lasso_value_t * outName );
```

lasso_typeGetNameW()

Same as lasso_typeGetName(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeGetNameW(
lasso_request_t token,
lasso_type_t target,
auto_lasso_value_w_t * outName );
```

lasso_typeIsA()

This function returns the type of an object.

```
LCAPICALL osError lasso_typeIsA(
lasso_request_t token,
lasso_type_t target,
LP_TypeDesc type );
```

lasso_typeRunTag()

Used to execute a given tag. The tag can be run given a specific name and parameters, and the return value of the tag can be accessed. If the tag is a member tag, the instance of which it is a member can be passed using the final parameter. The params, returnValue, and optionalTarget parameters may all be NULL.

```
LCAPICALL osError lasso_typeRunTag (
lasso_request_t token,
const char * name,
lasso_type_t tagType,
int paramCount,
lasso_type_t * params, lasso_type_t * returnValue,
lasso_type_t optionalTarget );
```

lasso_typeRunTagW()

Same as lasso_typeRunTag(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeRunTagW (
lasso_request_t token,
const UChar * name,
lasso_type_t tagType,
int paramCount,
lasso_type_t * params,
lasso_type_t * returnValue,
lasso_type_t optionalTarget );
```

lasso_typeAssign()

This performs an assignment of one type to another. The result will be the same as if the following had been executed in LDML.

```
// #left_hand_side = #right_hand_side
LCAPICALL osError lasso_typeAssign(
lasso_request_t token,
lasso_type_t left_hand_side,
lasso_type_t right_hand_side );
```

lasso_typeStealValue()

This function transfers the data from one type to another type. Both types must be valid and pre-allocated. After the call, victim will still be valid, but will be of type null.

```
LCAPICALL osError lasso_typeStealValue(
lasso_request_t token,
lasso_type_t thief,
lasso_type_t victim );
```

lasso_handleExternalConversion()

Converts a Lasso type into single-byte or binary data using the specific encoding name. The default for all database, column, table names should be "iso8859-1".

```
LCAPICALL osError lasso_handleExternalConversion(
lasso_request_t token,
lasso_type_t instance,
const char * encoding,
auto_lasso_value_t * outVal);
```

lasso_handleInternalConversion()

Converts a single-byte or binary representation of a Lasso type back into an instance of that type.

```
LCAPICALL osError lasso_handleInternalConversion(
lasso_request_t token,
const char * src,
unsigned int srcLen,
const char * encoding,
LP_TypeDesc closestLassoType,
lasso_type_t * outType);
```

lasso_typeInheritFrom

This function changes the inheritence structure of a type. Sets newParent to be the new parent of child. Any parent that child currently has will be destroyed.

```
LCAPICALL osError lasso_typeInheritFrom(
lasso_request_t token,
lasso_type_t child,
lasso_type_t newParent );
```

Custom Types

lasso_typeAllocCustom()

This function is used within lasso_tag_funcs that were registered as being a type initializer (flag_typeInitializer). It initializes a blank custom type and sets the type's __type_name__ member to the provided value. The new type does not yet have a lineage and has no members added to it besides __type_name__. New data or tag members should be added using lasso_typeAddMember. The new custom type should be the return value of the type initializer. Any inherited members will be added to the type after the LCAPI call returns.

Warning: Do not call this unless you are in a type initializer. If you are not in a type initializer, the result will be a type that will never be fully initialized.

```
LCAPICALL osError lasso_typeAllocCustom(
lasso_request_t token,
lasso_type_t * outCustom,
const char * name );
```

lasso_typeAllocCustomW()

Same as lasso_typeAllocCustom(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeAllocCustomW(
lasso_request_t token,
lasso_type_t * outCustom,
const UChar * name );
```

lasso_typeSetCustomDtor()

Adds a function to be called on the custom type after onDestroy.

```
LCAPICALL osError lasso_typeSetCustomDtor(
lasso_type_t the_custom_type,
void (*dtor)(lasso_type_t the_custom_type) );
```

lasso_typeAddMember()

This is used to add new members to type instances. The member can be any sort of type including tags or other custom types.

```
LCAPICALL osError lasso_typeAddMember(
lasso_request_t token,
lasso_type_t to,
const char * named,
lasso_type_t member );
```

lasso_typeAddMemberW()

Same as lasso_typeAddMember(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeAddMemberW(
lasso_request_t token,
lasso_type_t to,
const UChar * named,
lasso_type_t member );
```

lasso_typeAllocFromProto()

Allocate a new type based on the given type. The given type's tag members will be referenced in the new type. No data members are added except for the typename member. Proto must be a custom type.

```
LCAPICALL osError lasso_typeAllocFromProto(
lasso_request_t token,
lasso_type_t proto,
lasso_type_t * out );
```

lasso_typeAllocOneOff()

Allocate a new type with the given name. The type does not have to have been registered as a type initializer or registered at all. The new type will have no tag or data members, but those may be added using the appropriate LCAPI call at any time. If no parent type is provided (a NULL pointer or empty string is passed in), type null will be the default. If a parent type is provided, it must have been a validly registered type initializer. onCreate will be called for the parent and beyond

```
LCAPICALL osError lasso_typeAllocOneOff(
lasso_request_t token,
const char * name,
const char * parentTypeName,
lasso_type_t * out );
```

lasso_typeAllocOneOffW()

Same as lasso_typeAllocOneOff(), but Unicode-compliant for Lasso Professional 7.

```
LCAPICALL osError lasso_typeAllocOneOffW(
lasso_request_t token,
const UChar * name,
const UChar * parentTypeName,
lasso_type_t * out );
```

Logging Functions

lasso_log()

Logs a message. The message goes to the prefered destination for the message level. Messages sent to a file are limited to 2048 bytes in length. Messages sent to the console are limited to 512 bytes in length. Messages sent to the database are limited a little less than 2048 bytes since the total length of the sql statement used to insert the message is limited to 2048 bytes. The msgLevel parameter must be one of the following: LOG_LEVEL_CRITICAL, LOG_LEVEL_WARNING, or LOG_LEVEL_DETAIL.

```
LCAPICALL osError lasso_log(
log_level_t msgLevel,
const char * fmt, ... );
```

MIME Headers

lasso_getResultHeader()

Retrieves current value of the result (HTTP) header. Part of the header that is returned to browsers is automatically built by Lasso, and can be modified or added to by LDML tags on the page. This function retrieves the current set of MIME headers that would be sent back to the browser if page processing were to stop now.

```
LCAPICALL osError lasso_getResultHeader(
lasso_request_t token,
auto_lasso_value_t * result );
```

lasso_getResultHeaderW()

Same as lasso_getResultHeader(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_getResultHeaderW(
lasso_request_t token,
auto_lasso_value_w_t * result );
```

lasso_setResultHeader()

Sets the result header, any data will be validated so as to be in the proper format.

```
LCAPICALL osError lasso_setResultHeader(
lasso_request_t token,
const char * header );
```

lasso_setResultHeaderW()

Same as lasso_setResultHeader(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_setResultHeaderW(
lasso_request_t token,
const UChar * header );
```

lasso_addResultHeader()

Simply appends the supplied data to the header, any data will be validated so as to be in the proper format.

```
LCAPICALL osError lasso_addResultHeader(
lasso_request_t token,
const char * data );
```

lasso_addResultHeaderW()

Same as lasso_addResultHeader(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_addResultHeaderW(
lasso_request_t token,
const UChar * data );
```

lasso_getCookieValue()

Retrieves a cookie value from the passed-in data sent by the client browser.

```
LCAPICALL osError lasso_getCookieValue(
lasso_request_t token,
const char * named,
auto_lasso_value_t * value );
```

lasso_getCookieValueW()

Same as lasso_getCookieValue(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_getCookieValueW(
lasso_request_t token,
const UChar * named,
auto_lasso_value_w_t * value );
```

Page Variables

lasso_getVariableCount()

Retrieves the number of array values which the named global variable has. Returns 1 if the global variable is not an array. Global variables are the same variables which you create in LDML statements, like [var: 'fred'=1234.56]. These variables last only as long as the current format file is executing; as soon as the hit gets sent back to the browser, these variables all get destroyed.

```
LCAPICALL osError lasso_getVariableCount(
lasso_request_t token,
const char * named,
int * count );
```

lasso_getVariableCountW()

Same as lasso_getVariableCount(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_getVariableCountW(
lasso_request_t token,
const UChar * named,
int * count );
```

lasso_getVariable()

Retrieves the value of the named global variable. If the global variable is an array, then the index specifies which array value to retrieve. If the global variable is not an array, then 0 is the only valid index. Array indices start at 0.

```
LCAPICALL osError lasso_getVariable(
lasso_request_t token,
const char * named,
int index,
auto_lasso_value_t * value );
```

lasso_getVariableW()

Same as lasso_getVariable(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_getVariableW(
lasso_request_t token,
const UChar * named,
int index,
auto_lasso_value_w_t * value );
```

lasso_getVariable2()

Retrieves the value of the named global variable while preserving the variable type.

```
LCAPICALL osError lasso_getVariable2(
lasso_request_t token,
const char * key,
lasso_type_t * value );
```

lasso_getVariable2W()

Same as lasso_getVariable2(), but Unicode-compiant for Lasso Professional 7.

```
LCAPICALL osError lasso_getVariable2W(
lasso_request_t token,
const UChar * key,
lasso_type_t * value );
```

lasso_setVariable()

Stores a new value into the named global variable. If the global variable is an array, then the 0-based index determines which array item to replace.

```
LCAPICALL osError lasso_setVariable(
lasso_request_t token,
const char * named,
const char * value,
int index );
```

lasso_setVariableW()

Same as lasso_setVariable(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_setVariableW(
lasso_request_t token,
const UChar * named,
const UChar * value,
int index );
```

lasso_setVariable2()

Stores a new global variable while preserving the type.

```
LCAPICALL osError lasso_setVariable2(
lasso_request_t token,
const char * key,
lasso_type_t value );
```

lasso_setVariable2W()

Same as lasso_setVariable2(), but Unicode-compiant for Lasso Professional 7.

```
LCAPICALL osError lasso_setVariable2W(
lasso_request_t token,
const UChar * key,
lasso_type_t value );
```

lasso_removeVariable()

Removes the specified variable (destroys it so it becomes undefined, as though it had never been created). If the named variable is an array, then you may pass in an index (0-based) to remove that array element. Once the array has 0 elements, then calling removeVariable on it will destroy the array itself.

```
LCAPICALL osError lasso_removeVariable(
lasso_request_t token,
const char * named,
int index );
```

lasso_removeVariableW()

Same as lasso_removeVariable(), but Unicode-compiant for Lasso Professional 7.

```
osError lasso_removeVariableW(
lasso_request_t token,
const UChar * named,
int index );
```

Interpret LDML tags

lasso_formatBuffer()

Formats the supplied buffer and put the resulting data in the data member of the auto_lasso_value_t. The buffer should consist of plain text and bracketed Lasso tags.

```
LCAPICALL osError lasso_formatBuffer(
lasso_request_t token,
const char * buffer,
auto_lasso_value_t * output );
```

Persistent Storage Tags

lasso_storeHasData()

Returns osErrNoErr if the data, specified by key, exists. The length of the stored data can be returned in the length parameter if you pass a PTR to an INT. You may pass NULL if for the length PTR if you don't want to retrieve the length of the stored data.

```
LCAPICALL osError lasso_storeHasData(
lasso_request_t token,
const char * key,
unsigned int * length );
```

lasso_storePutData()

Adds the data to Lasso's storage. Key is the unique identifier for the data.

```
LCAPICALL osError lasso_storePutData(
lasso_request_t token,
const char * key,
const void * data,
unsigned int length );
```

Administration

lasso_isAdministrator()

Returns osErrNoErr if the current user has administrator privileges. This is useful for doing module administration that only the administrator should be able to do.

```
LCAPICALL osError lasso_isAdministrator(
lasso_request_t token );
```

Data Source Functions

lasso_addDataSourceResult()

Sometimes Lasso Professional will ask your data source function to return some information, such as a list of database names or table names which your data source module controls. Your module will call this function once

for each name you add to the list, so if you have three database names you want to report back to Lasso Professional, you would call this function three times, once per database name.

```
LCAPICALL osError lasso_addDataSourceResult(
lasso_request_t token, const char * data );
```

lasso_getDataSourceName()

Use this function when you want to ask Lasso Professional what database is being operated on. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the database which Lasso Professional is asking you to search. It corresponds to the value of the parameter -Database=blah passed to inlines. Optionally, you can use the third (useHostDefault) parameter to determine whether the current database inherits its host default settings.

Note: Even though the name of the function is lasso_getDataSourceName, it really retrieves the database name. This is purely cosmetic, and just happens to be how the APIs were spelled when they were originally designed.

```
LCAPICALL osError lasso_getDataSourceName(
lasso_request_t token,
auto_lasso_value_t * t,
bool * useHostDefault,
auto_lasso_value_t * usernamepassword )
```

lasso_getDSConnection()

This function accesses the current datasource connection. May recurse the data source call by sending it the datasourceOpenConnection message

```
osError lasso_getDSConnection(
lasso_request_t token,
lasso_dsconnection_t * conn);
```

lasso_setDSConnection()

This function sets the current connection for the data source. May recurse to deliver the datasourceCloseConnection message if there is already a valid lasso_dbconnection_t set.

```
osError lasso_setDSConnection(
lasso_request_t token,
lasso_dsconnection_t conn);
```

lasso_getDataHost()

Use this function when you want to ask Lasso Professional 7 what database host is being operated on. On return, auto_lasso_value_t will contain the name and port of the database host.

```
LCAPICALL osError lasso_getDataHost(
lasso_request_t token,
auto_lasso_value_t * host,
auto_lasso_value_t * usernamepassword )
```

lasso_getDataHost2()

Same as lasso_getDataHost() but allows the usage of a host schema parameter for JDBC data sources.

```
LCAPICALL osError lasso_getDataHost2(
lasso_request_t token,
auto_lasso_value_t * host,
auto_lasso_value_t * schema,
auto_lasso_value_t * usernamepassword )
```

lasso_getSchemaName()

Use this function when you want to ask Lasso Professional what host schema is being operated on for a JDBC data source. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the schema which Lasso Professional is asking you to use for the search. It corresponds to the value of the parameter -Schema=blah passed to inlines.

```
LCAPICALL osError lasso_getSchemaName(
lasso_request_t token,
auto_lasso_value_t * t );
```

lasso_getTableName()

Use this function when you want to ask Lasso Professional what table is being operated on. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the table which Lasso Professional is asking you to search. It corresponds to the value of the parameter -Layout=blah or -Table=blah passed to inlines.

```
LCAPICALL osError lasso_getTableName(
lasso_request_t token,
auto_lasso_value_t * t );
```

lasso_getSkipRows()

You can ask Lasso Professional to tell you how many records should be skipped during a search by calling this function. It corresponds to the value of the -SkipRecords parameter in the inline search which is being executed at the moment your data source function is being called.

```
LCAPICALL osError lasso_getSkipRows(
lasso_request_t token,
int * recs );
```

lasso_getMaxRows()

You can ask Lasso Professional to tell you the maximum number of records to be returned during a search by calling this function. It corresponds to the value of the -MaxRecords parameter in the inline search which is being executed at the moment your data source function is being called.

```
LCAPICALL osError lasso_getMaxRows(
lasso_request_t token,
int * recs );
```

lasso_getPrimaryKeyColumn()

You can ask Lasso Professional to tell you which field is being used as the primary key. This value corresponds to the -KeyField parameter value used in the inline.

```
LCAPICALL osError lasso_getPrimaryKeyColumn(
lasso_request_t token,
auto_lasso_value_t * v );
```

lasso_getInputColumnCount()

Tells how many fields were sent as parameters to the inline. For instance, if an LDML programmer wants to append a new record to a table, and passes in name, address, city, state, zip with values for each field, then this function will return the number 5 to indicate that five fields were passed to the inline. You can then retrieve the values of each of these parameters by calling lasso_getInputColumn by index, once per field. This function is smart enough to ignore parameters which are not fields, such as -Database, -Layout, etc.

```
LCAPICALL osError lasso_getInputColumnCount(
lasso_request_t token,
int * count );
```

lasso_getInputColumn()

Retrieve the name and value of field data parameters from the inline, starting at index zero. If five fields were entered into the inline, then you can retrieve each of their names and values by calling this function five times, once per field.

```
[Inline: -Database='MyDatabase', -Table='Main', 'MyFirstField'='Bill',
'MySecondField'='Ted', -Search]
```

In the above example, calling lasso_GetInputColumn(token, 0, &v) will fill the v variable with v.name=MyFirstField, v.data=Bill. Notice it is smart enough to ignore well-known parameters such as -Table, thus only retrieving field information.

```
LCAPICALL osError lasso_getInputColumn(
lasso_request_t token,
int index,
auto_lasso_value_t * v );
```

lasso_getSortColumnCount()

Analogous to lasso_GetInputColumnCount(), this function retrieves the number of sort columns which were specified in the inline code. It basically counts how many -SortField parameters were passed. You can use this count to tell you how many times to enumerate through calls to lasso_getSortColumn().

```
LCAPICALL osError lasso_getSortColumnCount(
lasso_request_t token,
int * count );
```

lasso_getSortColumn()

Analogous to lasso_getInputColumn(), this function retrieves the names of sort parameters, starting at index zero. After calling this, the value of variable v.data will contain a c-string with the name of the sort field.

```
LCAPICALL osError lasso_getSortColumn(
lasso_request_t token,
int index,
auto_lasso_value_t * v );
```

lasso_getRowID()

Retrieves the current specified record ID (FileMaker Pro only).

```
LCAPICALL osError lasso_getRowID(
lasso_request_t token,
int * id );
```

lasso_setRowID()

Sets the record ID of the added record. After your custom LCAPI data source finishes adding a record to a database, it can call this function to let the caller know what the unique record ID of the added record was.

In FileMaker, this record ID is a standard feature of all records in its tables. In MySQL, this value is 0 unless there exists an AUTO_INCREMENT column. Results are not guaranteed for all database server software.

```
LCAPICALL osError lasso_setRowID(
lasso_request_t token,
int * id );
```

lasso_findInputColumn()

Analogous to lasso_getInputColumn(), except that it searches by name instead of index. If you already know the name of a field parameter you're interested in, then you can ask for the value of that parameter which was passed into the inline.

```
[Inline: -Database='MyDatabase', -Table='Main', 'MyFirstField'='Bill',
'MySecondField'='Ted', -Search]
```

In the example above, calling lasso_findInputColumn(token, MySecondField, &v) will fill the v variable's data member with v.data=Ted.

```
LCAPICALL osError lasso_findInputColumn(
lasso_request_t token,
const char * name,
auto_lasso_value_t * value );
```

lasso_findInputColumnW()

Same as lasso_findInputColumn(), but Unicode-compliant for Lasso Professional 7.

```
osError lasso_findInputColumnW(
lasso_request_t token,
const UChar * name,
auto_lasso_value_t * value );
```

lasso_getLogicalOp()

Call this to retrieve the logical operator (kLassoAND, kLassoOR) which was passed to this inline. It corresponds to the value of -LogicalOperator passed into the inline. This function simply retrieves a single logical operator parameter. For more complex logical operations, with multiple operators, you will have to design a convention whereby you name your input fields in some unique way, and then retrieve those custom logical operators

using the lasso_getInputColumn() function in a particular order that matches your convention.

```
LCAPICALL osError lasso_getLogicalOp(
lasso_request_t token,
LP_TypeDesc * op );
```

lasso_getReturnColumnCount()

You can ask Lasso Professional to tell you how many columns (fields) are expected to be returned from a search operation. This counts how many -ReturnField parameters were encountered.

```
LCAPICALL osError lasso_getReturnColumnCount(
lasso_request_t token,
int * count );
```

lasso_getReturnColumn()

Once you know how many return columns are expected (from lasso_getReturnColumnCount), then you can enumerate through them to get their fieldnames. Use this information to retrieve field data from your database table, and populate the result rows when asked to perform a search operation.

```
LCAPICALL osError lasso_getReturnColumn(
lasso_request_t token,
int num,
auto_lasso_value_t * v );
```

lasso_addColumnInfo()

In order to return a row of data from your data source (perhaps as a result of a search), you must first indicate what the structure of the table columns is. Call this function for as many table columns as your database has, providing the fieldname, true/false if nulls are OK in this field, the field type (numeric, string, date, etc), and field protection (readonly, writeable, etc).

```
LCAPICALL osError lasso_addColumnInfo(
lasso_request_t token,
const char * name,
int nullOK,
LP_TypeDesc type,
LP_TypeDesc protection );
```

lasso_addResultRow()

Call this function once per row of records you want to return (perhaps from a search operation). You must construct an array of pointers that

contains pointers to each of your fields (binary data is OK), and provide an array of sizes (field data lengths) that corresponds to the length of each field. Because you are providing length information, there is no need to null-terminate c-strings.

```
LCAPICALL osError lasso_addResultRow(
lasso_request_t token,
const char ** columns,
unsigned int *sizes,
int numColumns );
```

lasso_setNumRowsFound()

Corresponds to [Found_Count] in LDML. Call this when you know how many records your data source is going to return, and make sure you call lasso_addResultRow() this many times in order to populate the rows.

```
LCAPICALL osError lasso_setNumRowsFound(
lasso_request_t token,
int num );
```

Semaphores

lasso_createSem()

Creates a named semaphore sufficient for synchronizing multithreaded operations. Make sure you delete these when you're done with them. The Lasso Connector for MySQL example creates one of these at init time, and destroys it at terminate time.

```
LCAPICALL osError lasso_createSem(
lasso_request_t token,
const char * name );
```

lasso_destroySem()

Destroys a named semaphore which had been created earlier by the lasso_createSem() function.

```
LCAPICALL osError lasso_destroySem(
lasso_request_t token,
const char * name );
```

lasso_acquireSem()

Attempts to acquire a lock on a semaphore; waits until the owning thread has released the semaphore before acquiring the lock and continuing execution.

```
LCAPICALL osError lasso_acquireSem(
lasso_request_t token,
const char * name );
```

lasso_releaseSem()

When you are done with a semaphore whose lock you've acquired, call lasso_releaseSem() to release it so other threads waiting for this semaphore can continue execution.

```
LCAPICALL osError lasso_releaseSem(
lasso_request_t token,
const char * name );
```

LCAPI Data Type Reference

lasso_request_t

Opaque data structure which must be passed to/from all calls in LCAPI. This token gives LCAPI the state information it needs to distinguish between different simultaneous hits to Lasso Service.

lasso_type_t

Opaque data structure which represents data of any type. A lasso_type_t can represent any of the native internal types or any custom type written in LDML, LCAPI or LJAPI.

auto_lasso_value_t

Name/Value pair used to retrieve text from LCAPI calls. General-purpose pair of strings which destroys itself automatically. Has name and data member variables which can be treated like null-terminated c-strings.

auto_lasso_value_w_t

Same as auto_lasso_value_t, but used for Unicode text strings.

LP_TypeDesc

Used for both data types and field protection types. You pass these values when you specify the field/column in your table.

Parameters	Description
typeBlob	Binary Large Object.
typeChar	Null-terminated C-String.
typeDecimal	Floating point number.
typeLongInteger	Long long (8 bytes) signed integer.
typeBoolean	True/false, 1/0.
typeDateTime	String representing a date and time together.
typeArray	Array data type.
typeMap	Map data type.
typeTag	Tag data type.
typeReference	A type that is a reference to another type instance.
typePair	Pair data type.
typeCustom	Any custom data type.
kProtectionNone	This field has no data protection.
kProtectionReadOnly	This field cannot be modified.

Frequently Asked Questions

How do I install my custom tag?

Once you've compiled your tag module, you'll need to move the module to your installed Lasso Professional LassoModules folder, and then restart Lasso Service. Step-by-step instructions are available in the *Getting Started* section.

How do I return text from my custom tag?

Use either lasso_outputTagData() or lasso_outputTagDataF() to send text back to the output HTML page. Any text you send out with these functions will be streamed into the output HTML page send back to the browser, and will replace (substitute) the raw [xxx] tag text, just like any other LDML tag would do.

How do I debug my custom tag?

You can set breakpoints in your code and attach your module DLL to Lasso Service. Read the section on debugging LCAPI modules.

How do I get parameters that were passed into my tag?

Most of the parameters passed into your custom tag can be retrieved using the lasso_getTagParam() and lasso_findTagParam() parameter info APIs.

lasso_getTagParam() retrieves parameters by index and lasso_findTagParam() retrieves them by name.

How do I get the value of unnamed parameters passed into my tag?

While there is no direct way to get unnamed parameters (how do you know what name to ask for?), you can enumerate through all the parameters by index, and then pick out the ones which do not have names. If, after retrieving a parameter, you discover that its data member is an empty string, then that means it is an unnamed parameter, and you can get its value from the name member. An example of this is in the substitution tag tutorial.

What's an auto_lasso_value_t and how do I use it?

It's a data structure which contains both a name and a value (a name/value pair). Many LCAPI APIs fill in this structure for you, and you can access the name and data members directly as null-terminated C-strings.

How do I access variables from the LDML page I'm in?

You may need to get or even create LDML variables (the same variables that an LDML programmer makes when using the [var: 'fred'=12] variable syntax in a format file) from within your LCAPI module. You can retrieve a global variable, as long as it has already been assigned before your custom substitution tag is executed, by calling lasso_getVariable() with the variable's name.

How do I return fatal and non-fatal error codes?

It is very important that your substitution tag return an error code of 0 if nothing fatal happened. An example of a fatal error would be a missing required parameter, for instance. If you encounter a fatal error, then return a non-zero result code from your tag function, and the Lasso will stop processing the page at that point, and display an error page.

How do I write code that will compile easily across multiple operating systems?

While we cannot provide a complete cross-platform programming tutorial for you here, we can at least provide some guidance. The simplest way to make sure things compile across platforms is to make sure you use standard library functions (from stdio.h and stdlib.h) as much as possible: functions like strcpy(), malloc(), and strcmp() are always available on all platforms. Also note that Unix platforms are case-sensitive, so when you #include files, just make sure you keep the case the same as the file on disk. Finally, stay away from platform-specific functions, such as Windows APIs, which most often are not available on Unix platforms. Take a look at our Unix make-

files which are provided with the sample projects: notice the same source code is used for Windows, and all source files are saved with DOS-style cr/lf linebreaks so as not to confuse the Windows compilers. As a last resort, you can use #ifdef to show/hide portions of source code which are platform-specific.

7

Chapter 7
Lasso Connector Protocol

This chapter documents Lasso Connector Protocol (LCP) and describes how to develop Lasso Web server connectors.

- *Overview* introduces Lasso Connector Protocol.
- *Requirements* includes platform specific development environment details.
- *Lasso Web Server Connectors* introduces the theory of operation behind creating Lasso Web server connectors using LCP.
- *Building Web Server Connectors* is a quick start guide to building the provided samples.
- *Lasso Connector Operation* describes the theory and operation behind building Lasso Web server connectors.
- *Lasso Connector Tutorial* provides a step-by-step walk-through of building a sample Web server connector.
- *Lasso Connector Protocol Reference* provides a reference of all commands and parameters used in LCP.

Overview

Lasso Web server connectors are small modules written specifically for a particular brand of Web server. Lasso Professional 7 initially includes connectors for Microsoft IIS (Intel architecture), Apple Mac OS X's Apache (PowerPC architecture), and 4D WebSTAR Server Suite V for Mac OS X. A connector for Red Hat Apache (Intel architecture) is also available.

The purpose of Lasso Connector Protocol (LCP) is to provide an efficient and platform-independent way of communication between a Lasso

connector (client) and Lasso Service (server). Included are sample projects which give you full source code to the Web server connectors which ship with Lasso (e.g. Lasso Connector for IIS and Lasso Connector for Apache). Web servers you might want to develop connectors for include: Netscape Enterprise Server, O'Reilly WebSite, Zeus, Cobalt RAQ Apache, Solaris SPARC Apache, Microsoft IIS (Alpha chipset), and WebSTAR Server Suite 4. Blue World encourages developers to create and distribute new Web server connectors in order to give Lasso developers as many choices as possible for developing Lasso-based data-driven Web sites.

Requirements

In order to write your own Lasso Web server connector in C or C++, you will need the following:

Windows:

- Microsoft Windows 2000 or Windows XP Professional
- Microsoft Visual C++ 7.
- Windows Lasso Professional version 7.0 or higher.

Mac OS:

- Mac OS X 10.2 with GNU C++ compiler and linker (Dev Tools) installed.
- Mac OS X Lasso Professional version 7.0 or higher.

Lasso Web Server Connectors

All modern Web servers have some form of suffix mapping, where they re-route HTTP requests to various modules based on their file suffix (e.g. .lasso). Modules have different names depending on which Web server you're using: ISAPI DLL, Apache Module, W*API plugin, etc. Once the Web server calls the Lasso Web server connector, it is the job of the Lasso Web server connector to collect all the information from a particular request, and pass it all along to a Lasso Service application that it's set up to talk to. Then it waits for Lasso Service to finish processing the request, and receives back some MIME headers and HTML body text. At this point it's the connector's job to pass the text back to the Web server, which in turn sends it back out to the requesting browser. All communication is via TCP/IP, so the Web server connector and Lasso Service may be on separate machines with different architectures.

Lasso Web server connectors also have another job, which is to decode and write out HTTP-upload files. As you examine the sample source code, you'll see that it interprets the incoming POST arguments, writes out temporary files, and passes a special list of filename arguments through to Lasso Service on the other side of the TCP connection.

Note: Only a single Lasso Web server connector can connect with Lasso Service at a time in Lasso Professional 7.

Getting Started

This section provides a walk-through for building a custom Web server connector in Windows 2000 and Mac OS X.

To build a sample Web server connector in Windows 2000:

1. Browse to the Lasso Professional 7\Documentation\4-ExtendingLasso\LCP folder on the hard drive.
2. In the LCP folder, open the folder that corresponds to you operating system (e.g. Win2000, OSXApache).
3. Double-click the L6isapi.dsp project file — you need Microsoft Visual C++ 6 in order to open it.
4. Choose Build/Rebuild All to compile and make the Lasso6Isapi.DLL.
5. After building, Debug and Release folders will have been created inside your L6isapi project folder.
6. Open IIS Admin and shut down IIS (so that any previous Lasso6Isapi.DLL files will not be held open inside IIS).
7. Open the L6isapi/Debug folder and drag Lasso6Isapi.DLL into your WinNT/System32 folder.
8. Restart IIS using the Services menu in the windows Control Panel.
9. Assuming you already have Lasso installed on this machine, your suffix mappings should all work, and Lasso should function just as it did before.
10. Use a Web browser to view http://your.Web.server/Lasso/ and make sure the .lasso suffix mapping is still working.

To build a sample Web server connector in Mac OS X:

1. Open a Terminal window.
2. Change the current folder to the Documentation folder by entering the following:

```
cd /Applications/Lasso\ Professional\ 7/Documentation/4-ExtendingLasso/LCP/
OSXApache/'
```

3 Build the sample project using the provided makefile. You must be logged in as the root user to run this command.

```
make
```

4 After building, a Mac OS X dynamic library file will be in the current folder: LassoConnectorforApache.so. This is the module you'll install into the ApacheModules folder.

5 Copy the newly-created module to the LassoModules folder by entering the following:

```
cp LassoConnectorforApache.so /usr/libexec/httpd/
```

6 Logged in as root user, restart apache so it loads the new module.

```
su
<enter root password here>
apachectl restart
```

Assuming you already have Lasso installed on this machine, your suffix mappings should all work, and Lasso should function just as it did before.

7 In a Web browser, go to http://your.Web.server/Lasso/ and try a few things to make sure the .lasso suffix mapping is still working.

Debugging

This section describes the procedures for debugging a Lasso Web server connector built in Windows 2000 or Mac OS X.

Windows 2000

Detailed instructions for debugging Windows Internet Server extension DLL files can be found at the following URL:

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/
_mfcnotes_tn063.asp
```

Mac OS X

The debugging procedures for Lasso Web server connectors in Mac OS X are identical to the debugging procedures for LCAPI data source connectors in Mac OS X, with the exception of the files names and paths. See *Chapter 4: C/C++ API > Debugging* for detailed instructions.

Lasso Connector Operation

Communication between the Lasso Web server connector (client) and Lasso Service (server) is achieved by means of exchanging messages via a regular TCP/IP socket on port 14550. A typical session is initiated by a client and consists of the following steps:

1 Connect to Lasso Service host on port 14550.

2 Send the open request command.

3 Handle requests sent back from Lasso.

4 Repeat previous step until the close request command is received.

5 Close the connection.

All messages to and from Lasso Service begin with the LPCommandBlock structure, and are optionally followed by an arbitrary number of data bytes if needed. The LPCommandBlock structure is defined as follows:

```
typedef enum          LPCommand;
typedef int           LPRequestID;
typedef unsigned int  LPSequenceNum;
typedef struct        LPCommandBlock
{
  LPCommand           fCmd;
  int                 fResultCode;
  unsigned int        fDataSize;
};
```

The meaning of each LPCommandBlock structure member is explained in the following table.

Table 1: LPCommandBlock Structure Members

Command	Description
fCmd	The command. For a list of currently defined commands see the Command Reference at the end of this chapter.
fResultCode	The result of the command. Used if the command is a reply.
fDataSize	The size of the additional command-specific data to follow (may be zero).

Lasso Connector Tutorial

This section provides a walk-through of building an example Lasso connector for Mac OS X Apache to show how LCP features are used.

This code will be most similar to the sample LassoApache project, so in order to build this code, copy the LassoApache project folder and edit the project files inside it.

Lasso Connector Module Code

Shown below is the code for the Apache module. Line numbers are provided to the left of each line of code, and are referenced in the *Lasso Connector Module Walk-Through* section.

Note: The line numbers shown refer to the line numbers of the code in the actual file being created, not as shown in this page. Some single lines of code may carry into two or more lines as shown on this page.

Lasso Connector Module Code

```
1  int sock = -1;
2  module MODULE_VAR_EXPORT lasso_module =
3  {
4    STANDARD_MODULE_STUFF,
5    NULL, /* initializer */
6    NULL, /* dir config creater */
7    NULL, /* dir merger ensure strictness */
8    NULL, /* server config */
9    NULL, /* merge server config */
10   NULL, /* command table */
11   handlers, /* handlers */
12   NULL, /* filename translation */
13   NULL, /* check_user_id */
14   NULL, /* check auth */
15   NULL, /* check access */
16   NULL, /* type_checker */
17   NULL, /* fixups */
18   NULL, /* logger */
19   NULL, /* header parser */
20   child_init_handler, /* child_init */
21   child_term_handler, /* child_exit */
22   NULL /* post read-request */
23 };
24 static const handler_rec handlers[] =
25 {
26   { "lasso-handler", lasso_handler },
27   { NULL }
```

```
28 };
29 void child_init_handler(server_rec*, pool*)
30 {
31 sock = open_connection();
32 }
33 int open_connection()
34 {
35   int tries = 0;
36   go:
37   int res = 0;
38   int s = socket(AF_INET, SOCK_STREAM, 0);
39   sockaddr_in addr;
40   char buff[256];
41   char server[128] = "127.0.0.1";
42   char port[128] = "14550";
43   addr.sin_family = AF_INET;
44   addr.sin_port = htons((short)atoi(port));
45   addr.sin_addr.s_addr = inet_addr(server);
46   addr.sin_zero[0] = addr.sin_zero[1] = addr.sin_zero[2] = addr.sin_zero[3] = 0;
47   res = connect(s, (sockaddr*)&addr, sizeof(sockaddr_in));
48 if ( res == 0 )
49 {
50   return s;
51 }
52 if ( s != -1 )
53 {
54   close(s);
55   s = -1;
56 }
57   return -1;
58 }
59   void child_term_handler(server_rec*, pool*)
60 {
61   close(sock);
62   sock = -1;
63 }
64   int get_post( int sock, request_rec * r, POSTReader * reader )
65 {
66   if ( ap_setup_client_block(r, REQUEST_CHUNKED_ERROR) == OK )
67   {
68     if ( ap_should_client_block(r) == 1)
69       {
70       char * buffer = new char[r->remaining+1];
71       int readLen = 0;
72       int toRead = r->remaining;
73       int readSize = r->remaining;
74       ap_hard_timeout("rpPostKeyword", r);
75       while ( (readLen = ap_get_client_block(r, buffer, readSize)) > 0 )
```

```
76          {
77            buffer[readLen] = '\0';
78            reader->AddToBuffer(buffer, readLen);
79            ap_reset_timeout(r);
80            toRead -= readLen;
81            readSize = toRead < DEFAULT_BUFFER ? toRead : DEFAULT_BUFFER;
82            }
83          ap_kill_timeout(r);
84          delete [] buffer;
85        }
86      }
87  return 1;
88  }
89    int lasso_handler (request_rec *r)
90  {
91    int res = 0;
92    LPCommandBlock block;
93    char * data = NULL;
94    int reqOpen = 1;
95    unsigned int timeout = 300;
96    int openTries = 0;
97    int read = 0;
98    POSTReader reader(ap_table_get(r->headers_in, "Content-Type"));
99    get_post(sock, r, &reader);
100   openCon:
101   if ( sock == -1 )
102   {
103     sock = open_connection(gChildPool, r->server);
104   }
105   if ( sock == -1 )
106   return SERVER_ERROR;
107   /* start reading command blocks from Lasso */
108   while ( reqOpen == 1 && (read = read_block(sock, &block, &data, &timeout)) == 1 )
109   {
110     switch( block.fCmd )
111     {
112       case cmdGetParam:
113         {
114           char * myData = data;
115           int numParams = 0;
116           // scan to see how many params
117           for (unsigned int pos = 0; pos < block.fDataSize;)
118           {
119             ++numParams;
120             pos += sizeof(RequestParamKeyword);
121             int dSize = *(int*)(data+pos);
122             pos += (ntohl(dSize)+sizeof(int));
123           }
```

```
124         block.fCmd = cmdGetParamRep;
125         block.fDataSize = 0;
126         block.fResultCode = 0;
127         struct ptr_size
128         {
129         char * data;
130         int size;
131       };
132       ptr_size * rec = (ptr_size*)malloc(numParams*sizeof(ptr_size));
133       for ( int x = 0; x < numParams; ++x )
134       {
135         RequestParamKeyword word = ntohl(*(RequestParamKeyword*)myData);
136         int dLen = ntohl(*(int*)(myData+sizeof(RequestParamKeyword)));
137         char * pData = NULL;
138         int dSize = 0;
139         myData += (sizeof(RequestParamKeyword) + sizeof(int));
140         if ( word == rpPostKeyword )
141         {
142           const std::string * str = reader.GetPostArgs();
143           if ( str != NULL )
144           {
145             pData = (char*)malloc(str->size()+1);
146             dSize = str->size()+1;
147             memcpy(pData, str->c_str(), str->size());
148             pData[str->size()] = '\0';
149           }
150         }
151         else
152         {
153           int res = get_param(r, word, dLen > 0 ? myData : NULL, &pData, &dSize);
154           if ( res < 0 )
155           block.fResultCode = res;
156         }
157         rec[x].data = pData;
158         rec[x].size = dSize;
159         block.fDataSize += (dSize + sizeof(int));
160       }
161       /* alloc a buffer for the data portion of the block*/
162       char * pData = (char*)malloc(block.fDataSize);
163       char * ppData = pData;
164       for ( int x = 0; x < numParams; ++x )
165       {
166         int mSize = htonl(rec[x].size);
167         memcpy(ppData, &mSize, sizeof(int));
168         memcpy(ppData+sizeof(int), rec[x].data, rec[x].size);
169         ppData += (sizeof(int) + rec[x].size);
170         free(rec[x].data);
171       }
```

```
172             res = send_block(sock, &block, pData);
173             // dispose rec and each pointer within!
174             free(rec);
175             free(pData);
176           }
177           break;
178           case cmdPushData:
179           ap_rwrite(data, block.fDataSize, r);
180           break;
181           case cmdCloseReq:
182           reqOpen = 0;
183           break;
184           default:
185           break;
186         }
187         free(data);
188         data = NULL;
189       }
190       if ( read != 1 ) // some error
191       {
192         close(sock);
193         sock = -1;
194         if ( ++openTries < 6 )
195         goto openCon;
196         return SERVER_ERROR;
197       }
198     return OK;
199     }
```

Lasso Connector Module Walk-Through

This section provides a step-by-step walk through for building a Lasso Web server connector for Mac OS X Apache.

To build a Lasso connector:

1 Define a global variable to hold the reference to the socket used by the Apache module for communicating with Lasso Service.

```
1  int sock = -1;
```

2 Define which functions in our module will be called by the Apache Web server for processing HTTP requests.

```
2  module MODULE_VAR_EXPORT lasso_module =
3  {
4     STANDARD_MODULE_STUFF,
5     NULL,                    /* initializer */
6     NULL,                    /* dir config creater */
7     NULL,                    /* dir merger ensure strictness */
```

```
 8      NULL,                   /* server config */
 9      NULL,                   /* merge server config */
10     NULL,                            /* command table */
11     handlers,               /* handlers */
12     NULL,                            /* filename translation */
13     NULL,                            /* check_user_id */
14     NULL,                            /* check auth */
15     NULL,                            /* check access */
16     NULL,                            /* type_checker */
17     NULL,                            /* fixups */
18     NULL,                            /* logger */
19     NULL,                            /* header parser */
20     child_init_handler,     /* child_init */
21     child_term_handler,     /* child_exit */
22     NULL                    /* post read-request */
23 };
```

3 The Apache API allows the module to define any number of content handlers. Our module implements a single handler named lasso-handler.

```
24 static const handler_rec handlers[] =
25 {
26     { "lasso-handler", lasso_handler },
27     { NULL }
28 };
```

4 Every time the Apache server forwards a new child process, it calls the child_init_handler() function as specified by lasso_module structure. At this point, we open a connection to Lasso Service and prepare for processing the request.

```
29 void child_init_handler(server_rec*, pool*)
30 {
31  sock = open_connection();
32 }
```

5 In order to open a connection, we need to create a socket of type SOCK_STREAM. A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams, similar to pipes.

```
33 int open_connection()
34 {
35  int tries = 0;
36 go:
37   int res = 0;
38   int s = socket(AF_INET, SOCK_STREAM, 0);
```

6 Next, specify the host address and port of the computer Lasso Service is running on. In our sample code, both server addresses (127.0.0.1) and port (14550) are hard-coded. For an example of how to read these settings from a file, see the LassoConnectorforApache.c source.

```
39    sockaddr_in addr;
40    char buff[256];
41    char server[128] = "127.0.0.1";
42    char port[128] = "14550";
43    addr.sin_family = AF_INET;
44    addr.sin_port = htons((short)atoi(port));
45    addr.sin_addr.s_addr = inet_addr(server);
46    addr.sin_zero[0] = addr.sin_zero[1] = addr.sin_zero[2] = addr.sin_zero[3] = 0;
```

7 Finally, call the connect() method to open a connection to Lasso Service.

```
47    res = connect(s, (sockaddr*)&addr, sizeof(sockaddr_in));
```

8 If a connection has been established successfully, then return the socket ID. Otherwise, close the connection before returning.

Note: LassoProtoUtils.cpp includes various utility routines, including two important methods: send_block() and read_block(). These methods are used extensively in the sample code for sending/receiving various LCP messages:

```
48      if ( res == 0 )
49      {
50        return s;
51      }
52      if ( s != -1 )
53      {
54      close(s);
55      s = -1;
56      }
57  return -1;
58  }
```

9 After processing the request, close the connection. Apache calls this as the child process is terminated.

```
59  void child_term_handler(server_rec*, pool*)
60  {
61   close(sock);
62   sock = -1;
63  }
```

10 Define a user function for handling POST arguments. While most of the work of decoding and saving uploaded file data on the disk is being handled by the POSTReader class (defined in the POSTReader.cpp file, and included with Lasso Professional 7), you still need to provide a simple function for reading the POST data.

```
64  int get_post( int sock, request_rec * r, POSTReader * reader )
65   {
66   if ( ap_setup_client_block(r, REQUEST_CHUNKED_ERROR) == OK )
67    {
```

```
68      if ( ap_should_client_block(r) == 1)
69      {
70        char * buffer = new char[r->remaining+1];
71        int readLen = 0;
72        int toRead = r->remaining;
73        int readSize = r->remaining;
74       ap_hard_timeout("rpPostKeyword", r);
75       while ( (readLen = ap_get_client_block(r, buffer, readSize)) > 0 )
76       {
77         buffer[readLen] = '\0';
78         reader->AddToBuffer(buffer, readLen);
79         ap_reset_timeout(r);
80         toRead -= readLen;
81          readSize = toRead < DEFAULT_BUFFER ? toRead : DEFAULT_BUFFER;
82       }
83       ap_kill_timeout(r);
84      delete [] buffer;
85    }
86  }
87 return 1;
88 }
```

11 The actual work of handling the HTTP request is done in the body of the lasso_handler() method, as specified by the lasso_module structure. First, define variables to hold the command block, status code, and any additional parameters.

```
89 int lasso_handler (request_rec *r)
90 {
91   int res = 0;
92  LPCommandBlock block;
93   char * data = NULL;
94   int reqOpen = 1;
95   unsigned int timeout = 300;
96   int openTries = 0;
97   int read = 0;
```

12 Initialize the POSTReader object, used for retrieving any POST arguments that may have been submitted as part of the HTTP request:

```
98    POSTReader reader(ap_table_get(r->headers_in, "Content-Type"));
99    get_post(sock, r, &reader);
```

13 Make sure the connection is still available, and open a new one, if necessary.

```
100   openCon:
101   if ( sock == -1 )
102   {
103     sock = open_connection(gChildPool, r->server);
104   }
```

```
105     if ( sock == -1 )
106     return SERVER_ERROR;
```

14 Process any further messages sent by Lasso Service in a loop, waiting for the "close request" command (cmdCloseReq), or until no more data is available for reading.

Note: LassoProtoUtils.cpp includes various utility routines, including two important methods: send_block() and read_block(). These methods are used extensively in the sample code for sending/receiving various LCP messages.

```
107     /* start reading command blocks from Lasso */
108     while ( reqOpen == 1 && (read = read_block(sock, &block, &data, &timeout))
    == 1 )
109     {
110       switch( block.fCmd )
111       {
112         case cmdGetParam:
113          {
```

15 In addition to the "close request" command, the Lasso connector should be prepared to handle three different types of commands, including the request to obtain a specific value of a named server variable or HTTP request parameter (cmdGetParam).

Every HTTP request may have a number of properties associated with it, such as a client IP address, HTTP headers sent by the Web browser, the URL of the requested file, etc. For a full list of the named parameters which could be sent by Lasso Service, see the RequestParams.h file.

```
114             char * myData = data;
115             int numParams = 0;
116             // scan to see how many params
117             for (unsigned int pos = 0; pos < block.fDataSize;)
118             {
119               ++numParams;
120               pos += sizeof(RequestParamKeyword);
121               int dSize = *(int*)(data+pos);
122               pos += (ntohl(dSize)+sizeof(int));
123             }
124             block.fCmd = cmdGetParamRep;
125             block.fDataSize = 0;
126             block.fResultCode = 0;
127             struct ptr_size
128             {
129               char * data;
130               int size;
131             };
132             ptr_size * rec = (ptr_size*)malloc(numParams*sizeof(ptr_size));
```

```
133             for ( int x = 0; x < numParams; ++x )
134             {
135               RequestParamKeyword word = ntohl(*(RequestParamKeyword*)myDat
    a);
136               int dLen = ntohl(*(int*)(myData+sizeof(RequestParamKeyword)));
137               char * pData = NULL;
138               int dSize = 0;
139               myData += (sizeof(RequestParamKeyword) + sizeof(int));
```

16 Unlike the rest of the named parameters, POST argument data (rpPost-Keyword parameter) requests require special handling on behalf of the POSTReader class.

```
140               if ( word == rpPostKeyword )
141               {
142               const std::string * str = reader.GetPostArgs();
143               if ( str != NULL )
144               {
145               pData = (char*)malloc(str->size()+1);
146               dSize = str->size()+1;
147               memcpy(pData, str->c_str(), str->size());
148               pData[str->size()] = '\0';
149             }
150           }
```

17 The rest of the named parameters can be easily handled by a single function. For a complete code listing of the get_param() function, see the LassoConnectorforApache.c source.

```
151         else
152         {
153             int res = get_param(r, word, dLen > 0 ? myData : NULL, &pData,
    &dSize);
154             if ( res < 0 )
155             block.fResultCode = res;
156         }
157          rec[x].data = pData;
158          rec[x].size = dSize;
159          block.fDataSize += (dSize + sizeof(int));
160       }
161       /* alloc a buffer for the data portion of the block*/
162       char * pData = (char*)malloc(block.fDataSize);
163       char * ppData = pData;
164        for ( int x = 0; x < numParams; ++x )
165        {
166          int mSize = htonl(rec[x].size);
167          memcpy(ppData, &mSize, sizeof(int));
168          memcpy(ppData+sizeof(int), rec[x].data, rec[x].size);
169           ppData += (sizeof(int) + rec[x].size);
170          free(rec[x].data);
```

```
171       }
```

18 Once the value of the requested named parameter has been obtained, send the result back to Lasso Service.

```
172       res = send_block(sock, &block, pData);
173       // dispose rec and each pointer within!
174       free(rec);
175       free(pData);
176     }
177     break;
```

19 Additional commands may include a signal of an unexpected error that occurred during the use of the protocol (cmdProtoErr), and a request to push partially-processed data back to the Web browser (cmdPushData).

```
178     case cmdPushData:
179     ap_rwrite(data, block.fDataSize, r);
180     break;
```

20 Continue to process incoming messages until the cmdCloseReq command is received from Lasso Service, signalling that there is no more data to be sent to the Web browser.

```
181     case cmdCloseReq:
182     reqOpen = 0;
183     break;
184     default:
185     break;
186     }
```

21 Finally, free up any memory previously allocated by the read_block() method during reading the message data.

```
187    free(data);
188    data = NULL;
189  }
```

22 If any errors occured during data transmission/reception, attempt to reconnect with the Lasso Service. Otherwise, inform the server whether the connector succeeded in handling the HTTP request.

```
190  if ( read != 1 ) // some error
191  {
192       close(sock);
193       sock = -1;
194       if ( ++openTries < 6 )
195       goto openCon;
196       return SERVER_ERROR;
197    }
198  return OK;
199  }
```

Lasso Connector Protocol Reference

LCP Commands

Listed here are all commands used in LCP.

cmdProtoErr

Indicates that an error has occurred in the use of the protocol.

Data Required	Four-byte integer indicating the error code. Any additional data will be a textual description of what went wrong.
Sent By	Lasso Service
Reply	None

cmdCloseReq

Sent by Lasso Service when there is no more data to be sent to the Web browser.

Data Required	None
Sent By	Lasso Service or client.
Reply	None

cmdGetParam

Request to return the value of a "named" parameter - server/environment variable or an HTTP request.

Data Required	RequestParamKeyword as defined in RequestParams.h Then a four-byte integer indicating the size of the data for the argument. Multiple params may follow.
Sent By	Lasso Service
Reply	cmdGetParamRep

cmdGetParamRep

Returns the value of a "named" parameter, as requested by cmdGetParam command.

Data Required	RequestParamKeyword as defined in RequestParams.h, then a four-byte integer indicating the size of the character data for the requested param. If multiple params were requested, the data for each param should follow in the original order.
Sent By	client
Reply	None

cmdPushData

Push partially processed data to a Web browser.

Data Required	The data that should be sent to the web browser.
Sent By	LassoService
Reply	None

Named Parameters

The following table lists all named parameters used in LCP.

Table 2: Named Parameters

Parameter	Description
rpSearchArgKeyword	All text in URL after the question mark.
rpUserKeyword	Username sent from browser.
rpPasswordKeyword	Password sent from browser.
rpAddressKeyword	IP address of client browser.
rpPostKeyword	HTTP object body (form data, etc.).
rpMethodKeyword	GET or POST, depending on <form method>.
rpServerName	IP address of server on which the Web server is running.
rpServerPort	IP port this hit came to (80 is common, 443 for SSL).
rpScriptName	Relative path from server root to this Lasso format file.
rpContentType	MIME header sent from client browser.
rpContentLength	The length in bytes of the POST data sent from <form POST>.
rpReferrerKeyword	URL of referring page.
rpUserAgentKeyword	Browser name and type.
rpClientIPAddress	IP address of client browser.
rpFullRequestKeyword	All MIME headers, uninterpreted.

8

Chapter 8
Lasso Java API

This chapter documents Lasso Java API (LJAPI) which can be used to develop new Lasso data source connectors, LDML tags, and data types.

- *Overview* introduces the API and describes the types of extensions that can be built.
- *What's New* describes what's new in LJAPI 7.
- *LJAPI 7 vs. LCAPI 7* compares and contrasts LJAPI 7 with LCAPI 7.
- *Requirements* includes platform-specific development environment details.
- *Getting Started* is a quick-start guide to building the LJAPI samples included with the Extending Lasso 7 Guide.
- *Debugging* includes platform-specific information about how to debug your projects.
- *Substitution Tag Operation* introduces the theory of operation behind creating substitution tags using LJAPI.
- *Substitution Tag Module Tutorial* describes authoring and building the sample LJAPI substitution tag.
- *Data Source Operation* introduces the theory of operation behind creating data source connectors using LJAPI.
- *Data Source Connector Tutorial* describes authoring and building the sample LJAPI data source connector.
- *LJAPI Interface Reference* includes details of every Java interface used in LJAPI.
- *LJAPI Class Reference* includes details of every Java class used in LJAPI.

Overview

LJAPI lets you write Java code to add new LDML tags, data source connectors, and data types to Lasso Professional 7. LJAPI is similar to LCAPI, but is tailored for the Java language.

It is generally recommended that data source connectors be developed using LCAPI instead of LJAPI, since they will offer easier installation compared to connectors built using LJAPI. However, as more database products appear on the market, many of them include support for Java to the extent that some are entirely Java-based. In this case, creating data source modules using LJAPI could provide advantages over LCAPI data sources in terms of development speed and efficiency, and in some cases could be the only available option to a developer.

In general, writing tags in LCAPI offers advantages over LJAPI and custom LDML tags in system speed and performance. However, tags must be compiled separately for Windows 2000/XP and Mac OS X in order to support each platform. Most importantly, while the client-side Java falls short of its original "write once, run everywhere" promise, it has become increasingly popular in the server application field where a graphical user interface is not needed and Java is implemented on the server side. In fact, certain aspects of Java, such as file and stream operations, could perform on par with or even outperform similar server components written in C/C++.

Finally, one of the important reasons for developing LJAPI modules is an enormous class library included with each Java VM install, covering almost every single programming need, from text processing to 2D/3D imaging to various network protocol implementations.

Custom tags written in LJAPI instantly support each platform. This chapter provides a walk-through for building an example substitution tag in LJAPI. Source code for the ZipCountTag module, as well as the code for the substitution tag, data source connector, and data type examples are included in the Lasso Professional 7/Documentation/4-ExtendingLasso/LJAPI folder on the hard drive.

What's New

Lasso Professional 7 introduces a complete rewrite of the Lasso Java Application Programming Interface (LJAPI). This new implementation provides many advantages over the former API, including a streamlined interface, increased speed, and feature parity with LCAPI 7.

Note: Older LJAPI modules written for Lasso Professional 5 will no longer work under Lasso Professional 7.

The most important change in LJAPI 7 is that it is now built upon LCAPI 7. Both API's share the same functionality and provide a single programming interface, making it easier for developers who wish to learn both APIs.

In addition to all of the features previously available in LCAPI 6 such as facilities for creating substitution tags and data source modules, LJAPI 7 also includes all new features in LCAPI 7. These features include:

- Asynchronous tags that run in their own thread.
- Container tags that operate like standard container tags.
- Data types which can have tag members, data members and callbacks.

LJAPI 7 also provides the ability to manipulate native LDML or other LCAPI/LJAPI data types by accessing their data members or running their member tags. For instance, developers can build arrays or maps that can be used from LDML format files using new LJAPI 7 functions. LJAPI 7 also allows any data type to be returned from an LJAPI tag, including binary data and complex data types such as arrays and maps.

LJAPI 7 provides new functions that allow tag parameters to be retrieved while preserving the parameter data type, and allows page variables of any type to be set and retrieved. LJAPI 7 also includes functions that aid in logging critical errors and debugging messages, and functions that query Lasso security for access permissions.

LJAPI 7 vs. LCAPI 7

Developers who have experience creating LCAPI modules will find themselves familiar with the new Lasso Java API. Similarly, Java developers who learn to use LJAPI 7 will find it easy to write LCAPI modules once they are ready to make a transition to a different language.

The following sections outlines a few basic differences between LCAPI 7 and LJAPI 7.

LJAPI 7 is Object-Oriented

The majority of Lasso API functions must be aware of the current Lasso state in order to operate correctly. In order to solve the problem resulting from the non-OO nature of the C-based Lasso API, LCAPI introduced the token concept. When Lasso calls one of the methods implemented by an LCAPI module, it passes an opaque parameter of type lasso_request_t, which encapsulates the information about the current state of the request.

The module then makes calls to Lasso while passing the token in the first parameter to every API function.

In LJAPI 7, the same state information is stored in an instance of the LassoCall Java class. All LJAPI 7 functions are implemented as members of the LassoCall class, which eliminates the need to pass a token parameter with each call.

This results in one of the most notable differences between LJAPI and LCAPI, in that LJAPI methods usually take one parameter less that their native LCAPI counterparts.

LJAPI Uses Shorter Function Names

In LCAPI, function names begin with the lasso_ prefix, reflecting the name space in which they reside. However, the corresponding LJAPI methods are implemented as members of LassoCall class. For this reason, the lasso_ prefix has been removed from all Java method names.

The following shows the lasso_getTagName function in LCAPI:

```
lasso_getTagName( lasso_request_t token, auto_lasso_value_t &name );
```

The following shows the equivalent getTagName method in LJAPI:

```
getTagName( LassoValue name );
```

Tokenless LCAPI Functions are Static Methods in LJAPI

There are few LCAPI functions that do not take the token state parameter. These functions are implemented in LJAPI as static methods of the LassoCall class:

The following shows the lasso_registerConstant function in LCAPI:

```
osError lasso_registerConstant( const char * name, lasso_type_t val );
```

The following shows the equivalent registerConstant method in LJAPI:

```
int LassoCall.registerConstant( String name, LassoTypeRef val );
```

LJAPI Does Not Use Function Pointers

Some LCAPI functions use a function pointer parameter of type lasso_tag_func. Since function pointers do not exist in Java, the corresponding LJAPI methods instead accept a pair of string parameters that specify the class and method name.

The following shows the lasso_typeAllocTag function in LCAPI:

```
osError lasso_typeAllocTag ( lasso_request_t token, lasso_type_t * outTag, lasso_
tag_func nativeTagFunction );
```

The following shows the equivalent typeAllocTag method in LJAPI:

```
int typeAllocTag ( LassoTypeRef outTag, String className, String methodName );
```

Requirements

In order to write your own LDML substitution tags, data source connectors, or custom data types in Java, you will need the following:

Windows

- Microsoft Windows 2000, Microsoft Windows XP Professional, or better.
- Java 2 SDK 1.4 or higher.
- Windows Lasso Professional 7 or higher.

Mac OS

- Mac OS X with Java 2 SDK installed (included).
- Mac OS X Lasso Professional 7 or higher.

Getting Started

This section provides a walk-through for building sample LJAPI tag modules in Windows 2000/XP and Mac OS X.

To build a sample LJAPI tag module using Apache Ant:

Apache Ant is a de-facto standard Java-based build tool, part of the Apache open-source initiative.

In order to build the sample code, you will need to install complete Ant package, downloadable from the following location:

```
http://ant.apache.org/
```

If you do not wish to install Apache Ant at this time, you can skip to the next section for instructions on building the code examples with the javac compiler tool.

Note: All LJAPI examples have been tested with the most recent stable version of the Ant tool (v1.5.2) available at the time of the Lasso Professional 7 release.

To build all included code examples:

1 Launch the command prompt (Windows 2000/XP), or open the Terminal application (Mac OS X)

2 Locate the following folder in the hard drive:

```
Lasso Professional 7/Documentation/4-ExtendingLasso/LJAPI/Sample Code
```

3 Make this folder your current working directory.

Windows:

```
cd "C:\Program Files\Blue World Communications\Lasso Professional 7\
Documentation\4-ExtendingLasso\LJAPI\Sample Code"
```

Mac OS X:

```
cd "/Applications/Lasso Professional 7/Documentation/4-ExtendingLasso/LJAPI/
Sample Code"
```

4 Invoke Ant tool by entering the "ant" command at the command prompt, optionally followed by the target (sub-project) name:

```
ant <target-name>
```

Compiled LJAPI modules will be placed in the Modules (output) folder located inside the Sample Code directory.

5 To install sample LJAPI modules using the Ant tool, enter the following command at the command prompt:

```
ant install
```

Sample LJAPI modules can also be installed manually, by dragging one or more Java class/jar files from the Modules (output) folder to the LassoModules folder.

6 Restart Lasso Professional.

Please note that, when launched without an optional target name parameter (step 4), Ant will execute the default target defined in the "build.xml" descriptor file. This target has been pre-configured to compile all sample LJAPI modules. Individual modules can be also built separately by specifying one of the following target names on the command line: zipcount, zip, pdf, nntp, mysql, xml or docs.

Two special targets (clean and install) can be used for deleting the contents of the Modules (output) directory, and copying LJAPI modules to the LassoModules folder, respectively.

For further details, please see the contents of the build.xml descriptor file.

Alternately, you can also build the ZipCountTag module using the <javac> command-line tool included with Java SDK from Sun Microsystems.

To build ZipCountTag module using the <javac> command-line tool:

1 Launch the Windows 2000/XP command prompt.

2 Make the following folder your current directory.

```
C:\Program Files\Blue World Communications\Lasso Professional 7\
Documentation\4-ExtendingLasso\LJAPI\Sample Code\Substitution Tags\ZipCountTag
```

3 Enter the path of the Java compiler tool javac, followed by the -classpath option keyword and the path to the LJAPI.jar file (contains all Java classes used by LJAPI modules), followed by the ZipCountTag module source file path:

```
javac -classpath ../../../../../../LassoModules/LJAPI.jar ZipCountTag.java
```

If Java SDK has been installed in the jdk1.4 folder, your command line might look like this:

```
C:\jdk1.4\bin\javac -classpath ..\LJAPI.jar ZipCountTag.java
```

4 After building, a ZipCountTag.class file will be created inside your ZipCountTag project folder.

5 Open the ZipCountTag folder and drag ZipCountTag.class into the Lasso Professional 7\LassoModules folder on the hard drive.

6 Stop and then restart Lasso Service.

7 The new tag [Zip_Count] is now part of the LDML language.

8 Drag the sample Lasso format file called ZipCountTag.lasso and the LJAPITest.zip test file into your Web server root.

9 In a Web browser, view http://localhost/ZipCountTag.lasso to see the new LDML tags in action.

To build a sample LJAPI tag module in Mac OS X:

1 Open a Terminal window.

2 Change the current folder to the Lasso Professional 7/Documentation folder using the following command:

```
cd /Applications/Lasso\ Professional\ 7/Documentation/4-ExtendingLasso/LJAPI/
Sample\ Code/Substitution\ Tags/ZipCountTag
```

3 Build the sample project using the provided makefile. This requires that you be logged in as the root user.

```
make
```

Alternatively, you can build the module by manually invoking the Java compiler:

```
javac -classpath ../../../../../../LJAPI.jar ZipCountTag.java
```

4 After building, a Java class file named ZipCountTag.class will be created in the current folder. This is the LJAPI module you'll install into the LassoModules folder.

5 Copy the newly-created module to the Lasso modules folder using the following command:

```
cp ZipCountTag.class /Applications/Lasso\ Professional\ 7/LassoModules
```

6 Quit Lasso Service if it's running, so that the next time it starts up, it will load the new module you just built (you'll need to know a root password to use sudo).

```
cd /Applications/Lasso\ Professional\ 7/Tools/
sudo ./stoplassoservice.command
```

7 Start the Lasso Service back up, so it will load the new module.

```
sudo ./startlassoservice.command
```

The new [Zip_Count] tag is now part of the LDML language.

8 Copy the sample Lasso format file called ZipCountTag.lasso and the LJAPITest.zip test file from your Lasso Professional 7/Documentation/4-ExtendingLasso/LJAPI/Tags/ZipCountTag folder into your Web server document root.

9 Use a Web browser to view http://localhost/ZipCountTag.lasso to see the new LDML tags in action.

Debugging

You can set breakpoints in your LJAPI class files and perform source-level debugging for your own code. In order to set this up, add path information to your project so it knows from where to load executables. For this section, we will use the provided substitution tag project as the example.

To set breakpoints in your LJAPI code:

1 Lasso Professional 7 allows you to specify Java Virtual Machine options used for launching JVM upon Lasso startup. These options are stored in the lasso_internal.global_prefs table as java_vm_options in the store_key field. To enable remote debugging on port 8000, add the following two options to the data column in the lasso_internal.global_pref table:

```
-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```

2 After restarting Lasso Professional 7, launch JDB with the following option:

```
jdb -attach 8000
```

3 Once attached to the JVM, you can set the breakpoints, single-step through your code, catch exceptions, etc. Please note that you can store multiple JVM options in the same column. To monitor the GC activity, add -verbose:gc option, or use -verbose:jni to print JNI messages to the standard output.

For more information on the options available for your platform and JVM, please consult the JVM vendor documentation. For a list of non-standard options available for your JVM, review the Xusage.txt file:

Mac OS X:

/System/Library/Frameworks/JavaVM.framework/Home/lib/Xusage.txt

Windows:

<path-to-jvm.dll>/Xusage.txt

Substitution Tag Operation

An LJAPI module is essentially a regular Java class file. When Lasso Professional 7 first starts up, it looks for module files (Windows DLLs or Mac OS X DYLIBS) in its LassoModules folder. As it encounters and loads an LJAPI 7 module, it launches the JVM and proceeds to scan the folder for other LJAPI modules. Upon finding a Java class file, Lasso attempts to determine if it is derived from the com.blueworld.lassopro.JavaModule class. If it is, then Lasso loads the class while performing necessary instantiation and calls the registerLassoModule() function that is implemented in that class:

```
public void registerLassoModule()
```

At this point, the module must call the following method as many times as needed, once for each tag implemented by the module:

```
void registerTagModule( String moduleName,
                        String tagName,
                        String methodName,
                        int flags,
                        String description);
```

After a tag module is registered with Lasso Professional 7, it can provide information about the name of the tag and the name of the Java method that is implementing that tag. It also can provide a short description, and any special flags describing unique features implemented by that tag.

All registered information is later used for dispatching the task of executing a particular tag found in a .lasso format file to an appropriate LJAPI module, or executing a data source action.

For example, the following code tells Lasso to call the Java class called ZipCountTag whenever the LDML [Zip_Count] is encountered inside a .lasso format file. The first parameter of the registerTagModule method is the module name, the second is the tag name, and the third one is the name of the function implementing the tag. The last two parameters are the tag type flag and a short description:

```
public void registerLassoModule()
  {
    registerTagModule( "ZipCountTag", "zip_count", "myZipCountFunc",
    FLAG_SUBSTITUTION, "Count items in a zip file" );
  }
```

Below is the LDML needed in a Lasso format file in order to get the custom tag to execute:

```
<html>
  <body>
    Count of items in the LjapiTest.zip file:
    [Zip_Count:'LjapiTest.zip']
    <!-- This should display "2" when page executes -->
  </body>
</html>
```

This will produce the following:

```
2
```

Substitution Tag Tutorial

The following section provides a walk-through of building an example tag to show how LJAPI features are used. This code will be most similar to the sample ZipCountTag LJAPI project. In order to build this project, copy the ZipCountTag project folder and edit the project files inside it.

The module relies on a Java class library to do most of the work, particularly the java.util.zip package which provides a variety of functions for manipulating the contents of Zip files—standard compressed archives widely used on the Internet.

The [Zip_Count] tag implemented in the ZipCountTag LJAPI module simply displays the number of files and directories stored in a Zip file when called from an LDML format file.

Example sample tag LDML syntax:

```
[Zip_Count: -Zipfile='LJAPITest.zip', -FilesOnly]
```

Notice the required convention of placing a dash in front of all named parameters in order to make them easier to spot in the LDML code, and

prevent ambiguities in the LDML parser. Notice the tag takes one string parameter named -Zipfile, and an optional keyword parameter named -FilesOnly. In general, LDML does not care about the order in which you pass parameters, so plan to make this tag as flexible as possible by not assuming anything about the order of parameters. The following variations should work exactly the same.

Example of sample tag with different ordered parameters:

```
[Zip_Count: -Zipfile='LJAPITest.zip', -FilesOnly]

[Zip_Count: -FilesOnly, -Zipfile='LJAPITest.zip']
```

Substitution Tag Module Code

Shown below is the code for the substitution tag module. Line numbers are provided to the left of each line of code, and are referenced in the *Substitution Tag Module Walk-Through* section.

Note: The line numbers shown refer to the line numbers of the code in the actual file being created, not as shown in this page. Some single lines of code may carry into two or more lines as shown on this page.

Substitution Tag Module Code

```
 1 import com.blueworld.lassopro.*;
 2 import java.io.*;
 3 import java.util.*;
 4 import java.util.zip.*;
 5 public class ZipCountTag extends LassoTagModule
 6 {
 7 public void registerLassoModule()
 8 {
 9   registerTagModule( “Zip”, “zip_count”, “myZipCountFunc”,
10   FLAG_SUBSTITUTION, “Count items in a zip file” );
11 }
12
13   public int myZipCountFunc(LassoCall lasso, int action)
14   {
15     int err = ERR_NOERR;
16     try {
17       IntValue count = new IntValue();
18       err = lasso.getTagParamCount( count );
19       if ( err == ERR_NOERR && count.intValue() > 0 )
20       {
21         String zipName = null;
22         boolean filesOnly = false;
23         LassoValue param1 = new LassoValue();
```

```
24          LassoValue param2 = new LassoValue();
25          err = lasso.findTagParam( "-zipfile", param1 );
26          if ( err != ERR_NOERR || param1.name() == null )
27            lasso.getTagParam( 0, param1 );
28          if ( param1.name() == null || param1.name().length() == 0 )
29            return LassoErrors.InvalidParameter;
30          if ( count.intValue() > 1 &&
31               lasso.getTagParam( 1, param2 ) == ERR_NOERR )
32            filesOnly = param2.equalsIgnoreCase("-filesonly");
33          String filePath = lasso.fullyQualifyPath( param1.name() );
34          filePath = lasso.resolvePath( filePath );
35          filePath = lasso.getPlatformSpecificPath( filePath );
36          ZipFile zip = new ZipFile( filePath );
37          Enumeration enum = zip.entries();
38          ZipEntry entry = null;
39          int zipcount = 0;
40          while ( enum.hasMoreElements() )
41          {
42            entry = (ZipEntry)enum.nextElement();
43            if ( !filesOnly || !entry.isDirectory() )
44              ++zipcount;
45          }
46          err = lasso.outputTagData( Integer.toString( zipcount ) );
47          zip.close();
48        }
49      }
50      catch ( java.io.Exception e )
51      {
52        lasso.setResultMessage( e.getMessage() );
53        return LassoErrors.FileNotFound;
54      }
55      return err;
56    }
57  }
```

Substitution Tag Module Walk-Through

This section provides a step-by-step walk-through for building the substitution tag module.

To write a sample LJAPI tag module:

1 First, import com.blueworld.lassopro.* classes as shown in line 1.

```
1  import com.blueworld.lassopro.*;
2  import java.io.*;
3  import java.util.*;
4  import java.util.zip.*;
```

2 Define your class to be a subclass of the blueworld.lasso.LassoSubstitutionTag class.

```
5  public class ZipCountTag extends LassoTagModule
```

3 Define the registerLassoModule method.

```
7  public void registerLassoModule() {
```

Every Lasso module must implement the registerLassoModule() method. This method will be called by Lasso at startup, giving your module a chance to register its tags.

4 Register the tags implemented by your module.

```
9  registerTagModule( “Zip”, “zip_count”, “myZipCountFunc”,
10 FLAG_SUBSTITUTION, “Count items in a zip file” );
```

Call this method as many times as there are tags implemented in your module. This method takes five parameters: the module name, the name of LDML tag, the name of the Java method implemented by your module (to be called when the corresponding LDML tag is found on the page), any additional tag feature flags, and a brief tag description.

5 Define the tag formatting method with the same name as indicated in the third parameter of the corresponding registerTagModule call.

```
13 public int myZipCountFunc(LassoCall lasso, int action)
```

This is the method that does all the work. Every tag registered by your module can have its own formatting method. Its purpose is to perform an action based on the parameters passed to the tag and/or current request properties. Most substitution tags would output the data, although some may perform other actions such as setting page variables, manipulating files, etc.

When Lasso encounters one of the tags registered by your module, it creates new module instance and calls the corresponding method, passing the LassoCall object which then can be used by the module for calling back into Lasso.

6 Define the variable to hold the result code returned by various LassoCall methods.

```
15  int err = ERR_NOERR;
```

7 Our [Zip_Count] LDML tag takes one required and one optional parameter. We need to make sure at least one parameter (filename) is present, otherwise we won’t be able to continue.

```
17 IntValue count = new IntValue();
18 err = lasso.getTagParamCount( count );
19 if ( err == ERR_NOERR && count.intValue() > 0 )
```

8 Define the storage for the zip file name, optional -FilesOnly parameter, and LassoValue object to be used with various LassoCall methods.

```
21 String zipName = null;
22 boolean filesOnly = false;
23 LassoValue param = new LassoValue();
```

9 Our tag should be flexible enough to accept both named and unnamed versions of the required parameter. First, try to search for the parameter by a name.

```
25 err = lasso.findTagParam( "-zipfile", param1 );
```

10 If this fails, assume the first unnamed tag parameter to hold the file path name. Call getTagParam() with the index 0 (tag parameter numbering is zero-based).

```
26 if ( err != ERR_NOERR || param1.name() == null )
27   err = lasso.getTagParam( 0, param1 );
```

11 Next, make sure we've got a valid value. If the filename parameter contains an empty string, immediately return from our method, passing InvalidParameter result code back to Lasso.

```
28 if ( err != ERR_NOERR || param1.name().length() == 0 )
29   return LassoErrors.InvalidParameter;
```

12 Our tag also accepts an optional boolean parameter -FilesOnly, indicating that directories must be ignored while counting zip file items. If more than one parameter was supplied to our tag, try determining if it was the optional -FilesOnly parameter.

```
30 if ( count.intValue() > 1 &&
31      lasso.getTagParam( 1, param2 ) == ERR_NOERR )
32   filesOnly = param2.equalsIgnoreCase("-filesonly");
```

13 The path to the zip file is relative to the server root. In order to find out the actual location of the file, you can use a number of LassoCall class methods suited for converting a file path name into a fully qualified platform-specific path. fullyQualifyPath() turns a relative path into a from-the-server-root path. resolvePath() converts a from-the-root path into a full internal path. Finally, getPlatformSpecificPath() will convert an internal path name into a platform-specific path name.

```
33 String filePath = lasso.fullyQualifyPath( param1.name() );
34 filePath = lasso.resolvePath( filePath );
35 filePath = lasso.getPlatformSpecificPath( filePath );
```

14 Now attempt to instantiate a ZipFile object using a platform-specific path name. Any exceptions thrown by the object constructor will be caught by the try/catch block wrapping our method's body.

```
36 ZipFile zip = new ZipFile( filePath );
```

15 Prepare to enumerate items in the zip file.

```
37 Enumeration enum = zip.entries();
```

16 Define the storage for holding the zip item count.

```
38  ZipEntry entry = null;
39  int zipcount = 0;
```

17 Iterate through the zip archive items, incrementing the counter for all items matching our criteria.

```
40  while ( enum.hasMoreElements() )
41  {
42    entry = (ZipEntry)enum.nextElement();
43    if ( !filesOnly || !entry.isDirectory() )
44      ++zipcount;
45  }
```

18 Output the resulting zip file item count.

```
46  err = lasso.outputTagData( Integer.toString( zipcount ) );
```

19 Close the zip file.

```
47  zip.close();
```

20 Make sure that any possible exceptions are handled correctly in your code. In this particular case, we simply pass the message retrieved from the Exception object back to Lasso, and return the FileNotFound error code. For a complete listing of error codes, see the variables defined in the LassoErrors class.

```
50  catch ( Exception e )
51  {
52    lasso.setResultMessage( e.getMessage() );
53    return LassoErrors.FileNotFound;
54  }
```

Data Source Connector Operation

When Lasso Professional 7 starts up, it looks for module files (Windows DLLs or Mac OS X DYLIBS) in the LassoModules folder. As Lasso encounters each module, it executes the module's registerLassoModule() function once and only once. It is the job of the LJAPI developer to write code to register each new data source (or custom tag) methods in this registerLassoModule() function. Both substitution tags and data sources may be registered at the same time, and the code for them can reside in the same module. The only difference between registering a data source and a substitution tag is whether registerTagModule() or registerDSModule() is called.

Data sources are typically more complex than substitution tags because Lasso Service calls them with many different actions during the course of various database operations. Whereas a substitution tag only needs to know how to format itself, a data source needs to enumerate its tables, search through records, add new records, delete records, and more. Even so,

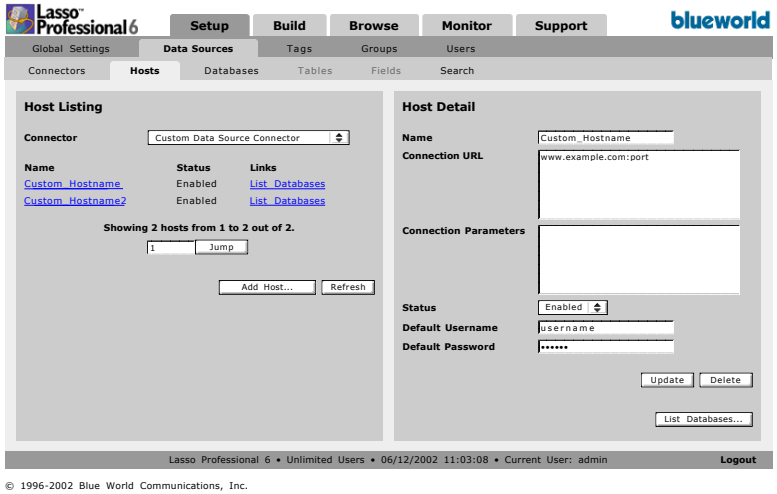
this added complexity is easily handled with a single switch() statement, as you will see in the *Data Source Connector Tutorial* section of this chapter.

Data Source Connectors and Lasso Administration

Once a custom data source connector module is registered by Lasso, it will appear in the *Setup > Data Sources > Connectors* section of Lasso Administration. If a connector appears here, then it has been installed correctly.

The administrator adds the data source connection information to the *Setup > Data Sources > Hosts* section of Lasso Administration, which sets the parameters by which Lasso connects to the data source via the connector. This information is stored in the Lasso_Internal Lasso MySQL database, where the connector can retrieve and use the data via function calls.

Figure 1: Custom Data Source Host Screen



The data that the administrator can submit in the *Setup > Data Sources > Hosts* section of Lasso Administration includes the following:

- **Name** – The administrator-defined name of the data source host.
- **Connection URL** – The URL string required for Lasso to connect to a data source via the connector. This typically includes the IP address of the machine hosting the data source.

- **Connection Parameters** – Additional parameters passed with the Connection URL. This can include the TCP/IP port number of the data source.
- **Status** – Allows the administrator to enable or disable the connector in Lasso Professional 7.
- **Default Username** – The data source username required for Lasso to gain access to the data source.
- **Default Password** – The data source password required for Lasso to gain access to the data source.

The Connection URL, Connection Parameters, Default Username, and Default Password values are passed to the data source via data source function methods in the com.blueworld.lassopro.LassoCall class, which are described in the *LJAPI Class Reference* section of chapter.

Data Source Connector Tutorial

The following section provides a walk-through of an example data source to show how some of the LJAPI features are used. This code will be most similar to the sample NNTPDataSource project, which is provided with Lasso Professional 7 in the following folder.

Lasso Professional 7/Documentation/4-ExtendingLasso/LJAPI/DataSourceConnectors/Nntp_ds

The example data source connector bridges a news (NNTP) server and Lasso Professional 7. Network News Transfer Protocol (NNTP) is used to read and post articles on Usenet news servers. This specific example has been tested with the Microsoft NNTP Service 5.0, and it provides a good start for any developer desiring to build a data source connector module supporting a large variety of other NNTP servers.

While an NNTP server is not exactly an RDBMS, there are some advantages to implementing the NNTP client as a data source connector. The hierarchy of a news storage is somewhat similar to that of a traditional RDBMS. News articles (rows) are organized in groups (tables), which in turn are parts of distributions (databases). However, due to a sheer number of news groups available on an average news server (2000-50000+), treating groups as database tables would put a big load on the internal Lasso security mechanism, which is required to keep track of permissions for every registered database table. Therefore, the hierarchy has been adopted to minimize the stress put on Lasso security.

The NNTP connector adds a single News database containing two static tables: Groups and Articles. Performing a search on the Group table returns

a list of groups available on the server. Similarly, executing a query on the Articles table retrieves a range of articles from a specific newsgroup.

Updating groups or articles is not supported by the NNTP protocol, so only search and insert data source actions are implemented by this connector. SQL actions are also not supported, although it is possible to build a simple parser for translating SQL statements into commands understood by NNTP servers.

Data Source Connector Code

Below is the code for the data source module. Line numbers are provided to the left of each line of code, and are referenced in the *Data Source Connector Walk-Through section*.

Data Source Connector Code

```
1  import com.blueworld.lassopro.*;
2  import java.net.*;
3  import java.io.*;
4  import java.util.*;
5  public class NNTP_DS extends LassoDSModule
6  {
7  Socket sock;
8  PrintStream printer;
9  BufferedReader reader;
10 String host = null;
11 int port = 0;
12 String user = null, pass = null;
13 String hostInfo = "";
14 Vector headers = new Vector(10);
15 int refsIdx = -1;
16 int xrefIdx = -1;
17 int bytesIdx = -1;
18 boolean useXpat = false;
19 String groupFilter = "";
20 String group = "";
21 String article = "";
22 int groupCount = -1;
23 int articleCount = -1;
24
25 public void registerLassoModule () {
26   registerDSModule("NNTP", "dsFunc", 0, "Lasso Connector for NNTP",
       "Simple Usenet client");
27 }
28 public int dsFunc (LassoCall lasso, int cmd, LassoValue value) {
29   int err = ERR_NOERR;
30   switch (cmd) {
```

```
31      case ACTION_INIT:
32        err = doInit(lasso);
33        break;
34      case ACTION_TERM:
35        err = doTerm(lasso);
36        break;
37      case ACTION_EXISTS:
38        if (!value.data().equalsIgnoreCase("News"))
39          err = LassoErrors.WebNoSuchObject;
40        break;
41      case ACTION_DB_NAMES:
42        err = doDBNames(lasso);
43        break;
44      case ACTION_TABLE_NAMES:
45        err = doTableNames(lasso, value.data());
46        break;
47      case ACTION_INFO:
48        err = doInfo(lasso, true);
49        break;
50      case ACTION_SEARCH:
51        err = doSearch(lasso);
52        break;
53    }
54    return err;
55  }
56  int doInit(LassoCall lasso) {
57    return ERR_NOERR;
58  }
59  int doTerm(LassoCall lasso) {
60    close();
61    return ERR_NOERR;
62  }
63  int doDBNames(LassoCall lasso) {
64    return lasso.addDataSourceResult("News");
65  }
66  int doTableNames(LassoCall lasso, String db) {
67    if (!db.equalsIgnoreCase("News"))
68      return -1;
69    lasso.addDataSourceResult("Groups");
70    lasso.addDataSourceResult("Articles");
71    return ERR_NOERR;
72  }
73  int doInfo(LassoCall lasso, boolean listAllCols) {
74    int err = ERR_NOERR;
75    LassoValue tbl = new LassoValue();
76    err = lasso.getTableName(tbl);
77    if (err != ERR_NOERR || tbl.data().length() == 0)
78      return LassoErrors.InvalidParameter;
```

```
79      if (!connect(lasso))
80      return LassoErrors.Network;
81    if (tbl.data().equalsIgnoreCase("Groups")) {
82      lasso.addColumnInfo("Group", 0,
            LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
83      lasso.addColumnInfo("Last", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
84      lasso.addColumnInfo("First", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
85      lasso.addColumnInfo("AllowPost", 0,
            LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
86    } else if (tbl.data().equalsIgnoreCase("Articles")) {
87      if (!this.headers.isEmpty()) {
88        String str;
89        int type, count = headers.size();
90        lasso.addColumnInfo("Number", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
91        for (int i = 0; i < count; ++i) {
92          str = (String)this.headers.elementAt(i);
93          if (str.equalsIgnoreCase("Lines") ||
              str.equalsIgnoreCase("Bytes"))
94            type = LassoValue.TYPE_INT;
95          else if (str.equalsIgnoreCase("Date"))
96            type = LassoValue.TYPE_DATETIME;
97          else
98            type = LassoValue.TYPE_CHAR;
99          err = lasso.addColumnInfo((String)headers.elementAt(i), 0,
                type, PROTECTION_READ_ONLY);
100       }
101       if (listAllCols) {
102         lasso.addColumnInfo("Headers", 0,
                LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
103         lasso.addColumnInfo("Body", 0,
                LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
104       }
105     }
106   }
107   return err;
108 }
109 int doSearch(LassoCall lasso) {
110   int err = ERR_NOERR;
111   int skip = 0;
112   int max = 50;
113   int totalcount = 0;
114   String filter = "", reply = "";
115   LassoValue tbl = new LassoValue();
116   LassoValue val = new LassoValue();
117   IntValue ival = new IntValue();
```

```
118   if (lasso.getSkipRows(ival) == ERR_NOERR)
119     skip = ival.intValue();
120   if (lasso.getMaxRows(ival) == ERR_NOERR)
121     max = ival.intValue();
122   lasso.getTableName(tbl);
123   lasso.getInputColumnCount(ival);
124   if (!connect(lasso))
125     return LassoErrors.Network;
126   if ((err = doInfo(lasso, max == 1)) != ERR_NOERR)
127     return err;
128   try {
129     if (tbl.data().equalsIgnoreCase("GROUPS")) {
130       if (lasso.findInputColumn("group", val) == ERR_NOERR) {
131         if (val.type() == LassoOperators.OP_ENDS_WITH)
132           filter = '*' + val.data();
133         else if (val.type() == LassoOperators.OP_CONTAINS)
134           filter = '*' + val.data() + '*';
135         else if (val.type() == LassoOperators.OP_EQUALS)
136           filter = val.data();
137         else
138           filter = val.data() + '*';
139       }
140       this.printer.print("LIST ACTIVE " + filter + "\r\n");
141       reply=reader.readLine();
142       if (!reply.startsWith("2"))
143         return setError(lasso, reply);
144       if (!this.groupFilter.equalsIgnoreCase(filter)) {
145         this.groupFilter = filter;
146         this.groupCount = -1;
147       }
148       err = addGroups(lasso, skip, max);
149     } else if (tbl.data().equalsIgnoreCase("ARTICLES")) {
150       if (lasso.findInputColumn("-group", val) == ERR_NOERR ||
151         lasso.findInputColumn("group", val) == ERR_NOERR) {
152         if (val.data().length() > 0) {
153           if (!val.data().equalsIgnoreCase(this.group))
154             this.articleCount = -1;
155           this.group = val.data();
156         }
157       }
158       if (this.group == null || this.group.length() < 1) {
159         lasso.setResultMessage("Missing group parameter.");
160         return LassoErrors.InvalidParameter;
161       }
162       String id = null;
163       ival.setInt(0);
164       if (lasso.getRowID(ival) == ERR_NOERR && ival.intValue() != -1)
165         id = Integer.toString(ival.intValue());
```

```
166     else if (lasso.getPrimaryKeyColumn(val) == ERR_NOERR &&
167           (val.name().equalsIgnoreCase("message-id") ||
168            val.name().equalsIgnoreCase("number")))
169       filter = val.data();
170     else if (lasso.findInputColumn("message-id", val) == ERR_NOERR ||
171           lasso.findInputColumn("number", val) == ERR_NOERR)
172       filter = val.data();
173     if (this.articleCount == -1) {
174       err = selectGroup(lasso);
175       if (err != ERR_NOERR)
176         return err;
177     }
178     if (max == 1 && (filter == null || filter.length() < 1))
179       id = getRange(lasso, skip, 1);
180     if (filter.startsWith("<") || filter.indexOf('-') == -1)
181       id = filter;
182     if (id != null && id.length() > 1) { // detail
183       this.printer.print("ARTICLE " + id + "\r\n");
184       reply=reader.readLine();
185       if (!reply.startsWith("2"))
186         return setError(lasso, reply);
187       int idx=0, i=0, bytes=0;
188       String str;
189       String[] data = new String[headers.size()+3];
190       data[0] = reply.substring(4, reply.indexOf(' ', 4));
191       data[data.length-1] = ""; // body
192       data[data.length-2] = ""; // headers
193       while(!(reply=reader.readLine()).startsWith(".")) {
194         bytes += reply.length() + 2;
195         i = reply.indexOf(": ");
196         if (i != -1) { // header
197           str = reply.substring(0,i);
198           idx = this.headers.indexOf(str);
199           if (idx != -1) // known header
200             data[idx+1] = reply.substring(i+2);
201           else
202             data[data.length-2] += reply + '\r';
203         } else { // body
204           StringBuffer buf = new StringBuffer();
205           while (!(reply=reader.readLine()).startsWith(".")) {
206             bytes += reply.length() + 2;
207             buf.append(reply).append('\r');
208           }
209           data[data.length-1] = buf.toString();
210           data[this.bytesIdx] = Integer.toString(bytes + 2);
211           break;
212         }
213       }
```

```
214           if (data[0].equals("0") && this.refsIdx != -1) {
215             idx = data[this.xrefIdx].lastIndexOf(group);
216             if (idx != -1) {
217               str = data[this.xrefIdx].substring(idx + group.length() + 1);
218               if ((idx=str.indexOf(' ')) != -1)
219                 str = str.substring(0, idx);
220               data[0] = str;
221             }
222           }
223           err = lasso.addResultRow(data);
224         } else { // GET LIST
225           if (filter == null || filter.length() == 0)
226             filter = getRange(lasso, skip, max);
227           this.printer.print("XOVER " + filter + "\r\n");
228           reply=reader.readLine();
229           if (!reply.startsWith("2"))
230             return setError(lasso, reply);
231           while(err == ERR_NOERR && !(reply=reader.readLine()).startsWith("."))
232             err = lasso.addResultRow(split(reply, "\t"));
233         }
234         lasso.setNumRowsFound(this.articleCount);
235       }
236     } catch (Exception e) {
237       System.err.println(e.toString());
238       lasso.setResultMessage(e.getMessage());
239       err = LassoErrors.Network;
240       try { this.sock.close(); }
241         catch (Exception e2) {}
242       this.sock = null;
243     }
244     return err;
245   }
246   int setError(LassoCall lasso, String reply) {
247     int err = -1;
248     try {
249       err = Integer.parseInt(reply.substring(0, 3));
250       lasso.setResultMessage(reply.substring(4));
251     } catch (Exception e) {};
252     lasso.setResultCode(err);
253     return err;
254   }
255   String[] split (String str, String ch) {
256     int i = 0;
257     int numcols = headers.size() + 1;
258     String cols[] = new String[ numcols ];
259     StringTokenizer tok = new StringTokenizer(str, ch);
260     int count = tok.countTokens();
261     while (tok.hasMoreTokens()) {
```

```
262       if (i == this.refsIdx && numcols > count)
263         cols[i++] = ""; // empty References field
264       cols[i++] = tok.nextToken();
265     }
266     return cols;
267   }
268   boolean connect(LassoCall lasso) {
269     if (getHostInfo(lasso) != ERR_NOERR)
270       return false;
271     try
272     {
273       String reply;
274       if (this.sock != null) {
275         this.printer.print("MODE READER\r\n"); // probe the connnection
276         reply = reader.readLine();
277         if (!reply.startsWith("2")) {
278           this.sock.close();
279           this.sock = null;
280         }
281       }
282       if (this.sock == null) {
283         this.sock=new Socket(this.host,this.port);
284         this.reader=new BufferedReader(new InputStreamReader(this.sock.getInputStrea
    m()), 2500);
285         this.printer=new PrintStream(new BufferedOutputStream(this.sock.getOutputStrea
    m(),2500),true);
286         this.hostInfo = this.reader.readLine();
287         login();
288         this.printer.print("MODE READER\r\n");
289         reader.readLine();
290         if (this.headers.isEmpty()) {
291           printer.print("LIST OVERVIEW.FMT\r\n");
292           reply = reader.readLine();
293           if (reply.startsWith("2")) {
294             int idx, i = 1;
295             while(!(reply=reader.readLine()).startsWith(".")) {
296               idx = reply.indexOf(':');
297               if (idx != -1)
298                 reply = reply.substring(0, idx);
299               this.headers.addElement(reply);
300               if (reply.equalsIgnoreCase("References"))
301                 this.refsIdx = i;
302               else if (reply.equalsIgnoreCase("Bytes"))
303                 this.bytesIdx = i;
304               else if (reply.equalsIgnoreCase("Xref"))
305                 this.xrefIdx = i;
306               ++i;
307             }
```

```
            }
          }
        }
      } catch (Exception e) {
        lasso.setResultMessage(e.getMessage());
        this.sock = null;
        return false;
      }
      return true;
    }
    void close() {
      this.host = null;
      this.headers.clear();
      this.groupCount = this.articleCount = -1;
      this.refsIdx = this.xrefIdx = this.bytesIdx = -1;
      try
      {
        this.printer.print("QUIT\r\n");
        this.reader.close();
        this.printer.close();
        this.sock.close();
        this.sock = null;
      } catch (Exception e) {}
    }
    boolean login()
    {
      if (user != null && user.length() > 0) {
        try
        {
          this.printer.print("AUTHINFO USER " + this.user + "\r\n");
          this.printer.print("AUTHINFO PASS " + this.pass + "\r\n");
          return (reply.startsWith("281"));
        } catch (Exception e) {}
      }
      return false;
    }
    int getHostInfo(LassoCall lasso) {
      int err = ERR_NOERR;
      LassoValue hostPort = new LassoValue();
      LassoValue userPass = new LassoValue();
      err = lasso.getDataHost(hostPort, userPass);
      if (err != ERR_NOERR ||
          hostPort.name() == null ||
          hostPort.name().length() == 0)
        return err;
      if (!hostPort.name().equalsIgnoreCase(this.host))
        close();
      this.host = hostPort.name();
```

```
356    this.user = userPass.name();
357    this.pass = userPass.data();
358    try {
359      this.port = Integer.parseInt(hostPort.data());
360    } catch (Exception e) {}
361
362    if (this.port == 0)
363      this.port = 119; // default NNTP port
364    return ERR_NOERR;
365  }
366  int addGroups(LassoCall lasso, int skip, int max) {
367    int err = ERR_NOERR;
368    int count = 0;
369    boolean getFirst = (max == 1 || this.group == null || this.group.length() < 1);
370    String reply = "";
371    try {
372      while ((skip-- > 0) && !(reply=reader.readLine()).startsWith("."))
373        count++;
374      if (!reply.startsWith(".")) {
375        String row[];
376        while (err == ERR_NOERR && !(reply=reader.readLine()).startsWith(".") && (max--
   > 0)) {
377          count++;
378          row = split(reply, " ");
379          if (getFirst) {
380            this.group = row[0];
381            err = lasso.addResultRow(row);
382            getFirst = false;
383          } else
384            err = lasso.addResultRow(row);
385        }
386        if (this.groupCount != -1) {
387          count = this.groupCount;
388          this.sock.close();
389          this.sock = null;
390        } else if (!reply.startsWith(".")) {
391          while (!(reader.readLine()).startsWith("."))
392            count++;
393          this.groupCount = count;
394        }
395      }
396    } catch (Exception e) { System.err.println(e.toString()); }
397    err = lasso.setNumRowsFound(count);
398    return err;
399  }
400  int selectGroup(LassoCall lasso) throws java.io.IOException {
401    this.printer.print("GROUP " + this.group + "\r\n");
402    String reply=reader.readLine();
```

```
403   if (!reply.startsWith("2"))
404     return setError(lasso, reply);
405   else
406     return ERR_NOERR;
407 }
408 String getRange(LassoCall lasso, int skip, int max) {
409   this.printer.print("LISTGROUP " + this.group + "\r\n");
410   StringBuffer result = new StringBuffer();
411   int count = 0;
412   try {
413     String reply=reader.readLine();
414     if (reply.startsWith("2")) {
415       String last = "";
416       while (!(reply=reader.readLine()).startsWith(".") && (skip-- > 0))
417         count++;
418       if (!reply.startsWith(".")) {
419         result.append(reply);
420         if (max != 1)
421           result.append("-");
422         while (!(reply=reader.readLine()).startsWith(".") && (--max > 0)) {
423           count++;
424           last = reply;
425         }
426         if (this.articleCount > -1) {
427           this.printer.println("QUIT\r\n");
428           this.sock.close();
429           this.sock = null;
430           count = this.articleCount;
431           if (connect(lasso))
432             selectGroup(lasso);
433         } else if (!reply.startsWith(".")) {
434           while (!(reader.readLine()).startsWith("."))
435             count++;
436           result.append(last);
437           this.articleCount = count;
438         }
439       }
440     }
441   } catch (Exception e) {} ;
442   return result.toString();
443 }
444 }
```

Data Source Connector Walk-Through

This section provides a step-by-step walk-through for building the described data source connector.

To build a sample LJAPI Data Source Connector:

1 First, import com.blueworld.lassopro.* classes as shown in line 1.

```
1  import com.blueworld.lassopro.*;
2  import java.net.*;
3  import java.io.*;
4  import java.util.*;
```

2 Define your module to be a subclass of the com.blueworld.lassopro.LassoDSM odule class:

```
6  public class NNTP_DS extends LassoDSModule {
```

Define the storage for global variables, which are objects used to communicate with an NNTP server, authentication and server info, etc.

```
7  Socket sock;
8  PrintStream printer;
9  BufferedReader reader;
...
```

3 Define the registerLassoModule method.

```
17  public void registerLassoModule() {
```

Every Lasso module must implement the registerLassoModule() method. This method will be called by Lasso at startup, giving your module a chance to register its data source(s).

4 Define your main data source method. This function gets called with various actions as Lasso Professional requests information from the data source. The method name should be identical to the string passed in the second parameter of the registerLassoModule() method.

```
28  public int dsFunc (LassoCall lasso, int cmd, LassoValue value)
```

5 Dispatch the action to corresponding Java method implemented in the module. The switch statement distinguishes between various actions. For a complete list of action constant values, see the LassoDSModule class reference.

```
30  switch (cmd) {
31    case ACTION_INIT:
32      err = doInit(lasso);
33      break;
34    case ACTION_TERM:
35      err = doTerm(lasso);
          break;
```

6 Among various actions that can be performed by a data source module the, ACTION_EXISTS command is sent by Lasso Professional to verify that a particular database exists on a specific host. If the name of the database being looked up is not known, the module must return a LassoErrors .WebNoSuchObject error:

```
37    case ACTION_EXISTS:
38      if (!value.data().equalsIgnoreCase("News"))
39        err = LassoErrors.WebNoSuchObject;
40      break;
```

7 Return the ERR_NOERR result code upon successful completion of the task. Returning a non-zero value will cause the Lasso Professional engine to report a fatal error and stop processing the page.

```
54  return err;
```

8 After successful registration, every data source module receives the ACTION_INIT command, which gives it a chance to establish connection with a data source or perform any other initialization tasks. Our module simply returns ERR_NOERR result code:

```
56  int doInit(LassoCall lasso) {
57    return ERR_NOERR;
58  }
```

9 Similarly, Lasso sends the ACTION_TERM command to all registered data source modules during its shutdown sequence. The sample data source uses this as a signal to close the connection to a NNTP server and perform additional clean-up tasks:

```
59  int doTerm(LassoCall lasso) {
60    close();
61    return ERR_NOERR;
62  }
```

10 The ACTION_DB_NAMES command is sent whenever Lasso Professional needs to get a list of databases which the data source provides access to. The developer must write code that discovers all the databases the module knows of, and call LassoCall.addDataSourceResult() once for each database it encounters:

```
65  int doDBNames(LassoCall lasso) {
66    return lasso.addDataSourceResult("News");
67  }
```

11 Whenever a data source module receives the ACTION_TABLE_NAMES command, it must examine the database name passed in the LassoValue parameter, and return the names of all tables available in the specified database:

```
68  int doTableNames(LassoCall lasso, String db) {
69    if (!db.equalsIgnoreCase("News"))
70      return -1;
71    lasso.addDataSourceResult("Groups");
72    lasso.addDataSourceResult("Articles");
73    return ERR_NOERR;
74  }
```

12 Lasso Professional sends the ACTION_INFO command when it needs to retrieve the information about columns contained in the result set. Inline tag actions like -FindAll and -Search usually return a result set containing certain number of rows/records, each consisting of one or more columns/fields. When data source module receives an ACTION_INFO command, it must call LassoCall.addColumnInfo() method once for each column stored in the result set.

```
73  int doInfo(LassoCall lasso, boolean listAllCols) {
74    int err = ERR_NOERR;
75    LassoValue tbl = new LassoValue();
76    err = lasso.getTableName(tbl);
77    if (err != ERR_NOERR || tbl.data().length() == 0)
78      return LassoErrors.InvalidParameter;
79       if (!connect(lasso))
80      return LassoErrors.Network;
81    if (tbl.data().equalsIgnoreCase("Groups")) {
82      lasso.addColumnInfo("Group", 0,
            LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
83      lasso.addColumnInfo("Last", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
84      lasso.addColumnInfo("First", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
85      lasso.addColumnInfo("AllowPost", 0,
            LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
86    } ...
```

13 The ACTION_SEARCH command is sent whenever Lasso Professional needs to perform the search action on a data source.

```
109  int doSearch(LassoCall lasso) {
```

14 All of the information about the current search parameters (database and table names, search arguments, sort arguments, etc.) can be retrieved by calling various LJAPI methods such as LassoCall.getDataSourceName() and LassoCall.getTableName(). Similarly, one can call getSkipRows() and getMaxRows() methods to retrieve the -SkipRecords and -MaxRecords inline parameter values. For a complete list of available methods, see LassoCall class reference.

```
117  IntValue ival = new IntValue();
118  if (lasso.getSkipRows(ival) == ERR_NOERR)
119    skip = ival.intValue();
120  if (lasso.getMaxRows(ival) == ERR_NOERR)
121    max = ival.intValue();
122  lasso.getTableName(tbl);
123  lasso.getInputColumnCount(ival);
```

15 The module needs to perform different actions depending on the search table name.

```
129  if (tbl.data().equalsIgnoreCase("GROUPS")) {
```

16 Some NNTP servers allow retrieval of newsgroup listings filtered by a matching pattern. The module builds the pattern string based on the value of the inline search operator (beginsWith, endsWith, etc.).

```
130  if (lasso.findInputColumn("group", val) == ERR_NOERR) {
131    if (val.type() == LassoOperators.OP_ENDS_WITH)
132      filter = '*' + val.data();
133    else if (val.type() == LassoOperators.OP_CONTAINS)
134      filter = '*' + val.data() + '*';
135    else if (val.type() == LassoOperators.OP_EQUALS)
136      filter = val.data();
137    else
138      filter = val.data() + '*';
139  }
```

17 In case the search is being performed on the ARTICLES table, we need to find out the name of a newsgroup before we can proceed any further.

```
149  } else if (tbl.data().equalsIgnoreCase("ARTICLES")) {
150    if (lasso.findInputColumn("-group", val) == ERR_NOERR ||
151    lasso.findInputColumn("group", val) == ERR_NOERR) {
```

18 Next, we check if an article number or message ID has been included in the search criteria, either as a primary keyfield, record ID, or as a named search field.

```
164  if (lasso.getRowID(ival) == ERR_NOERR && ival.intValue() != -1)
165    id = Integer.toString(ival.intValue());
166  else if (lasso.getPrimaryKeyColumn(val) == ERR_NOERR &&
167        (val.name().equalsIgnoreCase("message-id") ||
168         val.name().equalsIgnoreCase("number")))
169    filter = val.data();
170  else if (lasso.findInputColumn("message-id", val) == ERR_NOERR ||
171          lasso.findInputColumn("number", val) == ERR_NOERR)
172  filter = val.data();
```

19 If none of the above was found, yet the -MaxRecords inline parameter appears to limit the query results to a single row, we can try finding the desired article ID based on the current -SkipRecords value.

```
178  if (max == 1 && (filter == null || filter.length() < 1))
179    id = getRange(lasso, skip, 1);
```

20 If the article has been identified, proceed with retrieving the message in its entirety.

```
182  if (id != null && id.length() > 1) { // detail
183    this.printer.print("ARTICLE " + id + "\r\n");
184    reply=reader.readLine();
```

21 Otherwise, select the next group of news articles and retrieve their headers.

```
225  if (filter == null || filter.length() == 0)
226    filter = getRange(lasso, skip, max);
227  this.printer.print("XOVER " + filter + "\r\n");
228  reply=reader.readLine();
229  if (!reply.startsWith("2"))
230    return setError(lasso, reply);
```

22 The LassoCall.addResultRow() method is used to return the results of a data source action. It should be called as many times as there are records in the result set, once for each record.

LassoCall.addResultRow() method takes a single String array parameter. Each array element corresponds to a record column/field contained in the result set. The total number of array elements must be equal to the number of times LassoCall.addColumnInfo() method was called for this data source action. Since news article headers are transmitted in the form of a tab-delimited string, we use our custom split() method to convert the data to a String array, suitable for passing to addResultRow() method:

```
231  while(err == ERR_NOERR && !(reply=reader.readLine()).startsWith("."))
232    err = lasso.addResultRow(split(reply, "\t"));
```

23 Finally, implement a number of convenience methods, including the setError() routine used for standard error handling:

```
246  int setError(LassoCall lasso, String reply) {
247    int err = -1;
248    try {
249      err = Integer.parseInt(reply.substring(0, 3));
250      lasso.setResultMessage(reply.substring(4));
251    } catch (Exception e) {};
252    lasso.setResultCode(err);
253    return err;
254  }
```

Data Type Operation

Among other new features, Lasso Professional 7 Java API introduces the ability to create custom data types in Java. Creating a new data type in LJAPI 7 is similar to creating a substitution tag. When Lasso Professional 7 starts up, it scans the LassoModules folder for module files (Windows DLLs or Mac OS X DYLIBS). As it encounters each module, it executes the registerLassoModule() function for that module. The developer may register new LJAPI data types implemented by the module inside this function.

Custom data types are analogous to objects used in many other programming languages. They can have properties (fields) and member tags (methods).

Data Type Tutorial

The following section provides a walk-through of building an example custom type to show how LJAPI features are used. This code will be most similar to the sample ZipType LJAPI project, so in order to build this code, copy the ZipType project folder and edit the project files inside it.

The module relies on a Java class library to do most of the work, particularly the java.util.zip package which provides variety of functions for manipulating the contents of ZIP files—standard compressed archives widely used on the Internet.

The resulting type will be a "zip" file with the ability to read data from a zip file given a path. The following member tags will be implemented:

Table 1: Type initializer and Member Tags

Name	Description
[Zip:'Pathname']	Type initializer. Creates new instance of a custom type.
[Zip->File]	Return the name of this Zip file.
[Zip->Count]	Return the count of entries in this file.
[Zip->Size]	Synonym for [Zip->Count].
[Zip->Enumerate]	Enumerates zip entries, allowing to iterate through stored items via consecutive calls to [Zip->Next].
[Zip->Next]	Advance to the next entry, returning True if more items are available.
[Zip->Position]	Current iterator position, i.e. the index.

The rest of the member tags are item accessors, operating on the entries stored in a zip file:

Table 2: Accessors

Name	Description
[Zip->Name]	Returns the name of an indexed entry.
[Zip->Get]	Synonym for [Zip->Name].
[Zip->Comment]	Zip entry comment.
[Zip->Date]	Returns the entry creation date.
[Zip->Crc]	Checksum, or 0xffffffff if not available.
[Zip->Method]	Compression method: DEFLATED or STORED.
[Zip->Extra]	Returns any extra data stored with the entry.
[Zip->GetData]	Returns uncompressed entry data.
[Zip->CSize]	Returns the size of the compressed data.
[Zip->USize]	Returns the size of uncompressed data.
[Zip->IsDir]	Returns True if the entry is a directory.

All zip entry accessor tags, except for [Zip->GetData], can take either one or zero parameters. An integer parameter can specify the index (position) of the entry in a zip file, while a string parameter could be used to locate an entry by its name. When no parameters are provided, a corresponding action is performed on the "current" item, whose index can be obtained via the [Zip->Position] member tag.

Example sample tag LDML syntax:

The following shows an example of using a Zip custom type.

```
[Var:'zip' = zip:'/archive.zip']
[$zip->Count]
[$zip->Method]
[$zip->CSize]
[$zip->USize]
[While: $zip->Next]
  [$zip->CRC]
[/While]
```

Custom Data Type Module Code

Shown below is the code for the custom type tag module. Line numbers are provided to the left of each line of code, and are referenced in the *Custom Type Tag Module Walk-Through* section.

Note: The line numbers shown refer to the line numbers of the code in the actual file being created, not as shown in this page. Some single lines of code may carry into two or more lines as shown on this page.

Custom Data Type Module Code

```
1  import com.blueworld.lassopro.*;
2  import java.util.*;
3  import java.util.zip.*;
4  import java.io.*;
5  import java.text.DateFormat;
6  public class ZipType extends LassoTagModule
7  {
8  static final DateFormat df =
9    DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.MEDIUM);
10 static final String[] members = {
11   "File","Size","Count","Enumerate","Position","Next",
12   "GetData","Get","Name","Comment","Date","Crc",
13   "Method", "Extra", "CSize", "USize", "IsDir" };
14 ZipFile    zip = null;
15 Enumeration enum = null;
16 ZipEntry    entry = null;
17 int     index = 0;
18 public void registerLassoModule()
19 {
20   registerTagModule("ZipType", "zip", "format",
21         FLAG_SUBSTITUTION | FLAG_INITIALIZER, "zip custom type tag");
22 }
23 public int format(LassoCall lasso, int action)
24 {
25   int err = ERR_NOERR;
26   LassoValue param = new LassoValue();
27   String path name = null;
28   if (lasso.getTagParam(0, param) != ERR_NOERR ||
29      param.name().length() < 1)
30   {
31     lasso.setResultMessage("[Zip] invalid file path name parameter");
32     return LassoErrors.InvalidParameter;
33   }
34   try
35   {
36     IntValue count = new IntValue();
37     err = lasso.getTagParamCount(count);
38     if (err == ERR_NOERR && count.intValue() > 0)
39     {
40       String filePath = lasso.fullyQualifyPath(param.name());
41       filePath = lasso.resolvePath(filePath);
42       filePath = lasso.getPlatformSpecificPath(filePath);
43       this.zip = new ZipFile(filePath);
44       LassoTypeRef self = new LassoTypeRef();
45       if ((err = lasso.typeAllocCustom(self, "zip")) != ERR_NOERR)
46       {
47         lasso.setResultMessage("[Zip] couldn't create new zip type instance.");
```

```
48            return err;
49          }
50          LassoTypeRef ref = new LassoTypeRef();
51          String className = this.getClass().getName();
52          for (int i = 0; i < this.members.length; i++)
53          {
54            if ((err=lasso.typeAllocTag(ref, className, “memberFunc”)) != ERR_NOERR ||
55                (err=lasso.typeAddMember(self, members[i], ref)) != ERR_NOERR)
56            {
57              lasso.setResultMessage("[Zip] error adding member: “ + members[i]);
58              return err;
59            }
60          }
61          if (lasso.typeAllocTag(ref, className, “convertFunc”) == ERR_NOERR)
62            lasso.typeAddMember(self, “onConvert”, ref);
63          if (lasso.typeAllocTag(ref, className, “destroyFunc”) == ERR_NOERR)
64            lasso.typeAddMember(self, “onDestroy”, ref);
65          if ((err = lasso.typeSetCustomJavaObject(self, this)) != ERR_NOERR)
66          {
67            lasso.setResultMessage("[Zip] couldn’t attach java object to a custom type");
68            return err;
69          }
70          err = lasso.returnTagValue(self);
71        }
72      }
73      catch (Exception e)
74      {
75        System.err.println(e.toString());
76        lasso.setResultMessage(e.getMessage());
77        return LassoErrors.FileNotFound;
78      }
79      return err;
80    }
81    public int destroyFunc(LassoCall lasso, int action)
82    {
83      if (this.zip != null)
84      {
85        try { this.zip.close(); }
86          catch (IOException e) {}
87        this.zip = null;
88      }
89      return ERR_NOERR;
90    }
91    public int convertFunc(LassoCall lasso, int action)
92    {
93      LassoValue param = new LassoValue();
94      if (lasso.getTagParam(0, param) == ERR_NOERR &&
95        param.name().equalsIgnoreCase(“string”))
```

```
96    {
97      lasso.outputTagData("zip:(" + this.zip.getName() + ")");
98    }
99
100 return ERR_NOERR;
101 }
102 public int memberFunc(LassoCall lasso, int action)
103 {
104 LassoValue tag = new LassoValue();
105 LassoTypeRef out = new LassoTypeRef();
106 int err = lasso.getTagName(tag);
107 if (err != ERR_NOERR || tag.data().length() < 1)
108   return LassoErrors.InvalidParameter;
109 if (tag.data().equalsIgnoreCase("file"))
110   return lasso.outputTagData(zip.getName());
111 else if (tag.data().equalsIgnoreCase("size") ||
112       tag.data().equalsIgnoreCase("count"))
113 {
114   lasso.typeAllocInteger(out, zip.size());
115   return lasso.returnTagValue(out);
116 }
117 LassoValue param = new LassoValue();
118 ZipEntry item = this.entry;
119 if (lasso.getTagParam(0, param) == ERR_NOERR)
120 {
121   if (param.type() == LassoValue.TYPE_INT)
122   {
123     try {
124       int idx = Integer.parseInt(param.name());
125       if (idx < 1 || idx > zip.size())
126       {
127         lasso.setResultMessage("[Zip] index out of range: " + idx);
128         return LassoErrors.InvalidParameter;
129       }
130       else if (idx != index)
131       {
132         index = idx;
133         Enumeration enum2 = zip.entries();
134         while (enum2.hasMoreElements() && idx-- > 0)
135           item = (ZipEntry)enum2.nextElement();
136         entry = item;
137       }
138     } catch (NumberFormatException npe) {}
139   }
140   else if (param.type() == LassoValue.TYPE_CHAR)
141     item = zip.getEntry(param.name());
142 }
143 String result = null;
```

```
144 if (tag.data().equalsIgnoreCase("name") ||
145     tag.data().equalsIgnoreCase("get"))
146   result = item.getName();
147 else if (tag.data().equalsIgnoreCase("comment"))
148   result = item.getComment();
149 else if (tag.data().equalsIgnoreCase("crc"))
150   result = Long.toHexString(item.getCrc());
151 else if (tag.data().equalsIgnoreCase("method"))
152   result = (item.getMethod() == ZipEntry.DEFLATED ? "DEFLATED" : "STORED");
153 if (result != null)
154   return lasso.outputTagData(result);
155 if (tag.data().equalsIgnoreCase("usize"))
156   lasso.typeAllocInteger(out, item.getSize());
157 else if (tag.data().equalsIgnoreCase("csize"))
158   lasso.typeAllocInteger(out, item.getCompressedSize());
159 else if (tag.data().equalsIgnoreCase("date"))
160   lasso.typeAllocString(out, df.format(new Date(item.getTime())));
161 else if (tag.data().equalsIgnoreCase("isDir"))
162   lasso.typeAllocBoolean(out, entry.isDirectory());
163 else if (tag.data().equalsIgnoreCase("position"))
164   lasso.typeAllocInteger(out, index);
165 else if (tag.data().equalsIgnoreCase("enumerate"))
166 {
167   enum = zip.entries();
168   index = 0;
169 }
170 else if (tag.data().equalsIgnoreCase("getdata"))
171 {
172   int max = 0, skip = 0;
173
174   if (lasso.findTagParam("-skip", param) == ERR_NOERR)
175     skip = Integer.parseInt(param.data());
176   if (lasso.findTagParam("-max", param) == ERR_NOERR)
177     max = Integer.parseInt(param.data());
178   int count = 0;
179   int toRead = 1024;
180   if (max == 0 || max > item.getSize())
181     max = (int)item.getSize() - skip;
182   else if (max < 1024)
183     toRead = max;
184   try {
185     InputStream is = zip.getInputStream(item);
186     is.skip(skip);
187     byte b[] = new byte[toRead];
188     while ((count=is.read(b, 0, toRead)) > -1 && max > 0)
189     {
190       max -= count;
191       if (count > 0)
```

```
192             lasso.outputTagData(new String(b, 0, count));
193         }
194         is.close();
195       } catch (IOException ioe) {}
196     }
197     else if (tag.data().equalsIgnoreCase("next"))
198     {
199       boolean reset = (enum == null);
200       if (enum != null && !enum.hasMoreElements())
201         enum = null;
202       else if (reset)
203         enum = zip.entries();
204       boolean hasMore = (enum != null && enum.hasMoreElements());
205       lasso.typeAllocBoolean(out, hasMore);
206       if (hasMore)
207       {
208         entry = (ZipEntry)enum.nextElement();
209         if (reset)
210           index = 1;
211         else
212           index++;
213       }
214     }
215     if (!out.isNull())
216       return lasso.returnTagValue(out);
217     return err;
218   }
```

Custom Data Type Module Walk-Through

This section provides a step-by-step walk-through for building the custom type tag module.

To write a sample LJAPI tag module:

1 First, import com.blueworld.lassopro.* classes as shown in line 1.

```
1  import com.blueworld.lassopro.*;
2  import java.util.*;
3  import java.util.zip.*;
4  import java.io.*;
5  import java.text.DateFormat;
```

2 Define the class to be a subclass of the com.blueworld.lassopro.LassoTagModule class.

```
6  public class ZipType extends LassoTagModule
```

3 Store the names of member tags implemented by our custom type in a String array variable.

```
10 static final String[] members = {
11   "File","Size","Count","Enumerate","Position","Next",
12   "GetData","Get","Name","Comment","Date","Crc",
13   "Method", "Extra", "CSize", "USize", "IsDir" };
```

4 Register the custom type initializer method, passing FLAG_INITIALIZER flag in the fourth parameter of the registerLassoModule method.

```
18 public void registerLassoModule()
19 {
20   registerTagModule("ZipType", "zip", "format",
21         FLAG_SUBSTITUTION | FLAG_INITIALIZER, "zip custom type tag");
22 }
```

5 Define main tag formatting method with the same name as specified in the third parameter of previously called registerTagModule method.

```
23 public int format(LassoCall lasso, int action)
```

6 Examine parameters passed to our type initializer and create new instance of a java.util.zip.ZipFile object, using resolved file path name.

```
40 String filePath = lasso.fullyQualifyPath(param.name());
41 filePath = lasso.resolvePath(filePath);
42 filePath = lasso.getPlatformSpecificPath(filePath);
43 this.zip = new ZipFile(filePath);
```

7 Create a new com.blueworld.lassopro.LassoTypeRef variable to store a reference to the custom type, which is about to be created in the next step.

```
44 LassoTypeRef self = new LassoTypeRef();
```

8 Allocate new custom type instance, passing the LassoTypeRef variable and the type name to LassoCall.typeAllocCustom method.

```
45 if ((err = lasso.typeAllocCustom(self, "zip")) != ERR_NOERR)
46 {
47   lasso.setResultMessage("[Zip] couldn't create new zip type instance.");
48   return err;
49 }
```

9 Add member tags to the newly-allocated custom type. In our example, all member tags will be handled by the same Java method; however, LJAPI allows each member tag to have its own formatting method.

```
52 for (int i = 0; i < this.members.length; i++)
53 {
54   if ((err=lasso.typeAllocTag(ref, className, "memberFunc")) != ERR_NOERR ||
55       (err=lasso.typeAddMember(self, members[i], ref)) != ERR_NOERR)
56   {
57     lasso.setResultMessage("[Zip] error adding member: " + members[i]);
58     return err;
59   }
60 }
```

Note that adding the member tags to a custom type is a two-step process. First, an unnamed tag object is created and placed in a LassoTypeRef variable. In order to be successful, the second and third parameters in the LassoCall.typeAllocTag method must specify a valid class and method names used by Lasso for locating a formatting method in a Java class. Member tag methods have the same signature as a type initializer and regular substitution tag methods, and although not required they are most likely to be implemented in the same class with the main type initializer method.

Secondly, LassoCall.typeAddMember is used to add a reference to a newly-created tag (third parameter) to a custom type (first parameter), with the second parameter being a tag name.

10 Add all necessary callback methods, such as onConvert and onDestroy.

```
61 if (lasso.typeAllocTag(ref, className, “convertFunc”) == ERR_NOERR)
62   lasso.typeAddMember(self, “onConvert”, ref);
63 if (lasso.typeAllocTag(ref, className, “destroyFunc”) == ERR_NOERR)
64   lasso.typeAddMember(self, “onDestroy”, ref);
```

Callback methods are being triggered by the events that happen to a custom type in the course of its life. For example, when a type goes out of scope, its onDestroy tag method is called. When a custom type needs to be converted to a different data type such as string or integer, its onConvert method is invoked.

Callbacks are added to the custom types in a similar fashion as the other members, with only constraint being their tag names, which must conform to established convention for naming callback tags. For a full list of intrinsic member tag names, see the Lasso 7 Language Guide.

11 Attach this module instance to a custom type.

```
65 if ((err = lasso.typeSetCustomJavaObject(self, this)) != ERR_NOERR)
66 {
67   lasso.setResultMessage(“[Zip] couldn’t attach java object to a custom type”);
68   return err;
69 }
```

LassoCall.typeSetCustomJavaObject can be used to associate any private data with an instance of a custom type. Any Java object can be attached to a custom type and later retrieved with a call to a complimentary LassoCall.typeGetCustomJavaObject method. In the situation were associated object is an instance of the LassoTagModule subclass, Lasso will also try to invoke formatting methods on this object instead of creating a new instance (as it does for all substitution tag modules). Aside from producing much smaller overhead, this allows direct access to all instance (e.g. private) variables from any Java method implemented in that module.

12 Finally, return newly-generated custom type tag instance back to Lasso.

```
70  err = lasso.returnTagValue(self);
```

13 Implement formatting methods for onDestroy and onConvert callbacks.

```
81  public int destroyFunc(LassoCall lasso, int action)
82  {
83    if (zip != null)
84    {
85      try { zip.close(); }
86        catch (IOException e) {}
87      zip = null;
88    }
89    return ERR_NOERR;
90  }
```

14 In the case of onConvert callback, the first parameter passed to our method is the name of the type to which our custom Zip type should be converted to. If the desired type is a string, return the human-readable representation of the type, which consists of a type name and a zip file path name.

```
91  public int convertFunc(LassoCall lasso, int action)
92  {
93    LassoValue param = new LassoValue();
94    if (lasso.getTagParam(0, param) == ERR_NOERR &&
95      param.name().equalsIgnoreCase("string"))
96    {
97      lasso.outputTagData("zip:(" + this.zip.getName() + ")");
98    }
99
100 return ERR_NOERR;
101 }
```

15 Define our main member tag method memberFunc, that will take care of formatting over a dozen member tags. If tag name is File, return the full path name to the zip file.

```
109   if (tag.data().equalsIgnoreCase("File"))
110 return lasso.outputTagData(zip.getName());
```

16 If the member tag name is Count or Size, return an integer Zip entry count value.

```
111   else if (tag.data().equalsIgnoreCase("size") ||
112       tag.data().equalsIgnoreCase("count"))
113   {
114 lasso.typeAllocInteger(out, zip.size());
115 return lasso.returnTagValue(out);
116   }
```

17 Tags that output plain text can be processed first.

```
143   String result = null;
144   if (tag.data().equalsIgnoreCase("name") ||
```

```
145  tag.data().equalsIgnoreCase("get"))
146  result = item.getName();
147   else if (tag.data().equalsIgnoreCase("comment"))
148  result = item.getComment();
149   else if (tag.data().equalsIgnoreCase("crc"))
150  result = Long.toHexString(item.getCrc());
151   else if (tag.data().equalsIgnoreCase("method"))
152  result = (item.getMethod() == ZipEntry.DEFLATED ? "DEFLATED" : "STORED");
153   if (result != null)
154  return lasso.outputTagData(result);
```

18 Tags that return data types, such as integers or booleans, should allocate corresponding values using various LassoCall.typeAlloc... methods before passing them back to Lasso.

```
155   if (tag.data().equalsIgnoreCase("usize"))
156  lasso.typeAllocInteger(out, item.getSize());
157   else if (tag.data().equalsIgnoreCase("csize"))
158  lasso.typeAllocInteger(out, item.getCompressedSize());
159   else if (tag.data().equalsIgnoreCase("date"))
160  lasso.typeAllocString(out, df.format(new Date(item.getTime())));
161   else if (tag.data().equalsIgnoreCase("isDir"))
162  lasso.typeAllocBoolean(out, entry.isDirectory());
163   else if (tag.data().equalsIgnoreCase("position"))
164  lasso.typeAllocInteger(out, index);
```

19 The Enumerate member restarts a previously used enumeration. Unless it is called in a middle of iterating through the Zip entries, this tag has the same effect as calling Next for the very first time, or immediately after advancing past the very last enumerated item in a Zip file.

```
165   else if (tag.data().equalsIgnoreCase("enumerate"))
166   {
167  enum = zip.entries();
168  index = 0;
169   }
```

20 The GetData member tag reads uncompressed data from one of the zipped items. This tag accepts two optional parameters, -Skip and -Max, which are used to specify starting offset and maximum number of bytes to be read from the Zip archive entry.

```
170   else if (tag.data().equalsIgnoreCase("getdata"))
171   {
172  int max = 0;
173  int skip = 0;
174  if (lasso.findTagParam("-skip", param) == ERR_NOERR)
175    skip = Integer.parseInt(param.data());
176  if (lasso.findTagParam("-max", param) == ERR_NOERR)
177    max = Integer.parseInt(param.data());
178  int count = 0;
```

```
179  int toRead = 1024;
180  if (max == 0 || max > item.getSize())
181    max = (int)item.getSize() - skip;
182  else if (max < 1024)
183    toRead = max;
184  try {
185    InputStream is = zip.getInputStream(item);
186    is.skip(skip);
187    byte b[] = new byte[toRead];
188    while ((count=is.read(b, 0, toRead)) > -1 && max > 0)
189    {
190      max -= count;
191      if (count > 0)
192        lasso.outputTagData(new String(b, 0, count));
193    }
194    is.close();
195  } catch (IOException ioe) {}
196   }
```

21 The last member tag Next iterates through Zip archive entries, placing the internally maintained pointer at the next selected item. This tag provides fast sequential access to items stored in the Zip archive, and should be used in concert with various accessor tags implemented in this module. When the end of the file is reached and no more items are available, the tag returns False and restarts the iteration, positioning the internal pointer immediately before the first Zip item.

```
197   else if (tag.data().equalsIgnoreCase("next"))
198   {
199  boolean reset = (enum == null);
200  if (enum != null && !enum.hasMoreElements())
201    enum = null;
202  else if (reset)
203    enum = zip.entries();
204  boolean hasMore = (enum != null && enum.hasMoreElements());
205  lasso.typeAllocBoolean(out, hasMore);
206  if (hasMore)
207  {
208    entry = (ZipEntry)enum.nextElement();
209    if (reset)
210      index = 1;
211    else
212      index++;
213  }
214   }
```

22 Finally, if any of the previous operations produced a valid result, pass the resulting value back to Lasso, returning an ERR_NOERR error code to flag a successful member tag execution.

```
215  if (!out.isNull())
216 return lasso.returnTagValue(out);
217  return err;
```

LJAPI Interface Reference

This section provides a listing of all Java interfaces available for use in LJAPI 7. All variables, constructors, and methods for each interface are organized by category under each interface name.

com.blueworld.lassopro.LassoJavaModule

This is the base interface implemented by both substitution tag and data source LJAPI modules. Upon Lasso Service startup, the registerLassoModule method is called for every Java module located inside the LassoModules folder. Each module returns information about their name, implemented tags or data sources, method names, etc.

Data source modules are instantiated only once and then used repeatedly to perform various data source actions. Tag modules are instantiated every time Lasso resolves a tag implemented by a LassoTagModule.

Methods

registerLassoModule()

This method must be defined in all LJAPI modules. Lasso calls this once at startup to allow a module to register its tags or data sources.

```
public void registerLassoModule ( );
```

Variables

ERR_NOERR

On success, every method must return ERR_NOERR result code.

```
public static final int ERR_NOERR
```

LJAPI Class Reference

This section lists all the Java classes available for use in LJAPI 7. All variables, constructors, and methods for each interface are organized alphabetically under each interface name, unless specified otherwise.

com.blueworld.lassopro.FloatValue

Wrapper class for a primitive float or double type. Used for returning decimal values from the LassoCall.typeGetDecimal method.

Constructors

public FloatValue()
public FloatValue(float value)
public FloatValue(double value)

Methods

doubleValue()

Returns the value of a FloatValue object as a double.

public double doubleValue()

floatValue()

Returns the value of a FloatValue object as a float.

public float floatValue()

toString()

Converts an object to a string. Overrides toString() method in class Object.

public String toString()

com.blueworld.lassopro.IntValue

Wrapper for primitive integer types. Used for returning values from LassoCall methods, which in C would require passing the pointer-type parameters: int*, long* and LP_TypeDesc*. In addition, this class provides methods for converting a 4-byte int (LP_TypeDesc type in LCAPI) to a String and back.

Constructors

public IntValue()
public IntValue(int value)
public IntValue(long value)

Methods

byteValue()

Returns the value of an IntValue object as a 1-byte integer.

public byte byteValue()

shortValue()

Returns the value of an IntValue object as a 2-byte integer.

```
public short shortValue()
```

intValue()

Returns the value of an IntValue object as a 4-byte integer.

```
public int intValue()
```

longValue()

Returns the value of an IntValue object as an 8-byte integer.

```
public long longValue()
```

setByte()

Sets the value of an IntValue object to a 1-byte integer.

```
public void setByte( byte value )
```

setShort()

Sets the value of an IntValue object to a 2-byte integer.

```
public void setShort( short value )
```

setInt()

Sets the value of an IntValue object to a 4-byte integer.

```
public void setInt( int value )
```

setLong()

Sets the value of an IntValue object to an 8-byte integer.

```
public void setLong( long value )
```

toDescType()

Converts the lower 4 bytes of an IntValue value to a 4-char String.

```
public String toDescType()
```

toString()

Converts an object to a string. Overrides toString() method in class Object.

```
public String toString()
```

IntToFourCharString()

Static method used for converting an int to a 4-char String.

```
public static String IntToFourCharString( int value )
```

com.blueworld.lassopro.LassoCall

Of all Java classes listed in this section, the LassoCall class is of the utmost importance. All the interaction between an LJAPI module and Lasso

Professional 7 is achieved by means of invoking various methods implemented in the LassoCall class. These functions can be used to do any of the following: register your tags or data sources, allocate memory, return error messages, get tag or parameter information, get client/server environment information, output text, read/set MIME headers, access LDML variables, interpret/execute arbitrary LDML tags, store persistent data, check if the user is an administrator, perform data source functions, and safely access multiuser/multithreaded resources.

All class methods in this section are listed by their category.

Internal Value Methods

getLassoParam()

Fetches an internal server value such as path to LassoModules folder, name of the Lasso error log file, etc. For a full list of available parameters, please see the listing of constants defined in the LassoParams class.

```
public int getRequestParam( int key, LassoValue outResult );
```

getRequestParam()

Fetches an HTTP request value such as server port, cookies, root path, username, etc. For a full list of available parameters, please see the listing of constants defined in the LassoRequestParams class. Please note that some of these parameters may not be available on all HTTP servers.

```
public int getRequestParam( int key, LassoValue outResult );
```

Error Messages and Result Code Methods

setResultCode()

Sets the result code that can be displayed if the LDML programmer inserts [Error_CurrentError: -ErrorCode] into the format file after executing a custom LJAPI tag.

```
public int setResultCode( int err );
```

setResultMessage()

Sets the error message that can be displayed if the LDML programmer inserts [Error_CurrentError: -ErrorMessage] into the format file after executing a custom LJAPI tag.

```
public int setResultMessage( String msg );
```

Tag and Parameter Info Methods

getTagName()

Fetches the name of the tag that triggered this call (e.g. in the case of [my_tag: ...] the resulting value would be my_tag). This makes it possible to design a single tag function which can perform the duties of many different LDML tags, perhaps ones that all have similar functionality but different names.

```
public int getTagName( LassoValue result );
```

getTagParamCount()

Fetches the number of parameters that were passed to the tag. For instance, [my_tag: 'hello', -option=1, -hilite=false] will report that three parameters were passed (unnamed parameters are treated just like any other parameter).

```
public int getTagParamCount( IntValue result );
```

getTagParam()

Gets the name and value of a parameter given its index. Parameters are numbered left-to-right, starting at index 0: [my_tag: -param0='value0', -param1='value1', -param2=2].

```
public int getTagParam( int paramIndex, LassoValue result );
```

getTagParam2()

Get the parameter using the parameter index. This function differs from getTagParam() in that it preserves the actual type of the parameter instead of automatically converting it to a string. Keyword/value pairs are returned as a LASSO_PAIR type.

```
public int getTagParam2( int paramIndex, LassoTypeRef outValue );
```

tagParamIsDefined()

Returns ERR_NOERR if the parameter was defined. Otherwise, the parameter wasn't defined.

```
public int tagParamIsDefined( String paramName );
```

findTagParam()

Finds and fetches a tag parameter by name. A return value of ERR_NOERR means the parameter was found successfully.

```
public int findTagParam( String paramName, LassoValue result );
```

findTagParam2()

Finds and returns a tag parameter by name while preserving the original type. A returned value of ERR_NOERR means the parameter was successfully found.

```
public int findTagParam2( String paramName, LassoTypeRef outValue );
```

getTagEncoding()

Fetches the encoding method indicated for this tag. This is rarely used, because Lasso handles encoding and decoding for you.

```
public int getTagEncoding( IntValue method );
```

childrenRun()

Used to execute the contents of a container tag. Tags become containers when the FLAG_CONTAINER flag is used. The result parameter will contain the combined result data for all tags contained.

```
public int childrenRun( LassoTypeRef outValue );
```

runRequest()

Creates and runs a new LJAPI call on the given method (methodName of the className class). If there is already an active request on the current thread, the method will be run within the context of that thread. If there is no active request on the current thread, a new request will be created and run based on the global context. The tagAction parameter is passed to the methodName and can be used to signal or pass information to the function.

```
public static int runRequest( String className,
String methodName,
 int tagAction,
 int unused );
```

Output Methods

outputTagData()

Outputs any string data to the page. Lasso takes care of encoding, and this can be called as many times as needed. The second variant of this method is recommended for writing binary data.

```
public int outputTagData( String data );
```

```
public int outputTagData( byte[] data );
```

returnTagValue()

Specifies the return value for the tag. Note that only a single returnTagValue or outputTagData can be used from within a tag. returnTagValue is the prefered method for returning tag data as it allows data of any type to be returned (including binary data), while outputTagData is restricted to printable text data.

```
public int returnTagValue ( LassoTypeRef value );
```

Data Type Methods

typeAlloc()

This function will allocate a new type instance. The type is specified by the typeName parameter. An array of parameters can be passed to the type initializer. Types created through this function will be automatically destroyed after the LJAPI call has returned. In order to prevent this, typeDetach should be used.

```
public int typeAlloc ( String typeName,
        LassoTypeRef[] params,
        LassoTypeRef outType );
```

typeFree()

Attempts to free a type created using typeAlloc or any other method. The LassoCall variable may be null if the provided type has been detached using typeDetach.

```
public int typeFree ( LassoTypeRef inType );
```

typeDetach()

Prevents the type from being destroyed once the LJAPI call returns. Types that have been detached must eventually be destroyed using typeFree() (passing null in the LassoCall variable) or a memory leak will occur.

```
public int typeDetach( LassoTypeRef toDetach );
```

typeAllocNull()

This method allows new instances of LASSO_NULL data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocNull ( LassoTypeRef outNull);
```

typeAllocString()

This method allows new instances of string data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocString ( LassoTypeRef outString, String value);
```

typeAllocInteger()

This method allows new instances of integer data types to be allocated (Lasso integers are 8-byte signed INTs). Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocInteger ( LassoTypeRef outInteger, long value);
```

typeAllocDecimal()

This method allows new instances of decimal data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocDecimal ( LassoTypeRef outDecimal, double value);
```

typeAllocPair()

This method allows new instances of pair data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocPair ( LassoTypeRef outPair,
                LassoTypeRef inFirst,
                LassoTypeRef inSecond);
```

typeAllocReference()

This method allows new instances of reference data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
public int typeAllocReference ( LassoTypeRef outRef,
            LassoTypeRef referenced );
```

typeAllocTag()

This method allows new instances of tag data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned. Method methodName should have the same signature as TAG_METHOD_PROTOTYPE() method in the LassoTagModule class.

```
public int typeAllocTag ( LassoTypeRef outTag,
                String className,
                String methodName );
```

typeAllocArray()

This method allows new instances of array data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocArray ( LassoTypeRef outArray,
                        LassoTypeRef[] inElements);
```

typeAllocMap()

This method allows new instances of map data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

Two versions of the same method are provided: in the first case the count of elements of the inElements array must be divisible by 2 and contain both keys and values (odd = key, even = value). In the second case, map keys and values must be passed in a separate parameters.

```
public int typeAllocMap ( LassoTypeRef outMap,
        LassoTypeRef[] inElements );
public int typeAllocMap ( LassoTypeRef outMap,
        LassoTypeRef[] inKeys,
        LassoTypeRef[] inValues );
```

typeAllocBoolean()

This method allows new instances of boolean data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocBoolean( LassoTypeRef outBool, boolean inValue );
```

typeGetBytes()

This method returns the data of a type instance as an array of bytes.

```
public bytes[] typeGetString( LassoTypeRef type);
```

typeGetString()

This method gets the data from a previously created string instance. When setting a value, the type is converted if required.

```
public int typeGetString( LassoTypeRef type, LassoValue outValue );
```

typeGetInteger()

This method gets the data from a previously created integer instance. When setting a value, the type is converted if required.

```
public int typeGetInteger( LassoTypeRef type, IntValue outValue );
```

typeGetDecimal()

This method gets the data from a previously created decimal instance. When setting a value, the type is converted if required.

```
public int typeGetDecimal( LassoTypeRef type, FloatValue outValue );
```

typeGetBoolean()

This method gets the data from a previously created boolean instance. When setting a value, the type is converted if required.

```
public int typeGetBoolean( LassoTypeRef type, BoolValue outValue );
```

typeSetBytes()

This method sets the data of a type instance. The type is converted if required.

```
public int typeSetBytes( LassoTypeRef type, byte[] value );
```

typeSetString()

This method sets the value of a previously created string type instance.

```
public int typeSetString( LassoTypeRef type, String value );
```

typeSetInteger()

This method sets the value of a previously created integer instance.

```
public int typeSetInteger( LassoTypeRef type, long value );
```

typeSetDecimal()

This method sets the value of a previously created decimal instance.

```
public int typeSetDecimal( LassoTypeRef type, double value );
```

typeSetBoolean()

This method sets the value of a previously created boolean instance.

```
public int typeSetBoolean( LassoTypeRef type, boolean value );
```

arrayGetSize()

This method gets the size of a previously created array instance.

```
public int arrayGetSize( LassoTypeRef array, IntValue outLen );
```

arrayGetElement()

This method gets an array element from a previously created array instance.

```
public int arrayGetElement( LassoTypeRef array,
                            int index,
                            LassoTypeRef outElement );
```

arraySetElement()

This method sets an array element in a previously created array instance.

```
public int arraySetElement( LassoTypeRef array,
                            int index,
                            LassoTypeRef element );
```

arrayRemoveElement()

This method removes an element from a previously created array instance.

```
public int arrayRemoveElement( LassoTypeRef array, int index );
```

mapGetSize()

This method gets the size of a previously created map instance.

```
public int mapGetSize( LassoTypeRef map, IntValue outLen );
```

mapFindElement()

This method finds an element in a previously created map instance stored under unique key.

```
public int mapFindElement( LassoTypeRef map,
                           LassoTypeRef key,
                           LassoTypeRef outElement );
```

mapGetElement()

This method gets an element from a previously created map instance using the element index.

```
public int mapGetElement( LassoTypeRef map,
        int index,
        LassoTypeRef outPair );
```

mapSetElement()

This function sets an element in a previously created map instance. If no elements were previously stored under the specified key, the element will be added to the map, otherwise the old element will be replaced by a new value.

```
public int mapSetElement( LassoTypeRef map,
        LassoTypeRef key,
        LassoTypeRef value );
```

mapRemoveElement()

This method removes an element from a previously created map instance.

```
public int mapRemoveElement( LassoTypeRef map, LassoTypeRef key );
```

pairGetFirst()

This method gets the first element from a previously created pair instance.

```
public int pairGetFirst( LassoTypeRef pair, LassoTypeRef outValue );
```

pairGetSecond()

This method gets the second element from a previously created pair instance.

```
public int pairGetSecond( LassoTypeRef pair, LassoTypeRef outValue );
```

pairSetFirst()

This method sets the first element in a previously created pair instance.

```
public int pairSetFirst( LassoTypeRef pair, LassoTypeRef first );
```

pairSetSecond()

This function sets the second element in a previously created pair instance.

```
public int pairSetSecond( LassoTypeRef pair, LassoTypeRef second );
```

typeGetMember()

This function is used to retrieve a member from a type instance. Members are searched by name with tag members searched first. Data members are searched if no tag member is found with the given name.

```
public int typeGetMember( LassoTypeRef fromType,
                String named,
                LassoTypeRef outMember );
```

typeGetProperties()

This method has two uses. If the targetType parameter is not null, it is used to get all data and tag members from a given type. They are returned as a pair of arrays in the outPair value. The first element of each pair is the map of data members for the type. The second element is the map of tag members. Each element in the array represents the members of each type inherited by the targetType.

If the targetType parameter is null, typeGetProperties will return an array containing the variable maps for the currently active request.

```
public int typeGetProperties ( LassoTypeRef targetType,
                        LassoTypeRef outPair );
```

typeGetName()

Retrieves the name of the target type.

```
public int typeGetName( LassoTypeRef target, LassoValue outName );
```

typeRunTag()

Used to to execute a given tag. The tag can be run given a specific name and parameters, and the return value of the tag can be accessed. If the tag is a member tag, the instance of which it is a member can be passed using the final parameter. The params, returnValue, and optionalTarget parameters may all be null.

A slightly modified version of the same method is provided for convenience puproses. It accepts a single LassoTypeRef parameter instead of a LassoTypeRef array.

```
public int typeRunTag ( LassoTypeRef tagType,
                String named,
                LassoTypeRef[] params,
                LassoTypeRef returnValue,
                LassoTypeRef optionalTarget );

public int typeRunTag ( LassoTypeRef tagType,
                String named,
                LassoTypeRef parameter,
                LassoTypeRef returnValue,
                LassoTypeRef optionalTarget );
```

typeAssign()

This performs an assignment of one type to another. The result will be the same as if the following had been executed in LDML:
#left_hand_side = #right_hand_side

```
public int typeAssign( LassoTypeRef left_hand_side,
        LassoTypeRef right_hand_side);
```

typeStealValue()

This function transfers the data from one type to another type. Both types must be valid and pre-allocated. After the call, victim will still be valid, but will be of type null.

```
public int typeStealValue( LassoTypeRef thief, LassoTypeRef victim );
```

handleExternalConversion()

Converts a Lasso type into single-byte or binary data using the specific encoding name. The default for all database, column, table names should be "iso8859-1".

```
public byte[] handleExternalConversion(LassoTypeRef inInstance, String inEncoding);
```

handleInternalConversion()

Converts a single-byte or binary representation of a Lasso type back into an instance of that type.

```
public int handleInternalConversion( byte[] inData, String inEncoding, int
inClosestLassoType, LassoTypeRef outType );
```

typeInheritFrom

This function changes the inheritance structure of a type. Sets inNewParent to be the new parent of the child. Any parent that child currently has will be destroyed.

```
public int typeInheritFrom( LassoTypeRef inChild, LassoTypeRef inNewParent );
```

Custom Type Methods

typeAllocCustom()

This function is used within module methods that were registered as being a type initializer (FLAG_INITIALIZER). It initializes a blank custom type and sets the type's __type_name__ member to the provided value. The new type does not yet have a lineage and has no members added to it besides __type_name__. New data or tag members should be added using typeAddMember. The new custom type should be the return value of the type initializer. Any inherited members will be added to the type after the LJAPI call returns.

Warning: Do not call this unless you are in a type initializer. If you are not in a type initializer, the result will be a type that will never be fully initialized.

```
public int typeAllocCustom( LassoTypeRef outCustom, String name );
```

typeAddMember()

This is used to add new members to type instances. The member can be any sort of type including tags or other custom types.

```
public int typeAddMember( LassoTypeRef to,
                String named,
                LassoTypeRef member );
```

typeAllocFromProto()

Allocate a new type based on the given type. The given type's tag members will be referenced in the new type. No data members are added except for the typename member. Proto must be a custom type.

```
public int typeAllocFromProto( LassoTypeRef inProto, LassoTypeRef outType );
```

typeAllocOneOff()

Allocate a new type with the given name. The type does not have to have been registered as a type initializer or registered at all. The new type will have no tag or data members, but those may be added using the appropriate LCAPI call at any time. If no parent type is provided (a NULL pointer or empty string is passed in), type null will be the default. If a parent type is provided, it must have been a validly registered type initializer. onCreate will be called for the parent and beyond.

```
public int typeAllocOneOff( String inName, String inParentTypeName, LassoTypeRef
outType );
```

typeGetCustomJavaObject()

Custom types can have Java objects attached to them. The object can be retrieved at any point during the instance's lifetime. typeSetCustomJavaObject method retrieves the Java object associated with a custom type, or returns null if no object has been attached to this type.

```
public Object typeGetCustomJavaObject( LassoTypeRef type );
```

typeSetCustomJavaObject()

typeSetCustomJavaObject permits attachment of Java objects to custom types. Java object is retained until typeFreeCustomJavaObject is called, or the type is destroyed.

```
public int typeSetCustomJavaObject( LassoTypeRef type, Object object );
```

typeFreeCustomJavaObject()

Releases the Java object previously attached to a custom type. Must be called to free the Java object that is no longer needed, or to detach an old Java object before attaching a new one to the same custom type.

```
public int typeFreeCustomJavaObject( LassoTypeRef type );
```

Logging Function Methods

log()

Logs a message. The message goes to the prefered destination for the message level. Messages sent to a file are limited to 2048 bytes in length. Messages sent to the console are limited to 512 bytes in length. Messages sent to the database are limited a little less than 2048 bytes since the total length of the sql statement used to insert the message is limited to 2048 bytes. The msgLevel parameter must be one of the following: LOG_LEVEL_CRITICAL, LOG_LEVEL_WARNING, or LOG_LEVEL_DETAIL.

```
public static int log ( int msgLevel, String message );
```

logSetDestination()

Changes the system-wide log destination preference. You can log messages to more than one destination at a time by passing several flags in the destination parameter: FLAG_DEST_CONSOLE, FLAG_DEST_FILE, and/or FLAG_DEST_DATABASE.

```
public static int logSetDestination( int msgLevel, int destination );
```

MIME Header Methods

getResultHeader()

Retrieves current value of the result (HTTP) header. Part of the header that is returned to browsers is automatically built by Lasso, and can be modified or added to by LDML tags on the page. This function retrieves the current set of MIME headers that would be sent back to the browser if page processing were to stop now.

```
public int getResultHeader( LassoValue result );
```

setResultHeader()

Sets the result header, any data will be validated so as to be in the proper format.

```
public int setResultHeader( String header );
```

addResultHeader()

Simply appends the supplied data to the header, any data will be validated so as to be in the proper format.

```
public int addResultHeader( String data );
```

getCookieValue()

Retrieves a cookie value from the passed-in data sent by the client browser.

```
public int getCookieValue( String named, LassoValue value );
```

Page Variable Methods

getVariableCount()

Retrieves the number of array values which the named global variable has. Returns 1 if the global variable is not an array. Global variables are the same variables which you create in LDML statements, like [var: 'fred'=1234.56]. These variables last only as long as the current format file is executing; as soon as the hit gets sent back to the browser, these variables all get destroyed.

```
public int getVariableCount( String named, IntValue count );
```

getVariable()

Retrieves the value of the named global variable. If the global variable is an array, then the index specifies which array value to retrieve. If the global variable is not an array, then 0 is the only valid index. Array indices start at 0.

```
public int getVariable( String named, int index, LassoValue value );
```

getVariable2()

Retrieves the value of the named global variable while preserving the variable type.

```
public int getVariable2( String named, LassoTypeRef outValue );
```

setVariable()

Stores a new value into the named global variable. If the global variable is an array, then the 0-based index determines which array item to replace.

```
public int setVariable( String named, String value, int index );
```

setVariable2()

Stores a new global variable while preserving the type.

```
public int setVariable2( String named, LassoTypeRef inValue );
```

removeVariable()

Removes the specified variable (destroys it so it becomes undefined, as though it had never been created). If the named variable is an array, then you may pass in an index (0-based) to remove that array element. Once the array has 0 elements, then calling removeVariable on it will destroy the array itself.

```
public int removeVariable( String named, int index );
```

LDML Tag Interpreter Methods

formatBuffer()

Formats the supplied buffer and puts the resulting data in the data field of the LassoValue. The buffer should consist of plain text and bracketed Lasso tags.

```
public int formatBuffer( String buffer, LassoValue output );
```

Persistent Storage Tag Methods

storeHasData()

Returns ERR_NOERR if the data, specified by key, exists. The length of the stored data can be returned in the outLength parameter if you pass a valid IntValue object. You may pass null if you don't want to retrieve the length of the stored data.

```
public int storeHasData( String key, IntValue outLength );
```

storeGetData()

Fetches data that has been stored under the unique identifier key. The data will be returned in the data field of the LassoValue object.

```
public int storeGetData( String key, LassoValue outValue );
```

storePutData()

Adds the data to Lasso's storage. Key is the unique identifier for the data.

```
public int storePutData( String key, String data );
public int storePutData( String key, byte[] data );
```

Administration Methods

isAdministrator()

Returns ERR_NOERR if the current user has administrator privileges. This is useful for doing module administration that only the administrator should be able to do.

```
public int isAdministrator( );
```

Data Source Function Methods

getDSConnection()

This function accesses the current datasource connection.

```
public Object getDSConnection();
```

setDSConnection()

This function sets the current connection for the data source. May recurse to deliver the ACTION_CLOSE message if there is already a valid connection set.

```
public int setDSConnection( Object inConnection );
```

addDataSourceResult()

Sometimes Lasso Professional will query a data source function to return information, such as a list of database names or table names which the data source module controls. The module will call this function once for each name you add to the list, so if you have three database names you want to report back to Lasso Professional, you would call this function three times, once per database name.

```
public int addDataSourceResult( String data );
```

getDataSourceName()

Use this function when you want to ask Lasso Professional what database is being operated on. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the database which Lasso Professional is asking you to search. It corresponds to the value of the parameter -Database='blah' passed to inlines. Optionally, you can use the second (outUseHostDefault) parameter to determine whether the current database inherits its host default settings.

Note: Even though the name of the method is getDataSourceName, it really retrieves the database name. This is purely cosmetic, and just happens to be how the APIs were spelled when they were originally designed.

```
public int getDataSourceName( LassoValue outName,
                              BoolValue  outUseHostDefault,
                              LassoValue outUsernamePassword );
```

getDataHost()

Use this function when you want to ask Lasso Professional 7 what database host is being operated on. On return, LassoValue will contain the name and port of the database host.

```
public int getDataHost( LassoValue outHost,
                        LassoValue outUsernamePassword );
```

getDataHost2()

Same as getDataHost() but allows the usage of a host schema parameter for JDBC data sources.

```
public int getDataHost2( LassoValue outHost,
                         LassoValue outSchema,
                         LassoValue outUsernamePassword );
```

getSchemaName()

Use this function when you want to ask Lasso Professional what schema is being operated on for a JDBC data source. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the schema which Lasso Professional is asking you to use for the search. It corresponds to the value of the parameter -Schema='blah' passed to inlines.

```
public int getSchemaName( LassoValue outName );
```

getTableName()

Use this function when you want to ask Lasso Professional what table is being operated on. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the table which Lasso Professional is asking you to search. It corresponds to the value of the parameter -Layout='blah' or -Table='blah' passed to inlines.

```
public int getTableName( LassoValue outName );
```

getSkipRows()

You can ask Lasso Professional to tell you how many records should be skipped during a search by calling this function. It corresponds to the value of the -SkipRecords parameter in the inline search which is being executed at the moment your data source function is being called.

```
public int getSkipRows( IntValue outRows );
```

getMaxRows()

You can ask Lasso Professional to tell you the maximum number of records to be returned during a search by calling this function. It corresponds to the value of the -MaxRecords parameter in the inline search which is being executed at the moment your data source function is being called.

```
public int getMaxRows( IntValue outRows );
```

getPrimaryKeyColumn()

You can ask Lasso Professional to tell you which field is being used as the primary key. This value corresponds to the -KeyField parameter value used in the inline.

```
public int getPrimaryKeyColumn( LassoValue outColumn );
```

getInputColumnCount()

Tells how many fields were sent as parameters to the inline. For instance, if an LDML programmer wants to append a new record to a table, and passes in name, address, city, state, zip with values for each field, then this function will return the number 5 to indicate that five fields were passed to the inline. You can then retrieve the values of each of these parameters by calling getInputColumn by index, once per field. This function is smart

enough to ignore parameters which are not fields, such as -Database, -Layout, etc.

```
public int getInputColumnCount( IntValue outCount );
```

getInputColumn()

Retrieve the name and value of field data parameters from the inline, starting at index zero. If five fields were entered into the inline, then you can retrieve each of their names and values by calling this function five times, once per field.

```
[Inline: -Database='MyDatabase', -Table='Main', 'MyFirstField'='Bill',
'MySecondField'='Ted', -Search]
```

In the above example, calling getInputColumn(0, v) will fill the v variable with v.name=MyFirstField, v.data=Bill. Notice it is smart enough to ignore well-known parameters such as -Table, thus only retrieving field information.

```
public int getInputColumn( int index, LassoValue outColumn );
```

getSortColumnCount()

Analogous to getInputColumnCount, this method retrieves the number of sort columns which were specified in the inline code. It basically counts how many -SortField parameters were passed. You can use this count to tell you how many times to enumerate through calls to getSortColumn.

```
public int getSortColumnCount( IntValue outCount );
```

getSortColumn()

Analogous to lasso_getInputColumn(), this function retrieves the names of sort parameters, starting at index zero. After calling this, the data field of outColumn variable will contain a String with the name of the sort field.

```
public int getSortColumn( int index, LassoValue outColumn );
```

getRowID()

Retrieves the current specified record ID (datasource-specific).

```
public int getRowID( IntValue outId );
```

setRowID()

Sets the record ID of the added record. After your custom LCAPI data source finishes adding a record to a database, it can call this function to let the caller know what the unique record ID of the added record was.

In FileMaker, this record ID is a standard feature of all records in its tables. In MySQL, this value is 0 unless there exists an AUTO_INCREMENT column. Results are not guaranteed for all database server software.

```
public int setRowID( int id );
```

findInputColumn()

Analogous to getInputColumn, except that it searches by name instead of index. If you already know the name of a field parameter you're interested in, then you can ask for the value of that parameter which was passed into the inline.

```
[Inline: -Database='MyDatabase', -Table='Main', 'MyFirstField'='Bill',
'MySecondField'='Ted', -Search]
```

In the example above, calling findInputColumn("MySecondField", outColumn) will fill the outColumn variable's data member with v.data=Ted.

```
public int findInputColumn( String name, LassoValue outColumn );
```

getLogicalOp()

Call this to retrieve the logical operator (OP_AND, OP_OR) which was passed to this inline. It corresponds to the value of -LogicalOperator passed into the inline. This function simply retrieves a single logical operator parameter. For more complex logical operations, with multiple operators, you will have to design a convention whereby you name your input fields in some unique way, and then retrieve those custom logical operators using the getInputColumn function in a particular order that matches your convention.

```
public int getLogicalOp( IntValue outOp );
```

getReturnColumnCount()

Queries Lasso Professional to return the number of columns (fields) that are expected to be returned from a search operation. This counts how many -ReturnField parameters were encountered.

```
public int getReturnColumnCount( IntValue outCount );
```

getReturnColumn()

Once you know how many return columns are expected (from getReturnColumnCount), then you can enumerate through them to get their fieldnames. Use this information to retrieve field data from your database table, and populate the result rows when asked to perform a search operation.

```
public int getReturnColumn( int index, LassoValue outColumn );
```

addColumnInfo()

In order to return a row of data from your data source (perhaps as a result of a search), you must first indicate what the structure of the table columns is. Call this function for as many table columns as your database has, providing the fieldname, true/false if nulls are OK in this field, the field type (numeric, string, date, etc), and field protection (readonly, writeable, etc).

```
public int addColumnInfo( String name,
        int nullOK,
        int type,
        int protection );
```

addResultRow()

Call this method once per row of records you want to return (perhaps from a search operation). You may choose to return an array of Strings, or construct an array of byte arrays that contain data for each of your fields (binary data is OK).

```
public int addResultRow( String[] columns );
public int addResultRow( byte[][] columns );
```

setNumRowsFound()

Corresponds to [Found_Count] in LDML. Call this when you know how many records your data source is going to return, and make sure you call addResultRow this many times in order to populate the rows.

```
public int setNumRowsFound( int num );
```

Semaphore Methods

createSem()

Creates a named semaphore sufficient for synchronizing multithreaded operations, which should be deleted after they are used. The Lasso Connector for MySQL example creates one of these at initialization time, and destroys it at terminate time.

```
public int createSem( String name );
```

destroySem()

Destroys a named semaphore that was created by the createSem method.

```
public int destroySem( String name );
```

acquireSem()

Attempts to acquire a lock on a semaphore, and waits until the owning thread has released the semaphore before acquiring the lock and continuing execution.

```
public int acquireSem( String name );
```

releaseSem()

Releases a locked semaphore so that other threads waiting for the semaphore can continue execution.

```
public int releaseSem( String name );
```

com.blueworld.lassopro.LassoDSModule

Base class for all datasource modules. LassoDSModules are used to manipulate data sources. LassoDSModules are looked up by the datasource names they claim to support. They are instantiated once and used repeatedly by Lasso.

registerDSModule()

Your code must call this once at startup (from within your registerLassoModule() method) to register a data source with Lasso Professional. When Lasso encounters a data source request for moduleName, it calls the Java method methodName.

```
protected void registerDSModule( String datasourceName,
                                 String methodName,
                                 int flags,
                                 String moduleName,
                                 String description);
```

DS_METHOD_PROTOTYPE()

A prototype for all datasource action methods registered by registerDSModule. Since methods are being looked up by name, they must match exactly the values passed in a methodName parameter of the registerDSModule call.

```
public int DS_METHOD_PROTOTYPE( LassoCall lasso,
                                int action,
                                LassoValue data );
```

com.blueworld.lassopro.LassoEncodings

Constants for the various text encoding methods.

ENCODE_BREAK

Static variable in class blueworld.lasso.LassoTagEncodings.

```
public static final int ENCODE_BREAK
```

ENCODE_DEFAULT

Static variable in class com.blueworld.lassopro.LassoEncodings.

```
public static final int ENCODE_DEFAULT
```

ENCODE_NONE

Static variable in class com.blueworld.lassopro.LassoEncodings.

```
public static final int ENCODE_NONE
```

ENCODE_RAW

Static variable in class com.blueworld.lassopro.LassoEncodings.

```
public static final int ENCODE_RAW
```

ENCODE_SMART

Static variable in class com.blueworld.lassopro.LassoEncodings.

public static final int ENCODE_SMART

ENCODE_STRICT_URL

Static variable in class com.blueworld.lassopro.LassoEncodings.

public static final int ENCODE_STRICT_URL

ENCODE_URL

Static variable in class com.blueworld.lassopro.LassoEncodings.

public static final int ENCODE_URL

ENCODE_XML

Static variable in class com.blueworld.lassopro.LassoEncodings.

public static final int ENCODE_XML

com.blueworld.lassopro.LassoErrors

Constants for the various error codes which can be returned by your module.

NO_ERR

Static variable in class com.blueworld.lassopro.LassoErrors.

public static final int NO_ERR

Assert

Static variable in class com.blueworld.lassopro.LassoErrors.

public static final int Assert

StreamReadError

Could not write to stream.

public static final int StreamReadError

StreamWriteError

Could not read from stream.

public static final int StreamWriteError

Memory

Generic memory error.

public static final int Memory

InvalidMemoryObject

Invalid memory object.

public static final int InvalidMemoryObject

OutOfMemory

Not enough memory.

public static final int OutOfMemory

OutOfStackSpace

Stack overflow error.

public static final int OutOfStackSpace

CouldNotDisposeMemory

Error disposing an object.

public static final int CouldNotDisposeMemory

File

Generic file error.

public static final int File

FileInvalid

Trying to work with an invalid file.

public static final int FileInvalid

FileInvalidAccessMode

Trying to access a file in a mode that it doesn't support.

public static final int FileInvalidAccessMode

CouldNotCreateOrOpenFile

Could not create or open the file.

public static final int CouldNotCreateOrOpenFile

CouldNotCloseFile

Could not close the file.

public static final int CouldNotCloseFile

CouldNotDeleteFile

Could not delete the file.

public static final int CouldNotDeleteFile

FileNotFound

File does not exist.

public static final int FileNotFound

FileAlreadyExists

Trying to create a file that already exist.

public static final int FileAlreadyExists

FileCorrupt

File is corrupted.

public static final int FileCorrupt

VolumeDoesNotExist

Bad volume name.

public static final int VolumeDoesNotExist

DiskFull

No room left on disk.

public static final int DiskFull

DirectoryFull

No more items allowed in the directory.

public static final int DirectoryFull

IOError

I/O error.

public static final int IOError

InvalidPathname

Pathname is invalid.

public static final int InvalidPathname

InvalidFilename

Filename is invalid.

public static final int InvalidFilename

FileLocked

File is locked.

public static final int FileLocked

FileUnlocked

File is unlocked.

public static final int FileUnlocked

FileIsOpen

File is open.

public static final int FileIsOpen

FileIsClosed

File is closed.

public static final int FileIsClosed

BOF

Beginning of file reached.

public static final int BOF

EOF

End of file reached.

public static final int EOF

CouldNotWriteToFile

Unable to complete a write operation to the file.

public static final int CouldNotWriteToFile

CouldNotReadFromFile

Unable to complete a read operation from the file.

public static final int CouldNotReadFromFile

Resource

Unknown resource error.

public static final int Resource

ResNotFound

Resource not found.

public static final int ResNotFound

Network

Unknown networking error.

public static final int Network

InvalidUsername

The username supplied for the action is not valid.

public static final int InvalidUsername

InvalidPassword

The password supplied for the action is not valid.

public static final int InvalidPassword

InvalidDatabase

The database name supplied is not valid.

public static final int InvalidDatabase

NoPermission

General permissions error.

public static final int NoPermission

FieldRestriction

The specified action is restricted.

public static final int FieldRestriction

WebAddError

Add record error.

public static final int WebAddError

WebUpdateError

Update record error.

public static final int WebUpdateError

WebDeleteError

Delete record error.

public static final int WebDeleteError

InvalidParameter

An invalid parameter was passed to a function.

public static final int InvalidParameter

Overflow

Allocated memory was too small to hold the results.

public static final int Overflow

NilPointer

A pointer was null when it shouldn't have been.

public static final int NilPointer

UnknownError

Default when none of the cross-platform errors seem to fit.

public static final int UnknownError

FormattingLoopAborted

A looping tag was aborted; all looping tags must catch this exception.

public static final int FormattingLoopAborted

FormattingSyntaxError

Bad syntax used in a format file; parsing of the file was aborted.

public static final int FormattingSyntaxError

WebRequiredFieldMissing

Value missing for required field for Add.

public static final int WebRequiredFieldMissing

WebRepeatingRelatedField

Adding repeating related fields isn't supported.

public static final int WebRepeatingRelatedField

WebNoSuchObject

No records found.

public static final int WebNoSuchObject

WebTimeout

Operation timed out.

public static final int WebTimeout

WebActionNotSupported

Action not supported.

public static final int WebActionNotSupported

WebConnectionInvalid

The specified database was not found.

public static final int WebConnectionInvalid

WebModuleNotFound

The module was not found.

public static final int WebModuleNotFound

HTTPFileNotFound

The file was not found.

public static final int HTTPFileNotFound

DatasourceError

Third-party generic datasource error.

public static final int DatasourceError

com.blueworld.lassopro.LassoOperators

Operator constants used throughout LJAPI.

Variables

OP_AND

Logical operator AND.

public static final int OP_AND

OP_ANY

Used for -Random database action.

public static final int OP_ANY

OP_BEGINS_WITH

Field search operator BW.

public static final int OP_BEGINS_WITH

OP_CONTAINS

Field search operator CN.

public static final int OP_CONTAINS

OP_DEFAULT

Same as OP_BEGINS_WITH.

public static final int OP_DEFAULT

OP_ENDS_WITH

Field search operator EW.

public static final int OP_ENDS_WITH

OP_EQUALS

Field search operator EQ.

public static final int OP_EQUALS

OP_GREATER_THAN

Field search operator GT.

public static final int OP_GREATER_THAN

OP_GREATER_THAN_EQUALS

Field search operator GTE.

public static final int OP_GREATER_THAN_EQUALS

OP_IN_FULL_TEXT

Field search operator FT.

public static final int OP_IN_FULL_TEXT

OP_IN_LIST

Static variable in class com.blueworld.lassopro.LassoOperators.

public static final int OP_IN_LIST

OP_IN_REGEXP

Field search operator RX.

public static final int OP_IN_REGEXP

OP_LESS_THAN

Field search operator LT.

public static final int OP_LESS_THAN

OP_LESS_THAN_EQUALS

Field search operator LTE.

public static final int OP_LESS_THAN_EQUALS

OP_NO

Same as OP_NOT.

public static final int OP_NO

OP_NOT

Logical operator NOT.

public static final int OP_NOT

OP_NOT_BEGINS_WITH

Field search operator NBW.

public static final int OP_NOT_BEGINS_WITH

OP_NOT_CONTAINS

Field search operator NCN.

public static final int OP_NOT_CONTAINS

OP_NOT_ENDS_WITH

Field search operator NEW.

public static final int OP_NOT_ENDS_WITH

OP_NOT_EQUALS

Field search operator NEQ.

public static final int OP_NOT_EQUALS

OP_NOT_IN_LIST

Static variable in class com.blueworld.lassopro.LassoOperators.

public static final int OP_NOT_IN_LIST

OP_NOT_IN_REGEXP

Field search operator NRX.

public static final int OP_NOT_IN_REGEXP

OP_OR

Logical operator OR.

public static final int OP_OR

com.blueworld.lassopro.LassoParams

These constants signify the different parameters which can be retrieved from the LassoCall.getLassoParam method.

ModulesFolderPath

Path to the LassoModules folder.

public static final int ModulesFolderPath

StartupItemsFolderPath

Path to LassoStartup folder.

public static final int StartupItemsFolderPath

LassoErrorsFilePath

Path to Lasso error log file.

public static final int LassoErrorsFilePath

StorageHost

Location of Lasso MySQL datasource.

public static final int StorageHost

ScriptsRoot

Relative path to scripts root.

public static final int ScriptsRoot

ScriptsSiteRoot

Relative path to site scripts root (most likely includes ScriptsRoot).

public static final int ScriptsSiteRoot

com.blueworld.lassopro.LassoTagModule

Base class for any tag module. Most tag modules output data onto the Web page, though some tags may perform other actions based on the parameters passed to them.

Every LassoTagModule must implement registerLassoModule method, and one or more methods with the same signature as TAG_METHOD_PROTOTYPE.

Lasso calls registerLassoModule once at startup to give the module a chance to register its tags. LassoTagModule must then call registerTagModule as many times as there are tags implemented by this module.

Variables

FLAG_INITIALIZER

Type initializer tags can have their own members.

public static final int FLAG_INITIALIZER

FLAG_SUBSTITUTION

Regular substitution tags.

public static final int FLAG_SUBSTITUTION

FLAG_ASYNC

Async tags run asynchronously in their own thread.

public static final int FLAG_ASYNC

FLAG_CONTAINER

Container tags have opening and closing. This flag will cause Lasso Professional to raise an error if the closing tag can't be found.

public static final int FLAG_CONTAINER

Methods

registerTagModule()

Use this method to register substitution tags implemented by your module. You should call registerTagModule as many times as there are tags implemented in your module.

moduleName parameter is the name of the module as returned by [Lasso_TagModuleName] LDML tag. tagName is the name of the custom LDML tag implemented by this module. One or more OR logical FLAG constants can be passed in the flags parameter to specify unique tag features. Finally, a description parameter can be used to provide optional tag info, such as brief description of the tag usage.

```
protected void registerTagModule( String moduleName,
                                  String tagName,
                                  String methodName,
                                  int flags,
                                  String description);
```

com.blueworld.lassopro.LassoTypeRef

This class is used for creating and manipulating custom Lasso types. Unlike LassoValue or IntValue objects which store copies of the data, LassoTypeRef is merely a reference to a native object instance. Native objects exist for a fraction of a second while Lasso is processing a page, therefore the LassoTypeRef objects should never be stored or reused across multiple module invocations.

Variables

LASSO_ARRAY

The name of the built-in array type in Lasso Professional 7.

```
public static final String LASSO_ARRAY
```

LASSO_BOOLEAN

The name of the built-in boolean type in Lasso Professional 7.

```
public static final String LASSO_BOOLEAN
```

LASSO_DATE

The name of the built-in date type in Lasso Professional 7.

```
public static final String LASSO_DATE
```

LASSO_DECIMAL

The name of the built-in decimal type in Lasso Professional 7.

```
public static final String LASSO_DECIMAL
```

LASSO_INTEGER

The name of the built-in integer type in Lasso Professional 7.

```
public static final String LASSO_INTEGER
```

LASSO_MAP

The name of the built-in map type in Lasso Professional 7.

```
public static final String LASSO_MAP
```

LASSO_NULL

The name of the built-in null type in Lasso Professional 7.

```
public static final String LASSO_NULL
```

LASSO_PAIR

The name of the built-in pair type in Lasso Professional 7.

```
public static final String LASSO_PAIR
```

LASSO_STRING

The name of the built-in string type in Lasso Professional 7.

```
public static final String LASSO_STRING
```

LASSO_TAG

The name of the built-in tag type in Lasso Professional 7.

```
public static final String LASSO_TAG
```

Methods

isNull()

Returns true if this object does not refer to a valid type instance, which most likely would be a result of a failed LassoCall method.

```
public boolean isNull();
```

toString()

Returns string representation of the LassoTypeRef object. Overrides toString method in the class Object.

```
public String toString();
```

com.blueworld.lassopro.LassoValue

Used for retrieving values from various LassoCall methods. Has name and data member variables of type String. The type member is set to one of the TYPE constants, reflecting the original type of the value before it was converted to string.

Variables

TYPE_ARRAY

Array type.

```
public static final int TYPE_ARRAY
```

TYPE_BLOB

Binary data.

```
public static final int TYPE_BLOB
```

TYPE_BOOLEAN

Boolean type.

```
public static final int TYPE_BOOLEAN
```

TYPE_CHAR

String type.

```
public static final int TYPE_CHAR
```

TYPE_CODE

Substitution tag code.

```
public static final int TYPE_CODE
```

TYPE_CUSTOM

Custom type.

```
public static final int TYPE_CUSTOM
```

TYPE_DATETIME

Date type.

```
public static final int TYPE_DATETIME
```

TYPE_DECIMAL

Decimal type.

```
public static final int TYPE_DECIMAL
```

TYPE_INT

Integer type.

```
public static final int TYPE_INT
```

TYPE_MAP

Map type.

```
public static final int TYPE_MAP
```

TYPE_NULL

Null type.

```
public static final int TYPE_NULL
```

TYPE_PAIR

Pair type.

```
public static final int TYPE_PAIR
```

TYPE_REFERENCE

Reference type.

```
public static final int TYPE_REFERENCE
```

Constructors

```
public LassoValue();
public LassoValue(int type);
public LassoValue(String data);
public LassoValue(String name, String data);
public LassoValue(String name, String data, int type);
```

Methods

data()

Returns the String object stored in the data field.

```
public String data();
```

name()

Returns the String object stored in the name field.

```
public String name();
```

setData()

Sets the value of the data field.

```
public String setData(String data);
```

setName()

Sets the value of the name field.

```
public String setName(String name);
```

setType()

Sets the value of the type field.

```
public int setType(int type);
```

toString()

Converts this object to String.

```
public String toString()
```

type()

Returns the original type of the data retrieved from one of the LassoCall methods: TYPE_CHAR for strings, TYPE_INT for integers, and so on.

For unnamed tag parameters, the type field is set to the type of the data stored in the data field. For named tag parameters, it reflects the type of the value member.

```
public int type();
```

com.blueworld.lassopro.RequestParams

These constants signify the different parameters which can be retrieved from the LassoCall.getRequestParam method.

AddressKeyword

IP address of client browser.

```
public static final int AddressKeyword
```

ActionKeyword

Type of HTTP request (GET, POST, etc.).

```
public static final int ActionKeyword
```

ClientIPAddress

IP address of client browser.

```
public static final int ClientIPAddress
```

ContentLength

The length in bytes of the POST data sent from <form POST>.

```
public static final int ContentLength
```

ContentType

MIME header sent from client browser.

public static final int ContentType

FullRequestKeyword

All MIME headers, uninterpreted.

public static final int FullRequestKeyword

MethodKeyword

GET or POST, depending on <form method>.

public static final int MethodKeyword

PasswordKeyword

Password sent from browser.

public static final int PasswordKeyword

PostKeyword

HTTP object body (form data, etc.).

public static final int PostKeyword

ReferrerKeyword

URL of referring page.

public static final int ReferrerKeyword

ScriptName

Relative path from server root to a Lasso format file.

public static final int ScriptName

SearchArgKeyword

All text in URL after the question mark.

public static final int SearchArgKeyword

ServerName

IP address or host name of the server on which the Web server is running.

public static final int ServerName

ServerPort

IP port this hit came to (80 is common, 443 for SSL).

public static final int ServerPort

UserAgentKeyword

Browser name and type.

public static final int UserAgentKeyword

UserKeyword

Username sent from browser.

public static final int UserKeyword

A

Appendix A
Extending Lasso Copyright Notice

B

Appendix B
Index

SYMBOLS

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

V

W

X